Tkinter Cheat Sheet

The most popular GUI creation tool for Python, Tkinter provides a number of widgets and methods you can use to create a user interface for your application.

Tkinter Widgets

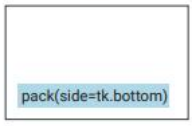
Code	from tkinter import *	from tkinter import * from tkinter.ttk import *

Widgets	Button
	instance = Button(root, text="Click me!", ...)
	
	Checkbutton
	instance = tk.Checkbutton(parent, option, ...)
	
	Entry
	instance = tk.Entry(master, option, ...)
	
	Frame
	instance = Frame(parent, option, ...)
	
	Label
	instance = tk.Label(text="some text")
	
	LabelFrame
	instance = LabelFrame(master, option, ...)
	
	Menubutton
	instance = Menubutton (master, options, ...)
	
	PanedWindow
	instance = PanedWindow(master, options, ...)
	
	Radiobutton
	instance = Radiobutton(master, options, ...)
	
	Scale
	instance = Scale (master, option, ...)
	
	Scrollbar
	instance = Scrollbar (master, options, ...)
		Combobox
		instance = ttk.Combobox(master, option=value, ...)
		
		Notebook
		instance = ttk.Notebook(container, options, ...)
		
		Progressbar
		instance = Progressbar(parent, options, ...)
		
		Separator
		# orient options are 'horizontal' or 'vertical':
		instance = ttk.Separator(container,orient='horizontal')
		
		Sizegrip
		instance = ttk.Sizegrip(master, options, ...)
		
		Treeview
		instance = ttk.Treeview(master, options, ...)

Position Widgets using `pack()`, `place()` or `grid()`

`pack()` organizes widgets in horizontal and vertical boxes that are limited to left, right, top, bottom positions. Each box is offset and relative to each other.

```
root.geometry('200x100')
test = tk.Label(root, text="pack(side=tk.bottom)", bg="teal")
test.pack(side=tk.bottom)
```

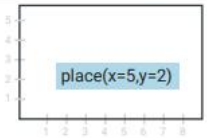


Options:

`padx` pads externally along the x axis
`pady` pads externally along the y axis
`ipadx` pads internally along the x axis
`ipady` pads internally along the y axis

`place()` places widgets in a two dimensional grid using x and y absolute coordinates.

```
root.geometry('200x100')
Label(root, text="place(x=5, y=2)",
bg="#A3DBE0").place(x=5, y=2)
```



`grid()` locates widgets in a two dimensional grid using row and column absolute coordinates.

```
root.geometry('200x100')
Label(root, text="grid(row=2, column=2)",
width=12).grid(row=2, column=2)
```

Tkinter Images with Pillow

```
# Pillow is imported as PIL
from PIL import ImageTk, Image

image1 = Image.open("<path/image_name>")
test = ImageTk.PhotoImage(image1)

label1 = tkinter.Label(image=test)
label1.image = test

# Position image as the background image
label1.place(x=1, y=1)
# Resize image to fit on button
photoimage = photo.subsample(1, 2)
# Position image on button
Button(root, image = photoimage,).pack(side = BOTTOM)
```

