CHAPTER

TWENTYSIX

GRAPHICAL USER INTERFACES WITH TK

Tk/Tcl has long been an integral part of Python. It provides a robust and platform independent windowing toolkit, that is available to Python programmers using the *tkinter* package, and its extension, the *tkinter.tix* and the *tkinter.ttk* modules.

The *tkinter* package is a thin object-oriented layer on top of Tcl/Tk. To use *tkinter*, you don't need to write Tcl code, but you will need to consult the Tk documentation, and occasionally the Tcl documentation. *tkinter* is a set of wrappers that implement the Tk widgets as Python classes. In addition, the internal module `_tkinter` provides a threadsafe mechanism which allows Python and Tcl to interact.

tkinter's chief virtues are that it is fast, and that it usually comes bundled with Python. Although its standard documentation is weak, good material is available, which includes: references, tutorials, a book and others. *tkinter* is also famous for having an outdated look and feel, which has been vastly improved in Tk 8.5. Nevertheless, there are many other GUI libraries that you could be interested in. For more information about alternatives, see the *Other Graphical User Interface Packages* section.

26.1 `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 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 Rozerman 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.

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.

26.1.1 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.

26.1.2 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
```

(continues on next page)

(continued from previous page)

```
        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()
```

26.1.3 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"
   ^       ^     \_____________________/
   |       |                |
```

(continues on next page)

(continued from previous page)

```
 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.

26.1.4 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 hyphened 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")
```

26.1.5 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.

26.1.6 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" `"#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 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
%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.

26.1.7 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.

26.2 `tkinter.ttk` — Tk themed widgets

Source code: Lib/tkinter/ttk.py

The *tkinter.ttk* module provides access to the Tk themed widget set, introduced in Tk 8.5. If Python has not been compiled against Tk 8.5, this module can still be accessed if *Tile* has been installed. The former method using Tk 8.5 provides additional benefits including anti-aliased font rendering under X11 and window transparency (requiring a composition window manager on X11).

The basic idea for *tkinter.ttk* is to separate, to the extent possible, the code implementing a widget's behavior from the code implementing its appearance.

See also:

Tk Widget Styling Support A document introducing theming support for Tk

26.2.1 Using Ttk

To start using Ttk, import its module:

```
from tkinter import ttk
```

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

```
from tkinter import *
from tkinter.ttk import *
```

That code causes several *tkinter.ttk* widgets (`Button`, `Checkbutton`, `Entry`, `Frame`, `Label`, `LabelFrame`, `Menubutton`, `PanedWindow`, `Radiobutton`, `Scale` and `Scrollbar`) to automatically replace the Tk widgets.

This has the direct benefit of using the 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.

See also:

Converting existing applications to use Tile widgets A monograph (using Tcl terminology) about differences typically encountered when moving applications to use the new widgets.

26.2.2 Ttk Widgets

Ttk comes with 18 widgets, twelve of which already existed in tkinter: `Button`, `Checkbutton`, `Entry`, `Frame`, `Label`, `LabelFrame`, `Menubutton`, `PanedWindow`, `Radiobutton`, `Scale`, `Scrollbar`, and *Spinbox*. The other six are new: *Combobox*, *Notebook*, *Progressbar*, `Separator`, `Sizegrip` and *Treeview*. And all them are subclasses of *Widget*.

Using the Ttk widgets gives the application an improved look and feel. As discussed above, there are differences in how the styling is coded.

Tk code:

```
l1 = tkinter.Label(text="Test", fg="black", bg="white")
l2 = tkinter.Label(text="Test", fg="black", bg="white")
```

Ttk code:

```
style = ttk.Style()
style.configure("BW.TLabel", foreground="black", background="white")

l1 = ttk.Label(text="Test", style="BW.TLabel")
l2 = ttk.Label(text="Test", style="BW.TLabel")
```

For more information about *TtkStyling*, see the *Style* class documentation.

26.2.3 Widget

`ttk.Widget` defines standard options and methods supported by Tk themed widgets and is not supposed to be directly instantiated.

Standard Options

All the `ttk` Widgets accepts the following options:

Option	Description
class	Specifies the window class. The class is used when querying the option database for the window's other options, to determine the default bindtags for the window, and to select the widget's default layout and style. This option is read-only, and may only be specified when the window is created.
cursor	Specifies the mouse cursor to be used for the widget. If set to the empty string (the default), the cursor is inherited for the parent widget.
takefocus	Determines whether the window accepts the focus during keyboard traversal. 0, 1 or an empty string is returned. If 0 is returned, it means that the window should be skipped entirely during keyboard traversal. If 1, it means that the window should receive the input focus as long as it is viewable. And an empty string means that the traversal scripts make the decision about whether or not to focus on the window.
style	May be used to specify a custom widget style.

Scrollable Widget Options

The following options are supported by widgets that are controlled by a scrollbar.

Option	Description
xscrollcommand	Used to communicate with horizontal scrollbars. When the view in the widget's window change, the widget will generate a Tcl command based on the scrollcommand. Usually this option consists of the method `Scrollbar.set()` of some scrollbar. This will cause the scrollbar to be updated whenever the view in the window changes.
yscrollcommand	Used to communicate with vertical scrollbars. For some more information, see above.

Label Options

The following options are supported by labels, buttons and other button-like widgets.

Option	Description
text	Specifies a text string to be displayed inside the widget.
textvariable	Specifies a name whose value will be used in place of the text option resource.
underline	If set, specifies the index (0-based) of a character to underline in the text string. The underline character is used for mnemonic activation.
image	Specifies an image to display. This is a list of 1 or more elements. The first element is the default image name. The rest of the list if a sequence of statespec/value pairs as defined by *Style.map()*, specifying different images to use when the widget is in a particular state or a combination of states. All images in the list should have the same size.
compound	Specifies how to display the image relative to the text, in the case both text and images options are present. Valid values are: • text: display text only • image: display image only • top, bottom, left, right: display image above, below, left of, or right of the text, respectively. • none: the default. display the image if present, otherwise the text.
width	If greater than zero, specifies how much space, in character widths, to allocate for the text label, if less than zero, specifies a minimum width. If zero or unspecified, the natural width of the text label is used.

Compatibility Options

Option	Description
state	May be set to "normal" or "disabled" to control the "disabled" state bit. This is a write-only option: setting it changes the widget state, but the *Widget.state()* method does not affect this option.

Widget States

The widget state is a bitmap of independent state flags.

Flag	Description
active	The mouse cursor is over the widget and pressing a mouse button will cause some action to occur
disabled	Widget is disabled under program control
focus	Widget has keyboard focus
pressed	Widget is being pressed
selected	"On", "true", or "current" for things like Checkbuttons and radiobuttons
background	Windows and Mac have a notion of an "active" or foreground window. The *background* state is set for widgets in a background window, and cleared for those in the foreground window
readonly	Widget should not allow user modification
alternate	A widget-specific alternate display format
invalid	The widget's value is invalid

A state specification is a sequence of state names, optionally prefixed with an exclamation point indicating that the bit is off.

ttk.Widget

Besides the methods described below, the `ttk.Widget` supports the methods `tkinter.Widget.cget()` and `tkinter.Widget.configure()`.

class `tkinter.ttk.Widget`

identify(*x*, *y*)
Returns the name of the element at position *x* *y*, or the empty string if the point does not lie within any element.

x and *y* are pixel coordinates relative to the widget.

instate(*statespec*, *callback=None*, **args*, ***kw*)
Test the widget's state. If a callback is not specified, returns `True` if the widget state matches *statespec* and `False` otherwise. If callback is specified then it is called with args if widget state matches *statespec*.

state(*statespec=None*)
Modify or inquire widget state. If *statespec* is specified, sets the widget state according to it and return a new *statespec* indicating which flags were changed. If *statespec* is not specified, returns the currently-enabled state flags.

statespec will usually be a list or a tuple.

26.2.4 Combobox

The `ttk.Combobox` widget combines a text field with a pop-down list of values. This widget is a subclass of `Entry`.

Besides the methods inherited from *Widget*: `Widget.cget()`, `Widget.configure()`, *Widget.identify()*, *Widget.instate()* and *Widget.state()*, and the following inherited from `Entry`: `Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.selection()`, `Entry.xview()`, it has some other methods, described at `ttk.Combobox`.

Options

This widget accepts the following specific options:

Option	Description
exportselection	Boolean value. If set, the widget selection is linked to the Window Manager selection (which can be returned by invoking Misc.selection_get, for example).
justify	Specifies how the text is aligned within the widget. One of "left", "center", or "right".
height	Specifies the height of the pop-down listbox, in rows.
postcommand	A script (possibly registered with Misc.register) that is called immediately before displaying the values. It may specify which values to display.
state	One of "normal", "readonly", or "disabled". In the "readonly" state, the value may not be edited directly, and the user can only selection of the values from the dropdown list. In the "normal" state, the text field is directly editable. In the "disabled" state, no interaction is possible.
textvariable	Specifies a name whose value is linked to the widget value. Whenever the value associated with that name changes, the widget value is updated, and vice versa. See `tkinter.StringVar`.
values	Specifies the list of values to display in the drop-down listbox.
width	Specifies an integer value indicating the desired width of the entry window, in average-size characters of the widget's font.

