`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

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.

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.

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.

Option	Description
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.

Option	Description
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.

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).

Option	Description
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*.

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*.

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.

Option	Description
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.

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.

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.

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.

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.

Option	Description
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.

region	meaning
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.

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()

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.