R

S

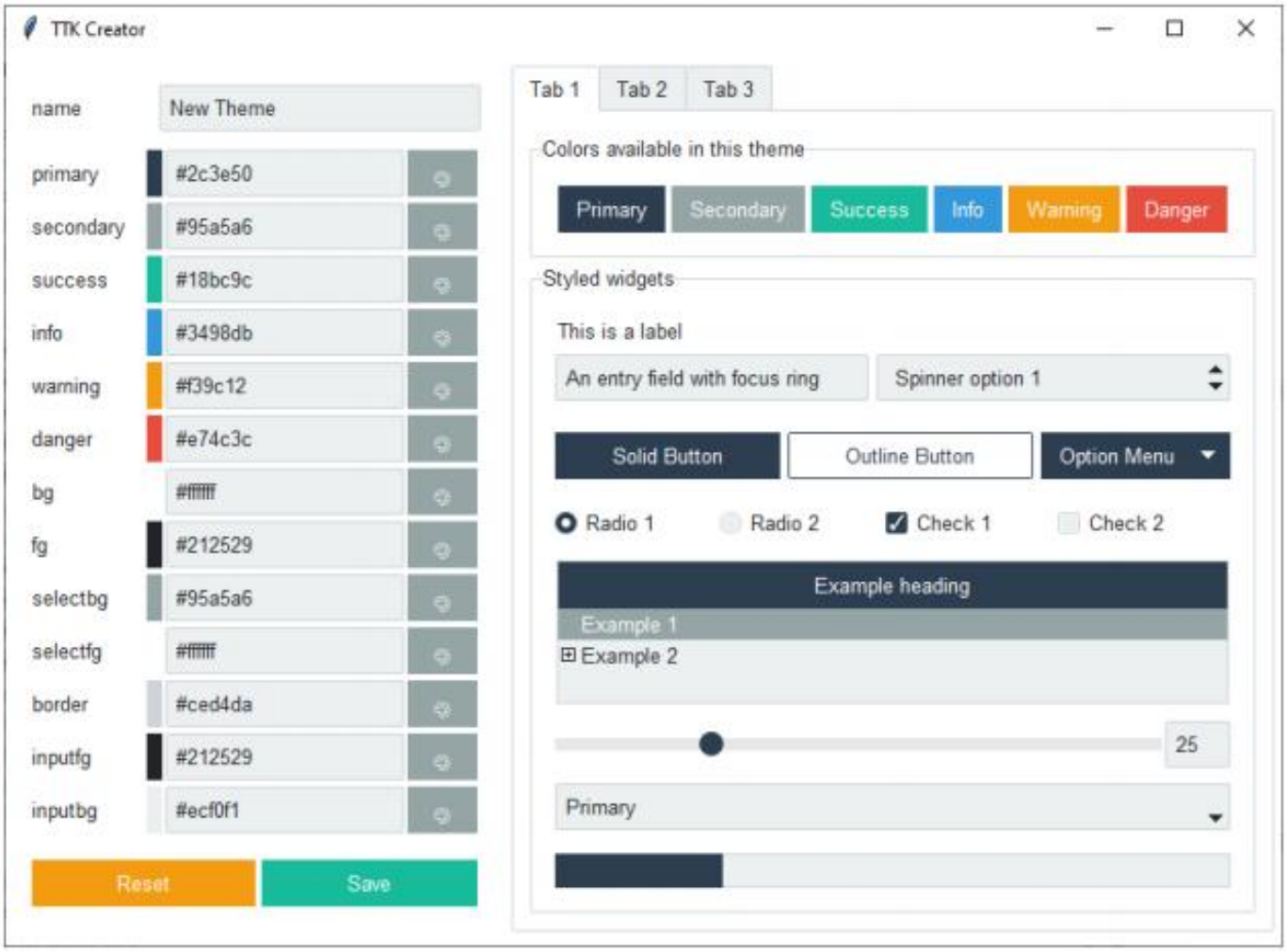
T

U

W

2.2.6 Create a new theme

TTK Creator is a program that makes it really easy to create and use your own defined themes.



2.2.6.1 Starting the application

From the console, type:

```
python -m ttkcreator
```

Binding Events

```
from Tkinter import *

def  hello(event):
   print 'Double click to exit'

def  quit(event):
   print 'caught a double click, leaving'
   import sys ; sys.exit()

widget = Button(None, text='Hello Event World')
widget.pack()
widget.bind('<Button-1>', hello)
widget.bind('<Double-1>' , quit)
widget.mainloop()
```