Virtual events

The combobox widgets generates a **<<ComboboxSelected>>** virtual event when the user selects an element from the list of values.

ttk.Combobox

`class tkinter.ttk.Combobox`

`current(`*newindex=None*`)`
: If *newindex* is specified, sets the combobox value to the element position *newindex*. Otherwise, returns the index of the current value or -1 if the current value is not in the values list.

`get()`
: Returns the current value of the combobox.

`set(`*value*`)`
: Sets the value of the combobox to *value*.

26.2.5 Spinbox

The `ttk.Spinbox` widget is a `ttk.Entry` enhanced with increment and decrement arrows. It can be used for numbers or lists of string values. This widget is a subclass of `Entry`.

Besides the methods inherited from *`Widget`*: `Widget.cget()`, `Widget.configure()`, *`Widget.identify()`*, *`Widget.instate()`* and *`Widget.state()`*, and the following inherited from `Entry`: `Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.xview()`, it has some other methods, described at `ttk.Spinbox`.

Options

This widget accepts the following specific options:

Option	Description
from	Float value. If set, this is the minimum value to which the decrement button will decrement. Must be spelled as `from_` when used as an argument, since `from` is a Python keyword.
to	Float value. If set, this is the maximum value to which the increment button will increment.
increment	Float value. Specifies the amount which the increment/decrement buttons change the value. Defaults to 1.0.
values	Sequence of string or float values. If specified, the increment/decrement buttons will cycle through the items in this sequence rather than incrementing or decrementing numbers.
wrap	Boolean value. If `True`, increment and decrement buttons will cycle from the `to` value to the `from` value or the `from` value to the `to` value, respectively.
format	String value. This specifies the format of numbers set by the increment/decrement buttons. It must be in the form "%W.Pf", where W is the padded width of the value, P is the precision, and '%' and 'f' are literal.
command	Python callable. Will be called with no arguments whenever either of the increment or decrement buttons are pressed.

Virtual events

The spinbox widget generates an **<<Increment>>** virtual event when the user presses <Up>, and a **<<Decrement>>** virtual event when the user presses <Down>.

ttk.Spinbox

`class tkinter.ttk.Spinbox`

`get()`
Returns the current value of the spinbox.

`set(`*value*`)`
Sets the value of the spinbox to *value*.

26.2.6 Notebook

Ttk Notebook widget manages a collection of windows and displays a single one at a time. Each child window is associated with a tab, which the user may select to change the currently-displayed window.

Options

This widget accepts the following specific options:

Option	Description
height	If present and greater than zero, specifies the desired height of the pane area (not including internal padding or tabs). Otherwise, the maximum height of all panes is used.
padding	Specifies the amount of extra space to add around the outside of the notebook. The padding is a list up to four length specifications left top right bottom. If fewer than four elements are specified, bottom defaults to top, right defaults to left, and top defaults to left.
width	If present and greater than zero, specified the desired width of the pane area (not including internal padding). Otherwise, the maximum width of all panes is used.

Tab Options

There are also specific options for tabs:

Option	Description
state	Either "normal", "disabled" or "hidden". If "disabled", then the tab is not selectable. If "hidden", then the tab is not shown.
sticky	Specifies how the child window is positioned within the pane area. Value is a string containing zero or more of the characters "n", "s", "e" or "w". Each letter refers to a side (north, south, east or west) that the child window will stick to, as per the `grid()` geometry manager.
padding	Specifies the amount of extra space to add between the notebook and this pane. Syntax is the same as for the option padding used by this widget.
text	Specifies a text to be displayed in the tab.
image	Specifies an image to display in the tab. See the option image described in *Widget*.
compound	Specifies how to display the image relative to the text, in the case both options text and image are present. See *Label Options* for legal values.
underline	Specifies the index (0-based) of a character to underline in the text string. The underlined character is used for mnemonic activation if *Notebook.enable_traversal()* is called.

Tab Identifiers

The tab_id present in several methods of `ttk.Notebook` may take any of the following forms:

- An integer between zero and the number of tabs
- The name of a child window
- A positional specification of the form "@x,y", which identifies the tab
- The literal string "current", which identifies the currently-selected tab
- The literal string "end", which returns the number of tabs (only valid for *Notebook.index()*)

Virtual Events

This widget generates a **<<NotebookTabChanged>>** virtual event after a new tab is selected.

ttk.Notebook

class tkinter.ttk.Notebook

add(*child*, ***kw*)
Adds a new tab to the notebook.

If window is currently managed by the notebook but hidden, it is restored to its previous position.

See *Tab Options* for the list of available options.

forget(*tab_id*)
Removes the tab specified by *tab_id*, unmaps and unmanages the associated window.

hide(*tab_id*)
Hides the tab specified by *tab_id*.

The tab will not be displayed, but the associated window remains managed by the notebook and its configuration remembered. Hidden tabs may be restored with the *add()* command.

identify(*x*, *y*)
Returns the name of the tab element at position *x*, *y*, or the empty string if none.

index(*tab_id*)
Returns the numeric index of the tab specified by *tab_id*, or the total number of tabs if *tab_id* is the string "end".

insert(*pos*, *child*, ***kw*)
Inserts a pane at the specified position.

pos is either the string "end", an integer index, or the name of a managed child. If *child* is already managed by the notebook, moves it to the specified position.

See *Tab Options* for the list of available options.

select(*tab_id=None*)
Selects the specified *tab_id*.

The associated child window will be displayed, and the previously-selected window (if different) is unmapped. If *tab_id* is omitted, returns the widget name of the currently selected pane.

tab(*tab_id*, *option=None*, ***kw*)
Query or modify the options of the specific *tab_id*.

If *kw* is not given, returns a dictionary of the tab option values. If *option* is specified, returns the value of that *option*. Otherwise, sets the options to the corresponding values.

tabs()
Returns a list of windows managed by the notebook.

enable_traversal()
Enable keyboard traversal for a toplevel window containing this notebook.

This will extend the bindings for the toplevel window containing the notebook as follows:

- `Control-Tab`: selects the tab following the currently selected one.
- `Shift-Control-Tab`: selects the tab preceding the currently selected one.
- `Alt-K`: where *K* is the mnemonic (underlined) character of any tab, will select that tab.

Multiple notebooks in a single toplevel may be enabled for traversal, including nested notebooks. However, notebook traversal only works properly if all panes have the notebook they are in as master.

26.2.7 Progressbar

The `ttk.Progressbar` widget shows the status of a long-running operation. It can operate in two modes: 1) the determinate mode which shows the amount completed relative to the total amount of work to be done and 2) the indeterminate mode which provides an animated display to let the user know that work is progressing.

Options

This widget accepts the following specific options:

Option	Description
orient	One of "horizontal" or "vertical". Specifies the orientation of the progress bar.
length	Specifies the length of the long axis of the progress bar (width if horizontal, height if vertical).
mode	One of "determinate" or "indeterminate".
maximum	A number specifying the maximum value. Defaults to 100.
value	The current value of the progress bar. In "determinate" mode, this represents the amount of work completed. In "indeterminate" mode, it is interpreted as modulo *maximum*; that is, the progress bar completes one "cycle" when its value increases by *maximum*.
variable	A name which is linked to the option value. If specified, the value of the progress bar is automatically set to the value of this name whenever the latter is modified.
phase	Read-only option. The widget periodically increments the value of this option whenever its value is greater than 0 and, in determinate mode, less than maximum. This option may be used by the current theme to provide additional animation effects.

ttk.Progressbar

class tkinter.ttk.**Progressbar**

start(*interval=None*)
Begin autoincrement mode: schedules a recurring timer event that calls *Progressbar.step()* every *interval* milliseconds. If omitted, *interval* defaults to 50 milliseconds.

step(*amount=None*)
Increments the progress bar's value by *amount*.

amount defaults to 1.0 if omitted.

stop()
Stop autoincrement mode: cancels any recurring timer event initiated by *Progressbar.start()* for this progress bar.

26.2.8 Separator

The `ttk.Separator` widget displays a horizontal or vertical separator bar.

It has no other methods besides the ones inherited from `ttk.Widget`.

Options

This widget accepts the following specific option:

Option	Description
orient	One of “horizontal” or “vertical”. Specifies the orientation of the separator.

26.2.9 Sizegrip

The `ttk.Sizegrip` widget (also known as a grow box) allows the user to resize the containing toplevel window by pressing and dragging the grip.

This widget has neither specific options nor specific methods, besides the ones inherited from `ttk.Widget`.

Platform-specific notes

- On MacOS X, toplevel windows automatically include a built-in size grip by default. Adding a `Sizegrip` is harmless, since the built-in grip will just mask the widget.

Bugs

- If the containing toplevel's position was specified relative to the right or bottom of the screen (e.g. ….), the `Sizegrip` widget will not resize the window.
- This widget supports only “southeast” resizing.

26.2.10 Treeview

The `ttk.Treeview` widget displays a hierarchical collection of items. Each item has a textual label, an optional image, and an optional list of data values. The data values are displayed in successive columns after the tree label.

The order in which data values are displayed may be controlled by setting the widget option `displaycolumns`. The tree widget can also display column headings. Columns may be accessed by number or symbolic names listed in the widget option columns. See *Column Identifiers*.

Each item is identified by a unique name. The widget will generate item IDs if they are not supplied by the caller. There is a distinguished root item, named `{}`. The root item itself is not displayed; its children appear at the top level of the hierarchy.

Each item also has a list of tags, which can be used to associate event bindings with individual items and control the appearance of the item.

The Treeview widget supports horizontal and vertical scrolling, according to the options described in *Scrollable Widget Options* and the methods *`Treeview.xview()`* and *`Treeview.yview()`*.

Options

This widget accepts the following specific options:

Option	Description
columns	A list of column identifiers, specifying the number of columns and their names.
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".
height	Specifies the number of rows which should be visible. Note: the requested width is determined from the sum of the column widths.
padding	Specifies the internal padding for the widget. The padding is a list of up to four length specifications.
selectmode	Controls how the built-in class bindings manage the selection. One of "extended", "browse" or "none". If set to "extended" (the default), multiple items may be selected. If "browse", only a single item will be selected at a time. If "none", the selection will not be changed. Note that the application code and tag bindings can set the selection however they wish, regardless of the value of this option.
show	A list containing zero or more of the following values, specifying which elements of the tree to display. • tree: display tree labels in column #0. • headings: display the heading row. The default is "tree headings", i.e., show all elements. **Note**: Column #0 always refers to the tree column, even if show="tree" is not specified.

Item Options

The following item options may be specified for items in the insert and item widget commands.

Option	Description
text	The textual label to display for the item.
image	A Tk Image, displayed to the left of the label.
values	The list of values associated with the item. Each item should have the same number of values as the widget option columns. If there are fewer values than columns, the remaining values are assumed empty. If there are more values than columns, the extra values are ignored.
open	True/False value indicating whether the item's children should be displayed or hidden.
tags	A list of tags associated with this item.

Tag Options

The following options may be specified on tags:

Option	Description
foreground	Specifies the text foreground color.
background	Specifies the cell or item background color.
font	Specifies the font to use when drawing text.
image	Specifies the item image, in case the item's image option is empty.

Column Identifiers

Column identifiers take any of the following forms:

- A symbolic name from the list of columns option.
- An integer n, specifying the nth data column.
- A string of the form #n, where n is an integer, specifying the nth display column.

Notes:

- Item's option values may be displayed in a different order than the order in which they are stored.
- Column #0 always refers to the tree column, even if show="tree" is not specified.

A data column number is an index into an item's option values list; a display column number is the column number in the tree where the values are displayed. Tree labels are displayed in column #0. If option displaycolumns is not set, then data column n is displayed in column #n+1. Again, **column #0 always refers to the tree column**.

Virtual Events

The Treeview widget generates the following virtual events.

Event	Description
<<TreeviewSelect>>	Generated whenever the selection changes.
<<TreeviewOpen>>	Generated just before settings the focus item to open=True.
<<TreeviewClose>>	Generated just after setting the focus item to open=False.

The *Treeview.focus()* and *Treeview.selection()* methods can be used to determine the affected item or items.

ttk.Treeview

class tkinter.ttk.Treeview

bbox(*item*, *column=None*)
Returns the bounding box (relative to the treeview widget's window) of the specified *item* in the form (x, y, width, height).

If *column* is specified, returns the bounding box of that cell. If the *item* is not visible (i.e., if it is a descendant of a closed item or is scrolled offscreen), returns an empty string.

get_children(*item=None*)
Returns the list of children belonging to *item*.

If *item* is not specified, returns root children.

set_children(*item*, **newchildren*)
Replaces *item*'s child with *newchildren*.

Children present in *item* that are not present in *newchildren* are detached from the tree. No items in *newchildren* may be an ancestor of *item*. Note that not specifying *newchildren* results in detaching *item*'s children.

column(*column*, *option=None*, ***kw*)
Query or modify the options for the specified *column*.

If *kw* is not given, returns a dict of the column option values. If *option* is specified then the value for that *option* is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are:

- **id** Returns the column name. This is a read-only option.
- **anchor: One of the standard Tk anchor values.** Specifies how the text in this column should be aligned with respect to the cell.
- **minwidth: width** The minimum width of the column in pixels. The treeview widget will not make the column any smaller than specified by this option when the widget is resized or the user drags a column.
- **stretch: True/False** Specifies whether the column's width should be adjusted when the widget is resized.
- **width: width** The width of the column in pixels.

To configure the tree column, call this with column = "#0"

`delete`(**items*)
Delete all specified *items* and all their descendants.

The root item may not be deleted.

`detach`(**items*)
Unlinks all of the specified *items* from the tree.

The items and all of their descendants are still present, and may be reinserted at another point in the tree, but will not be displayed.

The root item may not be detached.

`exists`(*item*)
Returns `True` if the specified *item* is present in the tree.

`focus`(*item=None*)
If *item* is specified, sets the focus item to *item*. Otherwise, returns the current focus item, or '' if there is none.

`heading`(*column*, *option=None*, ***kw*)
Query or modify the heading options for the specified *column*.

If *kw* is not given, returns a dict of the heading option values. If *option* is specified then the value for that *option* is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are:

- **text: text** The text to display in the column heading.
- **image: imageName** Specifies an image to display to the right of the column heading.
- **anchor: anchor** Specifies how the heading text should be aligned. One of the standard Tk anchor values.
- **command: callback** A callback to be invoked when the heading label is pressed.

To configure the tree column heading, call this with column = "#0".

`identify`(*component*, *x*, *y*)
Returns a description of the specified *component* under the point given by *x* and *y*, or the empty string if no such *component* is present at that position.

`identify_row`(*y*)
Returns the item ID of the item at position *y*.

identify_column(*x*)
Returns the data column identifier of the cell at position *x*.

The tree column has ID #0.

identify_region(*x*, *y*)
Returns one of:

region	meaning
heading	Tree heading area.
separator	Space between two columns headings.
tree	The tree area.
cell	A data cell.

Availability: Tk 8.6.

identify_element(*x*, *y*)
Returns the element at position *x*, *y*.

Availability: Tk 8.6.

index(*item*)
Returns the integer index of *item* within its parent's list of children.

insert(*parent*, *index*, *iid=None*, ***kw*)
Creates a new item and returns the item identifier of the newly created item.

parent is the item ID of the parent item, or the empty string to create a new top-level item. *index* is an integer, or the value "end", specifying where in the list of parent's children to insert the new item. If *index* is less than or equal to zero, the new node is inserted at the beginning; if *index* is greater than or equal to the current number of children, it is inserted at the end. If *iid* is specified, it is used as the item identifier; *iid* must not already exist in the tree. Otherwise, a new unique identifier is generated.

See *Item Options* for the list of available points.

item(*item*, *option=None*, ***kw*)
Query or modify the options for the specified *item*.

If no options are given, a dict with options/values for the item is returned. If *option* is specified then the value for that option is returned. Otherwise, sets the options to the corresponding values as given by *kw*.

move(*item*, *parent*, *index*)
Moves *item* to position *index* in *parent*'s list of children.

It is illegal to move an item under one of its descendants. If *index* is less than or equal to zero, *item* is moved to the beginning; if greater than or equal to the number of children, it is moved to the end. If *item* was detached it is reattached.

next(*item*)
Returns the identifier of *item*'s next sibling, or '' if *item* is the last child of its parent.

parent(*item*)
Returns the ID of the parent of *item*, or '' if *item* is at the top level of the hierarchy.

prev(*item*)
Returns the identifier of *item*'s previous sibling, or '' if *item* is the first child of its parent.

reattach(*item*, *parent*, *index*)
An alias for *Treeview.move()*.

see(*item*)
Ensure that *item* is visible.

Sets all of *item*'s ancestors open option to `True`, and scrolls the widget if necessary so that *item* is within the visible portion of the tree.

selection(*selop=None*, *items=None*)
If *selop* is not specified, returns selected items. Otherwise, it will act according to the following selection methods.

Deprecated since version 3.6, will be removed in version 3.8: Using `selection()` for changing the selection state is deprecated. Use the following selection methods instead.

selection_set(**items*)
items becomes the new selection.

Changed in version 3.6: *items* can be passed as separate arguments, not just as a single tuple.

selection_add(**items*)
Add *items* to the selection.

Changed in version 3.6: *items* can be passed as separate arguments, not just as a single tuple.

selection_remove(**items*)
Remove *items* from the selection.

Changed in version 3.6: *items* can be passed as separate arguments, not just as a single tuple.

selection_toggle(**items*)
Toggle the selection state of each item in *items*.

Changed in version 3.6: *items* can be passed as separate arguments, not just as a single tuple.

set(*item*, *column=None*, *value=None*)
With one argument, returns a dictionary of column/value pairs for the specified *item*. With two arguments, returns the current value of the specified *column*. With three arguments, sets the value of given *column* in given *item* to the specified *value*.

tag_bind(*tagname*, *sequence=None*, *callback=None*)
Bind a callback for the given event *sequence* to the tag *tagname*. When an event is delivered to an item, the callbacks for each of the item's tags option are called.

tag_configure(*tagname*, *option=None*, ***kw*)
Query or modify the options for the specified *tagname*.