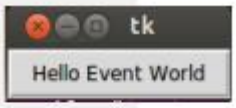


Table 1. Arguments of the `.grid()` geometry manager

`column`	The column number where you want the widget gridded, counting from zero. The default value is zero.
`columnspan`	Normally a widget occupies only one cell in the grid. However, you can grab multiple cells of a row and merge them into one larger cell by setting the `columnspan` option to the number of cells. For example, *w*`.grid(row=0, column=2, columnspan=3)` would place widget *w* in a cell that spans columns 2, 3, and 4 of row 0.
`in_`	To register *w* as a child of some widget w_2, use `in_=`w_2. The new parent w_2 must be a descendant of the *`parent`* widget used when *w* was created.
`ipadx`	Internal x padding. This dimension is added inside the widget inside its left and right sides.
`ipady`	Internal y padding. This dimension is added inside the widget inside its top and bottom borders.
`padx`	External x padding. This dimension is added to the left and right outside the widget.
`pady`	External y padding. This dimension is added above and below the widget.
`row`	The row number into which you want to insert the widget, counting from 0. The default is the next higher-numbered unoccupied row.
`rowspan`	Normally a widget occupies only one cell in the grid. You can grab multiple adjacent cells of a column, however, by setting the `rowspan` option to the number of cells to grab. This option can be used in combination with the `columnspan` option to grab a block of cells. For example, *w*`.grid(row=3, column=2, rowspan=4, columnspan=5)` would place widget *w* in an area formed by merging 20 cells, with row numbers 3–6 and column numbers 2–6.
`sticky`	This option determines how to distribute any extra space within the cell that is not taken up by the widget at its natural size. See below.

Tkinter Events and Binding

<Button-1>	- left mouse button	Mouse events
<Button-2>	- middle mouse button (on 3 button mouse)	
<Button-3>	- rightmost mouse button	
<B1-Motion>	- mouse moved with left button depressed	
<ButtonRelease-1>	- left button released	
<Double-Button-1>	- double click on button 1	
<Enter>	- mouse pointer entered widget	
<Leave>	- mouse pointer left the widget	
<FocusIn>	- Keyboard focus moved to a widget	Keyboard events
<FocusOut>	- Keyboard focus moved to another widget	
<Return>	- Enter key depressed	
<Key>	- A key was depressed	
<Shift-Up>	- Up arrow while holding Shift key	
<Configure>	- widget changed size or location	

.keysym	.keycode	.keysym_num	Key
Alt_L	64	65513	The left-hand *alt* key
Alt_R	113	65514	The right-hand *alt* key
BackSpace	22	65288	*backspace*
Cancel	110	65387	*break*
Caps_Lock	66	65549	*CapsLock*
Control_L	37	65507	The left-hand *control* key
Control_R	109	65508	The right-hand *control* key
Delete	107	65535	*Delete*
Down	104	65364	↓
End	103	65367	*end*
Escape	9	65307	*esc*
Execute	111	65378	*SysReq*
F1	67	65470	Function key *F1*
F2	68	65471	Function key *F2*
F_i	66+i	65469+i	Function key F_i
F12	96	65481	Function key *F12*
Home	97	65360	*home*
Insert	106	65379	*insert*
Left	100	65361	←
Linefeed	54	106	Linefeed (*control-J*)
KP_0	90	65438	0 on the keypad
KP_1	87	65436	1 on the keypad
KP_2	88	65433	2 on the keypad
KP_3	89	65435	3 on the keypad
KP_4	83	65430	4 on the keypad
KP_5	84	65437	5 on the keypad
KP_6	85	65432	6 on the keypad

KP_7	79	65429	7 on the keypad
KP_8	80	65431	8 on the keypad
KP_9	81	65434	9 on the keypad
KP_Add	86	65451	+ on the keypad
KP_Begin	84	65437	The center key (same key as 5) on the keypad
KP_Decimal	91	65439	Decimal (.) on the keypad
KP_Delete	91	65439	*delete* on the keypad
KP_Divide	112	65455	/ on the keypad
KP_Down	88	65433	↓ on the keypad
KP_End	87	65436	*end* on the keypad

.keysym	.keycode	.keysym_num	Key
KP_Enter	108	65421	*enter* on the keypad
KP_Home	79	65429	*home* on the keypad
KP_Insert	90	65438	*insert* on the keypad
KP_Left	83	65430	← on the keypad
KP_Multiply	63	65450	× on the keypad
KP_Next	89	65435	*PageDown* on the keypad
KP_Prior	81	65434	*PageUp* on the keypad
KP_Right	85	65432	→ on the keypad
KP_Subtract	82	65453	- on the keypad
KP_Up	80	65431	↑ on the keypad
Next	105	65366	*PageDown*
Num_Lock	77	65407	*NumLock*
Pause	110	65299	*pause*
Print	111	65377	*PrintScrn*
Prior	99	65365	*PageUp*
Return	36	65293	The *enter* key (*control-M*). The name `Enter` refers to a mouse-related event, not a keypress; see Section 54, "Events" (p. 157)
Right	102	65363	→
Scroll_Lock	78	65300	*ScrollLock*
Shift_L	50	65505	The left-hand *shift* key
Shift_R	62	65506	The right-hand *shift* key
Tab	23	65289	The *tab* key
Up	98	65362	↑