If *kw* is not given, returns a dict of the option settings for *tagname*. If *option* is specified, returns the value for that *option* for the specified *tagname*. Otherwise, sets the options to the corresponding values for the given *tagname*.

tag_has(*tagname*, *item=None*)
If *item* is specified, returns 1 or 0 depending on whether the specified *item* has the given *tagname*. Otherwise, returns a list of all items that have the specified tag.

Availability: Tk 8.6

xview(**args*)
Query or modify horizontal position of the treeview.

yview(**args*)
Query or modify vertical position of the treeview.

26.2.11 Ttk Styling

Each widget in `ttk` is assigned a style, which specifies the set of elements making up the widget and how they are arranged, along with dynamic and default settings for element options. By default the style name is the same as the widget's class name, but it may be overridden by the widget's style option. If you don't know the class name of a widget, use the method `Misc.winfo_class()` (somewidget.winfo_class()).

See also:

Tcl'2004 conference presentation This document explains how the theme engine works

`class tkinter.ttk.Style`

This class is used to manipulate the style database.

`configure`(*style*, *query_opt=None*, ***kw*)

Query or set the default value of the specified option(s) in *style*.

Each key in *kw* is an option and each value is a string identifying the value for that option.

For example, to change every default button to be a flat button with some padding and a different background color:

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

ttk.Style().configure("TButton", padding=6, relief="flat",
   background="#ccc")

btn = ttk.Button(text="Sample")
btn.pack()

root.mainloop()
```

`map`(*style*, *query_opt=None*, ***kw*)

Query or sets dynamic values of the specified option(s) in *style*.

Each key in *kw* is an option and each value should be a list or a tuple (usually) containing statespecs grouped in tuples, lists, or some other preference. A statespec is a compound of one or more states and then a value.

An example may make it more understandable:

```
import tkinter
from tkinter import ttk

root = tkinter.Tk()

style = ttk.Style()
style.map("C.TButton",
    foreground=[('pressed', 'red'), ('active', 'blue')],
    background=[('pressed', '!disabled', 'black'), ('active', 'white')]
    )

colored_btn = ttk.Button(text="Test", style="C.TButton").pack()

root.mainloop()
```

Note that the order of the (states, value) sequences for an option does matter, if the order is changed to `[('active', 'blue'), ('pressed', 'red')]` in the foreground option, for example, the result would be a blue foreground when the widget were in active or pressed states.

lookup(*style*, *option*, *state=None*, *default=None*)
Returns the value specified for *option* in *style*.

If *state* is specified, it is expected to be a sequence of one or more states. If the *default* argument is set, it is used as a fallback value in case no specification for option is found.

To check what font a Button uses by default:

```
from tkinter import ttk

print(ttk.Style().lookup("TButton", "font"))
```

layout(*style*, *layoutspec=None*)
Define the widget layout for given *style*. If *layoutspec* is omitted, return the layout specification for given style.

layoutspec, if specified, is expected to be a list or some other sequence type (excluding strings), where each item should be a tuple and the first item is the layout name and the second item should have the format described in *Layouts*.

To understand the format, see the following example (it is not intended to do anything useful):

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.layout("TMenubutton", [
   ("Menubutton.background", None),
   ("Menubutton.button", {"children":
       [("Menubutton.focus", {"children":
           [("Menubutton.padding", {"children":
               [("Menubutton.label", {"side": "left", "expand": 1})]
           })]
       })]
   }),
])

mbtn = ttk.Menubutton(text='Text')
mbtn.pack()
root.mainloop()
```

element_create(*elementname*, *etype*, **args*, ***kw*)
Create a new element in the current theme, of the given *etype* which is expected to be either "image", "from" or "vsapi". The latter is only available in Tk 8.6a for Windows XP and Vista and is not described here.

If "image" is used, *args* should contain the default image name followed by statespec/value pairs (this is the imagespec), and *kw* may have the following options:

- **border=padding** padding is a list of up to four integers, specifying the left, top, right, and bottom borders, respectively.

- **height=height** Specifies a minimum height for the element. If less than zero, the base image's height is used as a default.
- **padding=padding** Specifies the element's interior padding. Defaults to border's value if not specified.
- **sticky=spec** Specifies how the image is placed within the final parcel. spec contains zero or more characters "n", "s", "w", or "e".
- **width=width** Specifies a minimum width for the element. If less than zero, the base image's width is used as a default.

If "from" is used as the value of *etype*, *element_create()* will clone an existing element. *args* is expected to contain a themename, from which the element will be cloned, and optionally an element to clone from. If this element to clone from is not specified, an empty element will be used. *kw* is discarded.

element_names()
Returns the list of elements defined in the current theme.

element_options(*elementname*)
Returns the list of *elementname*'s options.

theme_create(*themename*, *parent=None*, *settings=None*)
Create a new theme.

It is an error if *themename* already exists. If *parent* is specified, the new theme will inherit styles, elements and layouts from the parent theme. If *settings* are present they are expected to have the same syntax used for *theme_settings()*.

theme_settings(*themename*, *settings*)
Temporarily sets the current theme to *themename*, apply specified *settings* and then restore the previous theme.

Each key in *settings* is a style and each value may contain the keys 'configure', 'map', 'layout' and 'element create' and they are expected to have the same format as specified by the methods *Style.configure()*, *Style.map()*, *Style.layout()* and *Style.element_create()* respectively.

As an example, let's change the Combobox for the default theme a bit:

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.theme_settings("default", {
   "TCombobox": {
       "configure": {"padding": 5},
       "map": {
           "background": [("active", "green2"),
                          ("!disabled", "green4")],
           "fieldbackground": [("!disabled", "green3")],
           "foreground": [("focus", "OliveDrab1"),
                          ("!disabled", "OliveDrab2")]
       }
   }
})

combo = ttk.Combobox().pack()
```

(continues on next page)

(continued from previous page)

```
root.mainloop()
```

theme_names()
Returns a list of all known themes.

theme_use(*themename=None*)
If *themename* is not given, returns the theme in use. Otherwise, sets the current theme to *themename*, refreshes all widgets and emits a <<ThemeChanged>> event.

Layouts

A layout can be just `None`, if it takes no options, or a dict of options specifying how to arrange the element. The layout mechanism uses a simplified version of the pack geometry manager: given an initial cavity, each element is allocated a parcel. Valid options/values are:

- **side: whichside** Specifies which side of the cavity to place the element; one of top, right, bottom or left. If omitted, the element occupies the entire cavity.
- **sticky: nswe** Specifies where the element is placed inside its allocated parcel.
- **unit: 0 or 1** If set to 1, causes the element and all of its descendants to be treated as a single element for the purposes of *Widget.identify()* et al. It's used for things like scrollbar thumbs with grips.
- **children: [sublayout...]** Specifies a list of elements to place inside the element. Each element is a tuple (or other sequence type) where the first item is the layout name, and the other is a *Layout*.

26.3 tkinter.tix — Extension widgets for Tk

Source code: Lib/tkinter/tix.py

Deprecated since version 3.6: This Tk extension is unmaintained and should not be used in new code. Use *tkinter.ttk* instead.

The *tkinter.tix* (Tk Interface Extension) module provides an additional rich set of widgets. Although the standard Tk library has many useful widgets, they are far from complete. The *tkinter.tix* library provides most of the commonly needed widgets that are missing from standard Tk: *HList*, *ComboBox*, *Control* (a.k.a. SpinBox) and an assortment of scrollable widgets. *tkinter.tix* also includes many more widgets that are generally useful in a wide range of applications: *NoteBook*, *FileEntry*, *PanedWindow*, etc; there are more than 40 of them.

With all these new widgets, you can introduce new interaction techniques into applications, creating more useful and more intuitive user interfaces. You can design your application by choosing the most appropriate widgets to match the special needs of your application and users.

See also:

Tix Homepage The home page for `Tix`. This includes links to additional documentation and downloads.

Tix Man Pages On-line version of the man pages and reference material.

Tix Programming Guide On-line version of the programmer's reference material.

Tix Development Applications Tix applications for development of Tix and Tkinter programs. Tide applications work under Tk or Tkinter, and include **TixInspect**, an inspector to remotely modify and debug Tix/Tk/Tkinter applications.

26.3.1 Using Tix

class tkinter.tix.Tk(*screenName=None*, *baseName=None*, *className='Tix'*)
Toplevel widget of Tix which represents mostly the main window of an application. It has an associated Tcl interpreter.

Classes in the *tkinter.tix* module subclasses the classes in the *tkinter*. The former imports the latter, so to use *tkinter.tix* with Tkinter, all you need to do is to import one module. In general, you can just import *tkinter.tix*, and replace the toplevel call to *tkinter.Tk* with `tix.Tk`:

```
from tkinter import tix
from tkinter.constants import *
root = tix.Tk()
```

To use *tkinter.tix*, you must have the Tix widgets installed, usually alongside your installation of the Tk widgets. To test your installation, try the following:

```
from tkinter import tix
root = tix.Tk()
root.tk.eval('package require Tix')
```

If this fails, you have a Tk installation problem which must be resolved before proceeding. Use the environment variable `TIX_LIBRARY` to point to the installed Tix library directory, and make sure you have the dynamic object library (`tix8183.dll` or `libtix8183.so`) in the same directory that contains your Tk dynamic object library (`tk8183.dll` or `libtk8183.so`). The directory with the dynamic object library should also have a file called `pkgIndex.tcl` (case sensitive), which contains the line:

```
package ifneeded Tix 8.1 [list load "[file join $dir tix8183.dll]" Tix]
```

26.3.2 Tix Widgets

Tix introduces over 40 widget classes to the *tkinter* repertoire.

Basic Widgets

class tkinter.tix.Balloon
A Balloon that pops up over a widget to provide help. When the user moves the cursor inside a widget to which a Balloon widget has been bound, a small pop-up window with a descriptive message will be shown on the screen.

class tkinter.tix.ButtonBox
The ButtonBox widget creates a box of buttons, such as is commonly used for `Ok Cancel`.

class tkinter.tix.ComboBox
The ComboBox widget is similar to the combo box control in MS Windows. The user can select a choice by either typing in the entry subwidget or selecting from the listbox subwidget.

class tkinter.tix.Control
The Control widget is also known as the `SpinBox` widget. The user can adjust the value by pressing the two arrow buttons or by entering the value directly into the entry. The new value will be checked against the user-defined upper and lower limits.

class tkinter.tix.LabelEntry
The LabelEntry widget packages an entry widget and a label into one mega widget. It can be used to simplify the creation of "entry-form" type of interface.

class tkinter.tix.LabelFrame
The LabelFrame widget packages a frame widget and a label into one mega widget. To create widgets inside a LabelFrame widget, one creates the new widgets relative to the `frame` subwidget and manage them inside the `frame` subwidget.

class tkinter.tix.Meter
The Meter widget can be used to show the progress of a background job which may take a long time to execute.

class tkinter.tix.OptionMenu
The OptionMenu creates a menu button of options.

class tkinter.tix.PopupMenu
The PopupMenu widget can be used as a replacement of the `tk_popup` command. The advantage of the `Tix` *PopupMenu* widget is it requires less application code to manipulate.

class tkinter.tix.Select
The Select widget is a container of button subwidgets. It can be used to provide radio-box or check-box style of selection options for the user.

class tkinter.tix.StdButtonBox
The StdButtonBox widget is a group of standard buttons for Motif-like dialog boxes.

File Selectors

class tkinter.tix.DirList
The DirList widget displays a list view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class tkinter.tix.DirTree
The DirTree widget displays a tree view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class tkinter.tix.DirSelectDialog
The DirSelectDialog widget presents the directories in the file system in a dialog window. The user can use this dialog window to navigate through the file system to select the desired directory.

class tkinter.tix.DirSelectBox
The *DirSelectBox* is similar to the standard Motif(TM) directory-selection box. It is generally used for the user to choose a directory. DirSelectBox stores the directories mostly recently selected into a ComboBox widget so that they can be quickly selected again.

class tkinter.tix.ExFileSelectBox
The ExFileSelectBox widget is usually embedded in a tixExFileSelectDialog widget. It provides a convenient method for the user to select files. The style of the *ExFileSelectBox* widget is very similar to the standard file dialog on MS Windows 3.1.

class tkinter.tix.FileSelectBox
The FileSelectBox is similar to the standard Motif(TM) file-selection box. It is generally used for the user to choose a file. FileSelectBox stores the files mostly recently selected into a *ComboBox* widget so that they can be quickly selected again.

class tkinter.tix.FileEntry
The FileEntry widget can be used to input a filename. The user can type in the filename manually. Alternatively, the user can press the button widget that sits next to the entry, which will bring up a file selection dialog.

Hierarchical ListBox

class tkinter.tix.HList
The HList widget can be used to display any data that have a hierarchical structure, for example, file system directory trees. The list entries are indented and connected by branch lines according to their places in the hierarchy.

class tkinter.tix.CheckList
The CheckList widget displays a list of items to be selected by the user. CheckList acts similarly to the Tk checkbutton or radiobutton widgets, except it is capable of handling many more items than checkbuttons or radiobuttons.

class tkinter.tix.Tree
The Tree widget can be used to display hierarchical data in a tree form. The user can adjust the view of the tree by opening or closing parts of the tree.

Tabular ListBox

class tkinter.tix.TList
The TList widget can be used to display data in a tabular format. The list entries of a *TList* widget are similar to the entries in the Tk listbox widget. The main differences are (1) the *TList* widget can display the list entries in a two dimensional format and (2) you can use graphical images as well as multiple colors and fonts for the list entries.

Manager Widgets

class tkinter.tix.PanedWindow
The PanedWindow widget allows the user to interactively manipulate the sizes of several panes. The panes can be arranged either vertically or horizontally. The user changes the sizes of the panes by dragging the resize handle between two panes.

class tkinter.tix.ListNoteBook
The ListNoteBook widget is very similar to the `TixNoteBook` widget: it can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages (windows). At one time only one of these pages can be shown. The user can navigate through these pages by choosing the name of the desired page in the `hlist` subwidget.

class tkinter.tix.NoteBook
The NoteBook widget can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages. At one time only one of these pages can be shown. The user can navigate through these pages by choosing the visual "tabs" at the top of the NoteBook widget.

Image Types

The *tkinter.tix* module adds:

- pixmap capabilities to all *tkinter.tix* and *tkinter* widgets to create color images from XPM files.
- Compound image types can be used to create images that consists of multiple horizontal lines; each line is composed of a series of items (texts, bitmaps, images or spaces) arranged from left to right. For example, a compound image can be used to display a bitmap and a text string simultaneously in a Tk `Button` widget.

Miscellaneous Widgets

class tkinter.tix.InputOnly
The InputOnly widgets are to accept inputs from the user, which can be done with the `bind` command (Unix only).

Form Geometry Manager

In addition, *tkinter.tix* augments *tkinter* by providing:

class tkinter.tix.Form
The Form geometry manager based on attachment rules for all Tk widgets.

26.3.3 Tix Commands

class tkinter.tix.tixCommand
The tix commands provide access to miscellaneous elements of `Tix`'s internal state and the `Tix` application context. Most of the information manipulated by these methods pertains to the application as a whole, or to a screen or display, rather than to a particular window.

To view the current settings, the common usage is:

```
from tkinter import tix
root = tix.Tk()
print(root.tix_configure())
```

tixCommand.tix_configure(*cnf=None*, ***kw*)
Query or modify the configuration options of the Tix application context. If no option is specified, returns a dictionary all of the available options. If option is specified with no value, then the method returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no option is specified). If one or more option-value pairs are specified, then the method modifies the given option(s) to have the given value(s); in this case the method returns an empty string. Option may be any of the configuration options.

tixCommand.tix_cget(*option*)
Returns the current value of the configuration option given by *option*. Option may be any of the configuration options.

tixCommand.tix_getbitmap(*name*)
Locates a bitmap file of the name `name.xpm` or `name` in one of the bitmap directories (see the *tix_addbitmapdir()* method). By using *tix_getbitmap()*, you can avoid hard coding the pathnames of the bitmap files in your application. When successful, it returns the complete pathname of the bitmap file, prefixed with the character `@`. The returned value can be used to configure the `bitmap` option of the Tk and Tix widgets.

tixCommand.tix_addbitmapdir(*directory*)
Tix maintains a list of directories under which the *tix_getimage()* and *tix_getbitmap()* methods will search for image files. The standard bitmap directory is `$TIX_LIBRARY/bitmaps`. The *tix_addbitmapdir()* method adds *directory* into this list. By using this method, the image files of an applications can also be located using the *tix_getimage()* or *tix_getbitmap()* method.

tixCommand.tix_filedialog([*dlgclass*])
Returns the file selection dialog that may be shared among different calls from this application. This method will create a file selection dialog widget when it is called the first time. This dialog will be returned by all subsequent calls to *tix_filedialog()*. An optional dlgclass parameter can be passed

as a string to specified what type of file selection dialog widget is desired. Possible options are `tix`, `FileSelectDialog` or `tixExFileSelectDialog`.

`tixCommand.tix_getimage`(*self*, *name*)
Locates an image file of the name `name.xpm`, `name.xbm` or `name.ppm` in one of the bitmap directories (see the *tix_addbitmapdir()* method above). If more than one file with the same name (but different extensions) exist, then the image type is chosen according to the depth of the X display: xbm images are chosen on monochrome displays and color images are chosen on color displays. By using *tix_getimage()*, you can avoid hard coding the pathnames of the image files in your application. When successful, this method returns the name of the newly created image, which can be used to configure the `image` option of the Tk and Tix widgets.

`tixCommand.tix_option_get`(*name*)
Gets the options maintained by the Tix scheme mechanism.

`tixCommand.tix_resetoptions`(*newScheme*, *newFontSet*[, *newScmPrio*])
Resets the scheme and fontset of the Tix application to *newScheme* and *newFontSet*, respectively. This affects only those widgets created after this call. Therefore, it is best to call the resetoptions method before the creation of any widgets in a Tix application.

The optional parameter *newScmPrio* can be given to reset the priority level of the Tk options set by the Tix schemes.

Because of the way Tk handles the X option database, after Tix has been has imported and inited, it is not possible to reset the color schemes and font sets using the `tix_config()` method. Instead, the *tix_resetoptions()* method must be used.

26.4 `tkinter.scrolledtext` — Scrolled Text Widget

Source code: Lib/tkinter/scrolledtext.py

The *tkinter.scrolledtext* module provides a class of the same name which implements a basic text widget which has a vertical scroll bar configured to do the "right thing." Using the `ScrolledText` class is a lot easier than setting up a text widget and scroll bar directly. The constructor is the same as that of the `tkinter.Text` class.

The text widget and scrollbar are packed together in a `Frame`, and the methods of the `Grid` and `Pack` geometry managers are acquired from the `Frame` object. This allows the `ScrolledText` widget to be used directly to achieve most normal geometry management behavior.

Should more specific control be necessary, the following attributes are available:

`ScrolledText.frame`
The frame which surrounds the text and scroll bar widgets.

`ScrolledText.vbar`
The scroll bar widget.

26.5 IDLE

Source code: Lib/idlelib/

IDLE is Python's Integrated Development and Learning Environment.

IDLE has the following features:

- coded in 100% pure Python, using the *tkinter* GUI toolkit
- cross-platform: works mostly the same on Windows, Unix, and macOS
- Python shell window (interactive interpreter) with colorizing of code input, output, and error messages
- multi-window text editor with multiple undo, Python colorizing, smart indent, call tips, auto completion, and other features
- search within any window, replace within editor windows, and search through multiple files (grep)
- debugger with persistent breakpoints, stepping, and viewing of global and local namespaces
- configuration, browsers, and other dialogs

26.5.1 Menus

IDLE has two main window types, the Shell window and the Editor window. It is possible to have multiple editor windows simultaneously. On Windows and Linux, each has its own top menu. Each menu documented below indicates which window type it is associated with.

Output windows, such as used for Edit => Find in Files, are a subtype of editor window. They currently have the same top menu but a different default title and context menu.

On macOS, there is one application menu. It dynamically changes according to the window currently selected. It has an IDLE menu, and some entries described below are moved around to conform to Apple guidlines.

File menu (Shell and Editor)

New File Create a new file editing window.

Open... Open an existing file with an Open dialog.

Recent Files Open a list of recent files. Click one to open it.

Open Module... Open an existing module (searches sys.path).

Class Browser Show functions, classes, and methods in the current Editor file in a tree structure. In the shell, open a module first.

Path Browser Show sys.path directories, modules, functions, classes and methods in a tree structure.

Save Save the current window to the associated file, if there is one. Windows that have been changed since being opened or last saved have a * before and after the window title. If there is no associated file, do Save As instead.

Save As... Save the current window with a Save As dialog. The file saved becomes the new associated file for the window.

Save Copy As... Save the current window to different file without changing the associated file.

Print Window Print the current window to the default printer.

Close Close the current window (ask to save if unsaved).

Exit Close all windows and quit IDLE (ask to save unsaved windows).

Edit menu (Shell and Editor)

Undo Undo the last change to the current window. A maximum of 1000 changes may be undone.

Redo Redo the last undone change to the current window.

Cut Copy selection into the system-wide clipboard; then delete the selection.

Copy Copy selection into the system-wide clipboard.

Paste Insert contents of the system-wide clipboard into the current window.

The clipboard functions are also available in context menus.

Select All Select the entire contents of the current window.

Find... Open a search dialog with many options

Find Again Repeat the last search, if there is one.

Find Selection Search for the currently selected string, if there is one.

Find in Files... Open a file search dialog. Put results in a new output window.

Replace... Open a search-and-replace dialog.

Go to Line Move cursor to the line number requested and make that line visible.

Show Completions Open a scrollable list allowing selection of keywords and attributes. See *Completions* in the Editing and navigation section below.

Expand Word Expand a prefix you have typed to match a full word in the same window; repeat to get a different expansion.

Show call tip After an unclosed parenthesis for a function, open a small window with function parameter hints. See *Calltips* in the Editing and navigation section below.

Show surrounding parens Highlight the surrounding parenthesis.

Format menu (Editor window only)

Indent Region Shift selected lines right by the indent width (default 4 spaces).

Dedent Region Shift selected lines left by the indent width (default 4 spaces).

Comment Out Region Insert ## in front of selected lines.

Uncomment Region Remove leading # or ## from selected lines.

Tabify Region Turn *leading* stretches of spaces into tabs. (Note: We recommend using 4 space blocks to indent Python code.)

Untabify Region Turn *all* tabs into the correct number of spaces.

Toggle Tabs Open a dialog to switch between indenting with spaces and tabs.

New Indent Width Open a dialog to change indent width. The accepted default by the Python community is 4 spaces.

Format Paragraph Reformat the current blank-line-delimited paragraph in comment block or multiline string or selected line in a string. All lines in the paragraph will be formatted to less than N columns, where N defaults to 72.

Strip trailing whitespace Remove trailing space and other whitespace characters after the last non-whitespace character of a line by applying str.rstrip to each line, including lines within multiline strings.

Run menu (Editor window only)

Python Shell Open or wake up the Python Shell window.

Check Module Check the syntax of the module currently open in the Editor window. If the module has not been saved IDLE will either prompt the user to save or autosave, as selected in the General tab of the Idle Settings dialog. If there is a syntax error, the approximate location is indicated in the Editor window.

Run Module Do Check Module (above). If no error, restart the shell to clean the environment, then execute the module. Output is displayed in the Shell window. Note that output requires use of `print` or `write`. When execution is complete, the Shell retains focus and displays a prompt. At this point, one may interactively explore the result of execution. This is similar to executing a file with `python -i file` at a command line.

Shell menu (Shell window only)

View Last Restart Scroll the shell window to the last Shell restart.

Restart Shell Restart the shell to clean the environment.

Previous History Cycle through earlier commands in history which match the current entry.

Next History Cycle through later commands in history which match the current entry.

Interrupt Execution Stop a running program.

Debug menu (Shell window only)

Go to File/Line Look on the current line. with the cursor, and the line above for a filename and line number. If found, open the file if not already open, and show the line. Use this to view source lines referenced in an exception traceback and lines found by Find in Files. Also available in the context menu of the Shell window and Output windows.

Debugger (toggle) When activated, code entered in the Shell or run from an Editor will run under the debugger. In the Editor, breakpoints can be set with the context menu. This feature is still incomplete and somewhat experimental.

Stack Viewer Show the stack traceback of the last exception in a tree widget, with access to locals and globals.

Auto-open Stack Viewer Toggle automatically opening the stack viewer on an unhandled exception.