5.8. Cursors

There are quite a number of different mouse cursors available. Their names and graphics are shown here. The exact graphic may vary according to your operating system.

Table 4. Values of the `cursor` option

 `arrow`	 `man`
 `based_arrow_down`	 `middlebutton`
 `based_arrow_up`	 `mouse`
 `boat`	 `pencil`
 `bogosity`	 `pirate`
 `bottom_left_corner`	 `plus`
 `bottom_right_corner`	 `question_arrow`
 `bottom_side`	 `right_ptr`
 `bottom_tee`	 `right_side`
 `box_spiral`	 `right_tee`
 `center_ptr`	 `rightbutton`
 `circle`	 `rtl_logo`
 `clock`	 `sailboat`
 `coffee_mug`	 `sb_down_arrow`
 `cross`	 `sb_h_double_arrow`
 `cross_reverse`	 `sb_left_arrow`
 `crosshair`	 `sb_right_arrow`
 `diamond_cross`	 `sb_up_arrow`
 `dot`	 `sb_v_double_arrow`

dotbox	shuttle
double_arrow	sizing
draft_large	spider
draft_small	spraycan
draped_box	star
exchange	target
fleur	tcross
gobbler	top_left_arrow
gumby	top_left_corner
hand1	top_right_corner
hand2	top_side
heart	top_tee
icon	trek
iron_cross	ul_angle
left_ptr	umbrella
left_side	ur_angle
left_tee	watch
leftbutton	xterm
ll_angle	X_cursor
lr_angle	

27.1. How to name a widget class

For example, suppose that `Jukebox` is a new widget class that you have created. It's probably best to have new widget classes inherit from the `Frame` class, so to *Tkinter* it acts like a frame, and you can arrange other widgets such as labels, entries, and buttons inside it.

You set the new widget's class name by passing the name as the `class_` option to the parent constructor in your new class's constructor. Here is a fragment of the code that defines the new class:

```
class Jukebox(tk.Frame):
    def __init__(self, master):
        '''Constructor for the Jukebox class
        '''
        tk.Frame.__init__(self, master, class_='Jukebox')
        self.__createWidgets()
        ...
```

Here is a trivial *Tkinter* program containing only a Quit button:

```
#!/usr/bin/env python                    1
import Tkinter as tk                     2

class Application(tk.Frame):                          3
    def __init__(self, master=None):
        tk.Frame.__init__(self, master)               4
        self.grid()                                   5
        self.createWidgets()

    def createWidgets(self):
        self.quitButton = tk.Button(self, text='Quit',
            command=self.quit)                        6
        self.quitButton.grid()                        7

app = Application()                                   8
app.master.title('Sample application')                9
app.mainloop()                                        10
```

1. This line makes the script self-executing, assuming that your system has Python correctly installed.
2. This line imports the *Tkinter* module into your program's namespace, but renames it as `tk`.
3. Your application class must inherit from *Tkinter*'s `Frame` class.
4. Calls the constructor for the parent class, `Frame`.
5. Necessary to make the application actually appear on the screen.
6. Creates a button labeled "Quit".
7. Places the button on the application.
8. The main program starts here by instantiating the `Application` class.
9. This method call sets the title of the window to "Sample application".
10. Starts the application's main loop, waiting for mouse and keyboard events.

1. A minimal application

Here is a trivial Tkinter program containing only a Quit button:

```
#!/usr/local/bin/python
from Tkinter import *       # Interface to Tk widgets

class Application(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.grid()
        self.createWidgets()

    def createWidgets(self):
        self.quitButton = Button ( self, text="Quit",
            command=self.quit )
        self.quitButton.grid()

app = Application()         # Instantiate the application class
app.master.title("Sample application")
app.mainloop()              # Wait for events
```

2.2.4 Modify a style

In a large application, you may need to customize widget styles. I've done this in several of *gallery applications*. To customize a style, you need to create a `Style` object first and then use the `configure` method using the pattern `newName.oldName`. In the *File Backup Utility*, I created a custom style for a frame that used the background color of the theme border.

For this example, let's say that color is *gray*.

```
style = Style()
style.configure('custom.TFrame', background='gray')
```

This would create a frame style with the background color of gray. To apply this new style, I would create a frame and then use the *style* option to set the new style.

```
myframe = ttk.Frame(style='custom.TFrame')
```

There is a widget style class whose name is '.' By configuring this widget style class, you will change some features' default appearance for every widget that is not already configured by another style.

```
style.configure('.', font=('Helvetica', 10))
```

5.6. Relief styles

The *relief style* of a widget refers to certain simulated 3-D effects around the outside of the widget. Here is a screen shot of a row of buttons exhibiting all the possible relief styles:

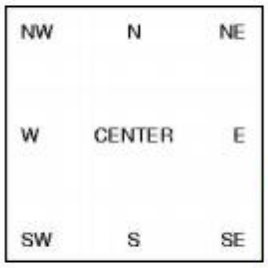
The width of these borders depends on the `borderwidth` option of the widget. The above graphic shows what they look like with a 5-pixel border; the default border width is 2.

5.7. Bitmaps

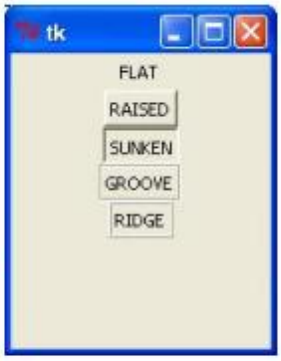
For `bitmap` options in widgets, these bitmaps are guaranteed to be available:

Standard attributes

- Dimensions
- Colors
- Fonts
- Anchors
  - used to define where text is positioned relative to a reference point.



- Relief styles
  - refers to certain simulated 3-D effects around the outside of the widget.



- Bitmaps
  - used to display a bitmap type: "error", "gray75", "gray50". "gray25", "gray12", "hourglass", "info", "questhead", "question", "warning"
- Cursors

Tk Widgets

Widget	Description
Button	Similar to a Label but provides additional functionality for mouse-overs, presses, and releases, as well as keyboard activity/events
Canvas	Provides ability to draw shapes (lines, ovals, polygons, rectangles); can contain images or bitmaps
Checkbutton	Set of boxes, of which any number can be "checked"
Entry	Single-line text field with which to collect keyboard input
Frame	Pure container for other widgets
Label	Used to contain text or images
LabelFrame	Combo of a label and a frame but with extra label attributes
Listbox	Presents the user with a list of choices from which to choose
Menu	Actual list of choices "hanging" from a Menubutton from which the user can choose
Menubutton	Provides infrastructure to contain menus (pulldown, cascading, etc.)
Message	Similar to a Label, but displays multiline text
PanedWindow	A container widget with which you can control other widgets placed within it
Radiobutton	Set of buttons, of which only one can be "pressed"
Scale	Linear "slider" widget providing an exact value at current setting; with defined starting and ending values
Scrollbar	Provides scrolling functionality to supporting widgets, for example, Text, Canvas, Listbox, and Entry
Spinbox	Combination of an entry with a button letting you adjust its value
Text	Multiline text field with which to collect (or display) text from user
Toplevel	Similar to a Frame, but provides a separate window container

2.2.3 Use themed widgets

ttkbootstrap includes many *pre-defined widget styles* that you can apply with the `style` option on ttk widgets. The style pattern is `Color.WidgetClass` where the color is a prefix to the ttk widget class. Most widgets include a style pattern for each main theme color (primary, secondary, success, info, warning, danger).