Options menu (Shell and Editor)

Configure IDLE Open a configuration dialog and change preferences for the following: fonts, indentation, keybindings, text color themes, startup windows and size, additional help sources, and extensions. On macOS, open the configuration dialog by selecting Preferences in the application menu. For more, see *Setting preferences* under Help and preferences.

Zoom/Restore Height Toggles the window between normal size and maximum height. The initial size defaults to 40 lines by 80 chars unless changed on the General tab of the Configure IDLE dialog.

Show/Hide Code Context (Editor Window only) Open a pane at the top of the edit window which shows the block context of the code which has scrolled above the top of the window. See *Code Context* in the Editing and Navigation section below.

Window menu (Shell and Editor)

Lists the names of all open windows; select one to bring it to the foreground (deiconifying it if necessary).

Help menu (Shell and Editor)

About IDLE Display version, copyright, license, credits, and more.

IDLE Help Display this IDLE document, detailing the menu options, basic editing and navigation, and other tips.

Python Docs Access local Python documentation, if installed, or start a web browser and open docs.python.org showing the latest Python documentation.

Turtle Demo Run the turtledemo module with example Python code and turtle drawings.

Additional help sources may be added here with the Configure IDLE dialog under the General tab. See the *Help sources* subsection below for more on Help menu choices.

Context Menus

Open a context menu by right-clicking in a window (Control-click on macOS). Context menus have the standard clipboard functions also on the Edit menu.

Cut Copy selection into the system-wide clipboard; then delete the selection.

Copy Copy selection into the system-wide clipboard.

Paste Insert contents of the system-wide clipboard into the current window.

Editor windows also have breakpoint functions. Lines with a breakpoint set are specially marked. Breakpoints only have an effect when running under the debugger. Breakpoints for a file are saved in the user's .idlerc directory.

Set Breakpoint Set a breakpoint on the current line.

Clear Breakpoint Clear the breakpoint on that line.

Shell and Output windows also have the following.

Go to file/line Same as in Debug menu.

The Shell window also has an output squeezing facility explained in the the *Python Shell window* subsection below.

Squeeze If the cursor is over an output line, squeeze all the output between the code above and the prompt below down to a 'Squeezed text' label.

26.5.2 Editing and navigation

Editor windows

IDLE may open editor windows when it starts, depending on settings and how you start IDLE. Thereafter, use the File menu. There can be only one open editor window for a given file.

The title bar contains the name of the file, the full path, and the version of Python and IDLE running the window. The status bar contains the line number ('Ln') and column number ('Col'). Line numbers start with 1; column numbers with 0.

IDLE assumes that files with a known .py* extension contain Python code and that other files do not. Run Python code with the Run menu.

Key bindings

In this section, 'C' refers to the `Control` key on Windows and Unix and the `Command` key on macOS.

- `Backspace` deletes to the left; `Del` deletes to the right
- `C-Backspace` delete word left; `C-Del` delete word to the right
- Arrow keys and `Page Up`/`Page Down` to move around
- `C-LeftArrow` and `C-RightArrow` moves by words
- `Home`/`End` go to begin/end of line
- `C-Home`/`C-End` go to begin/end of file
- Some useful Emacs bindings are inherited from Tcl/Tk:
  - `C-a` beginning of line
  - `C-e` end of line
  - `C-k` kill line (but doesn't put it in clipboard)
  - `C-l` center window around the insertion point
  - `C-b` go backward one character without deleting (usually you can also use the cursor key for this)
  - `C-f` go forward one character without deleting (usually you can also use the cursor key for this)
  - `C-p` go up one line (usually you can also use the cursor key for this)
  - `C-d` delete next character

Standard keybindings (like `C-c` to copy and `C-v` to paste) may work. Keybindings are selected in the Configure IDLE dialog.

Automatic indentation

After a block-opening statement, the next line is indented by 4 spaces (in the Python Shell window by one tab). After certain keywords (break, return etc.) the next line is dedented. In leading indentation, `Backspace` deletes up to 4 spaces if they are there. `Tab` inserts spaces (in the Python Shell window one tab), number depends on Indent width. Currently, tabs are restricted to four spaces due to Tcl/Tk limitations.

See also the indent/dedent region commands on the *Format menu*.

Completions

Completions are supplied for functions, classes, and attributes of classes, both built-in and user-defined. Completions are also provided for filenames.

The AutoCompleteWindow (ACW) will open after a predefined delay (default is two seconds) after a '.' or (in a string) an os.sep is typed. If after one of those characters (plus zero or more other characters) a tab is typed the ACW will open immediately if a possible continuation is found.

If there is only one possible completion for the characters entered, a `Tab` will supply that completion without opening the ACW.

'Show Completions' will force open a completions window, by default the `C-space` will open a completions window. In an empty string, this will contain the files in the current directory. On a blank line, it will contain the built-in and user-defined functions and classes in the current namespaces, plus any modules imported. If some characters have been entered, the ACW will attempt to be more specific.

If a string of characters is typed, the ACW selection will jump to the entry most closely matching those characters. Entering a `tab` will cause the longest non-ambiguous match to be entered in the Editor window

or Shell. Two `tab` in a row will supply the current ACW selection, as will return or a double click. Cursor keys, Page Up/Down, mouse selection, and the scroll wheel all operate on the ACW.

"Hidden" attributes can be accessed by typing the beginning of hidden name after a '.', e.g. '_'. This allows access to modules with `__all__` set, or to class-private attributes.

Completions and the 'Expand Word' facility can save a lot of typing!

Completions are currently limited to those in the namespaces. Names in an Editor window which are not via `__main__` and *sys.modules* will not be found. Run the module once with your imports to correct this situation. Note that IDLE itself places quite a few modules in sys.modules, so much can be found by default, e.g. the re module.

If you don't like the ACW popping up unbidden, simply make the delay longer or disable the extension.

Calltips

A calltip is shown when one types `(` after the name of an *accessible* function. A name expression may include dots and subscripts. A calltip remains until it is clicked, the cursor is moved out of the argument area, or `)` is typed. When the cursor is in the argument part of a definition, the menu or shortcut display a calltip.

A calltip consists of the function signature and the first line of the docstring. For builtins without an accessible signature, the calltip consists of all lines up the fifth line or the first blank line. These details may change.

The set of *accessible* functions depends on what modules have been imported into the user process, including those imported by Idle itself, and what definitions have been run, all since the last restart.

For example, restart the Shell and enter `itertools.count(`. A calltip appears because Idle imports itertools into the user process for its own use. (This could change.) Enter `turtle.write(` and nothing appears. Idle does not import turtle. The menu or shortcut do nothing either. Enter `import turtle` and then `turtle.write(` will work.

In an editor, import statements have no effect until one runs the file. One might want to run a file after writing the import statements at the top, or immediately run an existing file before editing.

Code Context

Within an editor window containing Python code, code context can be toggled in order to show or hide a pane at the top of the window. When shown, this pane freezes the opening lines for block code, such as those beginning with `class`, `def`, or `if` keywords, that would have otherwise scrolled out of view. The size of the pane will be expanded and contracted as needed to show the all current levels of context, up to the maximum number of lines defined in the Configure IDLE dialog (which defaults to 15). If there are no current context lines and the feature is toggled on, a single blank line will display. Clicking on a line in the context pane will move that line to the top of the editor.

The text and background colors for the context pane can be configured under the Highlights tab in the Configure IDLE dialog.

Python Shell window

With IDLE's Shell, one enters, edits, and recalls complete statements. Most consoles and terminals only work with a single physical line at a time.

When one pastes code into Shell, it is not compiled and possibly executed until one hits `Return`. One may edit pasted code first. If one pastes more that one statement into Shell, the result will be a *SyntaxError* when multiple statements are compiled as if they were one.

The editing features described in previous subsections work when entering code interactively. IDLE's Shell window also responds to the following keys.

- `C-c` interrupts executing command
- `C-d` sends end-of-file; closes window if typed at a `>>>` prompt
- `Alt-/` (Expand word) is also useful to reduce typing

  Command history

  - `Alt-p` retrieves previous command matching what you have typed. On macOS use `C-p`.
  - `Alt-n` retrieves next. On macOS use `C-n`.
  - `Return` while on any previous command retrieves that command

Text colors

Idle defaults to black on white text, but colors text with special meanings. For the shell, these are shell output, shell error, user output, and user error. For Python code, at the shell prompt or in an editor, these are keywords, builtin class and function names, names following `class` and `def`, strings, and comments. For any text window, these are the cursor (when present), found text (when possible), and selected text.

Text coloring is done in the background, so uncolorized text is occasionally visible. To change the color scheme, use the Configure IDLE dialog Highlighting tab. The marking of debugger breakpoint lines in the editor and text in popups and dialogs is not user-configurable.

26.5.3 Startup and code execution

Upon startup with the `-s` option, IDLE will execute the file referenced by the environment variables `IDLESTARTUP` or `PYTHONSTARTUP`. IDLE first checks for `IDLESTARTUP`; if `IDLESTARTUP` is present the file referenced is run. If `IDLESTARTUP` is not present, IDLE checks for `PYTHONSTARTUP`. Files referenced by these environment variables are convenient places to store functions that are used frequently from the IDLE shell, or for executing import statements to import common modules.

In addition, `Tk` also loads a startup file if it is present. Note that the Tk file is loaded unconditionally. This additional file is `.Idle.py` and is looked for in the user's home directory. Statements in this file will be executed in the Tk namespace, so this file is not useful for importing functions to be used from IDLE's Python shell.

Command line usage