For example, the `ttk.Button` has a widget class of *TButton*. The style patterns available on the button include:

- primary.TButton
- secondary.TButton
- success.TButton
- info.TButton
- warning.TButton
- danger.TButton

These style patterns produce the following buttons:

Consider the following example, which also shows the *Outline* style that is available on buttons:

```
# solid button
ttk.Button(window, text="Submit", style='success.TButton').pack(side='left', padx=5,␣
↪pady=10)

# outline button
ttk.Button(window, text="Submit", style='success.Outline.TButton').pack(side='left',␣
↪padx=5, pady=10)
```

2.2.5 Use themed colors

ttkbootstrap has a *Colors* class that contains the theme colors as well as several helper methods for manipulating colors. This class is attached to the `Style` object at run-time for the selected theme, and so is available to use with `Style.colors`. The colors can be accessed via dot notation or get method:

```
# dot-notation
Colors.primary

# get method
Colors.get('primary')
```

This class is an iterator, so you can iterate over the main style color labels (primary, secondary, success, info, warning, danger):

```
for color_label in Colors:
    color = Colors.get(color_label)
    print(color_label, color)
```

If, for some reason, you need to iterate over all theme color labels, then you can use the `Colors.label_iter` method. This will include all theme colors, including border, fg, bg, etc...

```
for color_label in Colors.label_iter():
    color = Colors.get(color_label)
    print(color_label, color)
```

Widget class	Style name
`Button`	`TButton`
`Checkbutton`	`TCheckbutton`
`Combobox`	`TCombobox`
`Entry`	`TEntry`
`Frame`	`TFrame`
`Label`	`TLabel`
`LabelFrame`	`TLabelFrame`
`Menubutton`	`TMenubutton`
`Notebook`	`TNotebook`
`PanedWindow`	`TPanedwindow` (*not* `TPanedWindow`!)
`Progressbar`	`Horizontal.TProgressbar` or `Vertical.TProgressbar`, depending on the `orient` option.
`Radiobutton`	`TRadiobutton`
`Scale`	`Horizontal.TScale` or `Vertical.TScale`, depending on the `orient` option.
`Scrollbar`	`Horizontal.TScrollbar` or `Vertical.TScrollbar`, depending on the `orient` option.
`Separator`	`TSeparator`
`Sizegrip`	`TSizegrip`
`Treeview`	`Treeview` (*not* `TTreview`!)

At runtime, you can retrieve a widget's widget class by calling its `.winfo_class()` method.

```
>>> b=ttk.Button(None)
>>> b.winfo_class()
'TButton'
>>> t=ttk.Treeview(None)
>>> t.winfo_class()
'Treeview'
>>> b.__class__     # Here, we are asking for the Python class
<class ttk.Button at 0x21c76d0>
```

2.2.1 Simple usage

```
from ttkbootstrap import Style
from tkinter import ttk

style = Style()

window = style.master
ttk.Button(window, text="Submit", style='success.TButton').pack(side='left', padx=5,␣
↪pady=10)
ttk.Button(window, text="Submit", style='success.Outline.TButton').pack(side='left',␣
↪padx=5, pady=10)
window.mainloop()
```

This results in the window below:

If you do not create an instance of `Tk()`, the `Style` object automatically creates one. You can access this root window through the **master** property.

By default, the **flatly** theme will be applied to the application if you do not explicitly select one.

If you want to use a different theme, you can pass the style name as a keyword argument when you create the `style` object:

```
style = Style(theme='darkly')
```

7. The `Button` widget

To create a pushbutton in a top-level window or frame named *`parent`*:

```
w = tk.Button(parent, option=value, ...)
```

The constructor returns the new `Button` widget. Its options include:

Table 5. `Button` widget options

`activebackground`	Background color when the button is under the cursor.
`activeforeground`	Foreground color when the button is under the cursor.
`anchor`	Where the text is positioned on the button. See Section 5.5, "Anchors" (p. 12). For example, `anchor=tk.NE` would position the text at the top right corner of the button.

`bd` or `borderwidth`	Width of the border around the outside of the button; see Section 5.1, "Dimensions" (p. 9). The default is two pixels.
`bg` or `background`	Normal background color.
`bitmap`	Name of one of the standard bitmaps to display on the button (instead of text).
`command`	Function or method to be called when the button is clicked.
`cursor`	Selects the cursor to be shown when the mouse is over the button.
`default`	`tk.NORMAL` is the default; use `tk.DISABLED` if the button is to be initially disabled (grayed out, unresponsive to mouse clicks).
`disabledforeground`	Foreground color used when the button is disabled.
`fg` or `foreground`	Normal foreground (text) color.
`font`	Text font to be used for the button's label.
`height`	Height of the button in text lines (for textual buttons) or pixels (for images).
`highlightbackground`	Color of the focus highlight when the widget does not have focus.
`highlightcolor`	The color of the focus highlight when the widget has focus.
`highlightthickness`	Thickness of the focus highlight.
`image`	Image to be displayed on the button (instead of text).
`justify`	How to show multiple text lines: `tk.LEFT` to left-justify each line; `tk.CENTER` to center them; or `tk.RIGHT` to right-justify.
`overrelief`	The relief style to be used while the mouse is on the button; default relief is `tk.RAISED`. See Section 5.6, "Relief styles" (p. 12).
`padx`	Additional padding left and right of the text. See Section 5.1, "Dimensions" (p. 9) for the possible values for padding.
`pady`	Additional padding above and below the text.
`relief`	Specifies the relief type for the button (see Section 5.6, "Relief styles" (p. 12)). The default relief is `tk.RAISED`.
`repeatdelay`	See `repeatinterval`, below.
`repeatinterval`	Normally, a button fires only once when the user releases the mouse button. If you want the button to fire at regular intervals as long as the mouse button is held down, set this option to a number of milliseconds to be used between repeats, and set the `repeatdelay` to the number of milliseconds to wait before starting to repeat. For example, if you specify "`repeatdelay=500, repeatinterval=100`" the button will fire after half a second, and every

`text`	Text displayed on the button. Use internal newlines to display multiple text lines.
`textvariable`	An instance of `StringVar()` that is associated with the text on this button. If the variable is changed, the new value will be displayed on the button. See Section 52, "Control variables: the values behind the widgets" (p. 153).
`underline`	Default is `-1`, meaning that no character of the text on the button will be underlined. If nonnegative, the corresponding text character will be underlined. For example, `underline=1` would underline the second character of the button's text.
`width`	Width of the button in letters (if displaying text) or pixels (if displaying an image).
`wraplength`	If this value is set to a positive number, the text lines will be wrapped to fit within this length. For possible values, see Section 5.1, "Dimensions" (p. 9).

Methods on `Button` objects:

`.flash()`
Causes the button to flash several times between active and normal colors. Leaves the button in the state it was in originally. Ignored if the button is disabled.

`.invoke()`
Calls the button's `command` callback, and returns what that function returns. Has no effect if the button is disabled or there is no callback.

2.4.1.3 Style configuration

Use the following classes, states, and options when configuring or modifying a new ttk button style. See the python style documentation for more information on creating a style.

Create a new theme using TTK Creator if you want to change the default color scheme.

Class names

- TButton
- Outline.TButton
- Link.TButton

Dynamic states

- active
- disabled
- pressed
- readonly

Example: Calendar

```
# -*- coding: utf-8 -*-

# Copyright (c) Juliette Monsel 2018
# For license see LICENSE

from ttkwidgets import Calendar
import tkinter as tk

def validate():
    sel = calendar.selection
    if sel is not None:
        label.configure(text='Selected date: %s' % sel.strftime('%x'))

window = tk.Tk()
calendar = Calendar(window, year=2015, month=3, selectforeground='white',
                    selectbackground='red')
calendar.pack()

tk.Button(window, text='Select', command=validate).pack()
label = tk.Label(window, text='Selected date:')
label.pack()
window.mainloop()
```

5.2.2.1 ask_date

ttkbootstrap.widgets.calendar.**ask_date**(*parent=None*, *startdate=None*, *firstweekday=6*, *style='TCalendar'*)

Generate a popup date chooser and return the selected date

Parameters

- **parent** (*Widget*) – The parent widget; the popup will appear to the bottom-right of the parent widget. If no parent is provided, the widget is centered on the screen.
- **firstweekday** (*int*) – Specifies the first day of the week. `0` is Monday, `6` is Sunday (the default).
- **startdate** (*datetime*) – The date to be in focus when the widget is displayed; defaults to the current date.
- **style** (*str*) – The `ttk` style used to render the widget.