```
idle.py [-c command] [-d] [-e] [-h] [-i] [-r file] [-s] [-t title] [-] [arg] ...

-c command  run command in the shell window
-d          enable debugger and open shell window
-e          open editor window
-h          print help message with legal combinations and exit
-i          open shell window
-r file     run file in shell window
-s          run $IDLESTARTUP or $PYTHONSTARTUP first, in shell window
-t title    set title of shell window
-           run stdin in shell (- must be last option before args)
```

If there are arguments:

- If `-`, `-c`, or `r` is used, all arguments are placed in `sys.argv[1:...]` and `sys.argv[0]` is set to `''`, `'-c'`, or `'-r'`. No editor window is opened, even if that is the default set in the Options dialog.
- Otherwise, arguments are files opened for editing and `sys.argv` reflects the arguments passed to IDLE itself.

Startup failure

IDLE uses a socket to communicate between the IDLE GUI process and the user code execution process. A connection must be established whenever the Shell starts or restarts. (The latter is indicated by a divider line that says 'RESTART'). If the user process fails to connect to the GUI process, it displays a `Tk` error box with a 'cannot connect' message that directs the user here. It then exits.

A common cause of failure is a user-written file with the same name as a standard library module, such as *random.py* and *tkinter.py*. When such a file is located in the same directory as a file that is about to be run, IDLE cannot import the stdlib file. The current fix is to rename the user file.

Though less common than in the past, an antivirus or firewall program may stop the connection. If the program cannot be taught to allow the connection, then it must be turned off for IDLE to work. It is safe to allow this internal connection because no data is visible on external ports. A similar problem is a network mis-configuration that blocks connections.

Python installation issues occasionally stop IDLE: multiple versions can clash, or a single installation might need admin access. If one undo the clash, or cannot or does not want to run as admin, it might be easiest to completely remove Python and start over.

A zombie pythonw.exe process could be a problem. On Windows, use Task Manager to detect and stop one. Sometimes a restart initiated by a program crash or Keyboard Interrupt (control-C) may fail to connect. Dismissing the error box or Restart Shell on the Shell menu may fix a temporary problem.

When IDLE first starts, it attempts to read user configuration files in ~/.idlerc/ (~ is one's home directory). If there is a problem, an error message should be displayed. Leaving aside random disk glitches, this can be prevented by never editing the files by hand, using the configuration dialog, under Options, instead Options. Once it happens, the solution may be to delete one or more of the configuration files.

If IDLE quits with no message, and it was not started from a console, try starting from a console (`python -m idlelib`) and see if a message appears.

Running user code

With rare exceptions, the result of executing Python code with IDLE is intended to be the same as executing the same code by the default method, directly with Python in a text-mode system console or terminal window. However, the different interface and operation occasionally affect visible results. For instance, `sys.modules` starts with more entries, and `threading.activeCount()` returns 2 instead of 1.

By default, IDLE runs user code in a separate OS process rather than in the user interface process that runs the shell and editor. In the execution process, it replaces `sys.stdin`, `sys.stdout`, and `sys.stderr` with objects that get input from and send output to the Shell window. The original values stored in `sys.__stdin__`, `sys.__stdout__`, and `sys.__stderr__` are not touched, but may be `None`.

When Shell has the focus, it controls the keyboard and screen. This is normally transparent, but functions that directly access the keyboard and screen will not work. These include system-specific functions that determine whether a key has been pressed and if so, which.

IDLE's standard stream replacements are not inherited by subprocesses created in the execution process, whether directly by user code or by modules such as multiprocessing. If such subprocess use `input` from sys.stdin or `print` or `write` to sys.stdout or sys.stderr, IDLE should be started in a command line window. The secondary subprocess will then be attached to that window for input and output.

If `sys` is reset by user code, such as with `importlib.reload(sys)`, IDLE's changes are lost and input from the keyboard and output to the screen will not work correctly.

User output in Shell

When a program outputs text, the result is determined by the corresponding output device. When IDLE executes user code, `sys.stdout` and `sys.stderr` are connected to the display area of IDLE's Shell. Some of its features are inherited from the underlying Tk Text widget. Others are programmed additions. Where it matters, Shell is designed for development rather than production runs.

For instance, Shell never throws away output. A program that sends unlimited output to Shell will eventually fill memory, resulting in a memory error. In contrast, some system text windows only keep the last n lines of output. A Windows console, for instance, keeps a user-settable 1 to 9999 lines, with 300 the default.

Text widgets display a subset of Unicode, the Basic Multilingual Plane (BMP). Which characters get a proper glyph instead of a replacement box depends on the operating system and installed fonts. Newline characters cause following text to appear on a new line, but other control characters are either replaced with a box or deleted. However, `repr()`, which is used for interactive echo of expression values, replaces control characters, some BMP codepoints, and all non-BMP characters with escape codes before they are output.

Normal and error output are generally kept separate (on separate lines) from code input and each other. They each get different highlight colors.

For SyntaxError tracebacks, the normal '^' marking where the error was detected is replaced by coloring the text with an error highlight. When code run from a file causes other exceptions, one may right click on a traceback line to jump to the corresponding line in an IDLE editor. The file will be opened if necessary.

Shell has a special facility for squeezing output lines down to a 'Squeezed text' label. This is done automatically for output over N lines (N = 50 by default). N can be changed in the PyShell section of the General page of the Settings dialog. Output with fewer lines can be squeezed by right clicking on the output. This can be useful lines long enough to slow down scrolling.

Squeezed output is expanded in place by double-clicking the label. It can also be sent to the clipboard or a separate view window by right-clicking the label.

Developing tkinter applications

IDLE is intentionally different from standard Python in order to facilitate development of tkinter programs. Enter `import tkinter as tk; root = tk.Tk()` in standard Python and nothing appears. Enter the same in IDLE and a tk window appears. In standard Python, one must also enter `root.update()` to see the window. IDLE does the equivalent in the background, about 20 times a second, which is about every 50 milleseconds. Next enter `b = tk.Button(root, text='button'); b.pack()`. Again, nothing visibly changes in standard Python until one enters `root.update()`.

Most tkinter programs run `root.mainloop()`, which usually does not return until the tk app is destroyed. If the program is run with `python -i` or from an IDLE editor, a `>>>` shell prompt does not appear until `mainloop()` returns, at which time there is nothing left to interact with.

When running a tkinter program from an IDLE editor, one can comment out the mainloop call. One then gets a shell prompt immediately and can interact with the live application. One just has to remember to re-enable the mainloop call when running in standard Python.

Running without a subprocess

By default, IDLE executes user code in a separate subprocess via a socket, which uses the internal loopback interface. This connection is not externally visible and no data is sent to or received from the Internet. If firewall software complains anyway, you can ignore it.

If the attempt to make the socket connection fails, Idle will notify you. Such failures are sometimes transient, but if persistent, the problem may be either a firewall blocking the connection or misconfiguration of a particular system. Until the problem is fixed, one can run Idle with the -n command line switch.

If IDLE is started with the -n command line switch it will run in a single process and will not create the subprocess which runs the RPC Python execution server. This can be useful if Python cannot create the subprocess or the RPC socket interface on your platform. However, in this mode user code is not isolated from IDLE itself. Also, the environment is not restarted when Run/Run Module (F5) is selected. If your code has been modified, you must reload() the affected modules and re-import any specific items (e.g. from foo import baz) if the changes are to take effect. For these reasons, it is preferable to run IDLE with the default subprocess if at all possible.

Deprecated since version 3.4.

26.5.4 Help and preferences

Help sources

Help menu entry "IDLE Help" displays a formatted html version of the IDLE chapter of the Library Reference. The result, in a read-only tkinter text window, is close to what one sees in a web browser. Navigate through the text with a mousewheel, the scrollbar, or up and down arrow keys held down. Or click the TOC (Table of Contents) button and select a section header in the opened box.

Help menu entry "Python Docs" opens the extensive sources of help, including tutorials, available at docs.python.org/x.y, where 'x.y' is the currently running Python version. If your system has an off-line copy of the docs (this may be an installation option), that will be opened instead.

Selected URLs can be added or removed from the help menu at any time using the General tab of the Configure IDLE dialog .

Setting preferences

The font preferences, highlighting, keys, and general preferences can be changed via Configure IDLE on the Option menu. Non-default user settings are saved in a .idlerc directory in the user's home directory. Problems caused by bad user configuration files are solved by editing or deleting one or more of the files in .idlerc.

On the Highlights and Keys tab, select a built-in or custom color theme and key set. To use a newer built-in color theme or key set with older IDLEs, save it as a new custom theme or key set and it well be accessible to older IDLEs.

IDLE on macOS

Under System Preferences: Dock, one can set "Prefer tabs when opening documents" to "Always". This setting is not compatible with the tk/tkinter GUI framework used by IDLE, and it breaks a few IDLE features.

Extensions

IDLE contains an extension facility. Preferences for extensions can be changed with the Extensions tab of the preferences dialog. See the beginning of config-extensions.def in the idlelib directory for further information. The only current default extension is zzdummy, an example also used for testing.

26.6 Other Graphical User Interface Packages

Major cross-platform (Windows, Mac OS X, Unix-like) GUI toolkits are available for Python:

See also:

PyGObject PyGObject provides introspection bindings for C libraries using GObject. One of these libraries is the GTK+ 3 widget set. GTK+ comes with many more widgets than Tkinter provides. An online Python GTK+ 3 Tutorial is available.

PyGTK PyGTK provides bindings for an older version of the library, GTK+ 2. It provides an object oriented interface that is slightly higher level than the C one. There are also bindings to GNOME. An online tutorial is available.

PyQt PyQt is a `sip`-wrapped binding to the Qt toolkit. Qt is an extensive C++ GUI application development framework that is available for Unix, Windows and Mac OS X. `sip` is a tool for generating bindings for C++ libraries as Python classes, and is specifically designed for Python.

PySide PySide is a newer binding to the Qt toolkit, provided by Nokia. Compared to PyQt, its licensing scheme is friendlier to non-open source applications.

wxPython wxPython is a cross-platform GUI toolkit for Python that is built around the popular wxWidgets (formerly wxWindows) C++ toolkit. It provides a native look and feel for applications on Windows, Mac OS X, and Unix systems by using each platform's native widgets where ever possible, (GTK+ on Unix-like systems). In addition to an extensive set of widgets, wxPython provides classes for online documentation and context sensitive help, printing, HTML viewing, low-level device context drawing, drag and drop, system clipboard access, an XML-based resource format and more, including an ever growing library of user-contributed modules.

PyGTK, PyQt, and wxPython, all have a modern look and feel and more widgets than Tkinter. In addition, there are many other GUI toolkits for Python, both cross-platform, and platform-specific. See the GUI Programming page in the Python Wiki for a much more complete list, and also for links to documents where the different GUI toolkits are compared.