Returns The date selected; the current date if no date is selected.

5.2.2.2 DateChooserPopup

class ttkbootstrap.widgets.calendar.**DateChooserPopup**(*parent=None*, *firstweekday=6*, *startdate=None*, *style='TCalendar'*)

Bases: `object`

A custom **ttkbootstrap** widget that displays a calendar and allows the user to select a date which is returned as a `datetime` object for the date selected.

The widget displays the current date by default unless a `startdate` is provided. The month can be changed by clicking on the chevrons to the right and left of the month-year title which is displayed on the top-center of the widget. A "left-click" will move the calendar *one month*. A "right-click" will move the calendar *one year*.

A "right-click" on the *month-year* title will reset the calendar widget to the starting date.

The starting weekday can be changed with the `firstweekday` parameter for geographies that do not start the week on *Sunday*, which is the widget default.

The widget grabs focus and all screen events until released. If you want to cancel a date selection, you must click on the "X" button at the top-right hand corner of the widget.

Styles can be applied to the widget by using the *TCalendar* style with the optional colors: 'primary', 'secondary', 'success', 'info', 'warning', and 'danger'. By default, the *primary.TCalendar* style is applied.

Parameters

- **parent** (*Widget*) – The parent widget; the popup is displayed to the bottom-right of the parent widget.
- **startdate** (*datetime*) – The date to be in focus when the calendar is displayed. Current date is default.
- **firstweekday** (*int*) – Specifies the first day of the week. `0` is Monday, `6` is Sunday (the default).
- **style** (`str`) – The `ttk` style used to render the widget.
- ****kw** –

draw_calendar()
Create the days of the week elements

draw_titlebar()
Create the title bar

generate_widget_styles()
Generate all the styles required for this widget from the `base_style`.

on_date_selected(*index*)
Callback for selecting a date.

Assign the selected date to the `date_selected` property and then destroy the toplevel widget.

Parameters **index** (*Tuple[int]*) – a tuple containing the row and column index of the date selected to be found in the `monthdates` property.

on_next_month()
Callback for changing calendar to next month

on_next_year(**args*)
Callback for changing calendar to next year

on_prev_month()
Callback for changing calendar to previous month

on_prev_year(**args*)
Callback for changing calendar to previous year

on_reset_date(**args*)
Callback for clicking the month-year title; reset the date to the start date

set_geometry()
Adjust the window size based on the number of weeks in the month

setup()
Setup the calendar widget

weekday_header()
Creates and returns a list of weekdays to be used as a header in the calendar based on the firstweekday. The order of the weekdays is based on the `firstweekday` property.

Returns a list of weekday headers

2.4.3.3 Style configuration

Use the following classes, states, and options when configuring or modifying a new calendar style. Some options are only available in certain styles. See the python style documentation for more information on creating a style.

Create a new theme using TTK Creator if you want to change the default color scheme.

Class names

- TCalendar

Dynamic states

- active
- alternate
- disabled
- pressed
- selected
- readonly

Table 18. Callback substitution codes

`'%d'`	Action code: 0 for an attempted deletion, 1 for an attempted insertion, or -1 if the callback was called for focus in, focus out, or a change to the `textvariable`.

`'%i'`	When the user attempts to insert or delete text, this argument will be the index of the beginning of the insertion or deletion. If the callback was due to focus in, focus out, or a change to the `textvariable`, the argument will be `-1`.
`'%P'`	The value that the text will have if the change is allowed.
`'%s'`	The text in the entry before the change.
`'%S'`	If the call was due to an insertion or deletion, this argument will be the text being inserted or deleted.
`'%v'`	The current value of the widget's `validate` option.
`'%V'`	The reason for this callback: one of `'focusin'`, `'focusout'`, `'key'`, or `'forced'` if the `textvariable` was changed.
`'%W'`	The name of the widget.

2.4.2.3 Style configuration

Use the following classes, states, and options when configuring or modifying a new ttk checkbutton style. TTK Bootstrap uses an image layout for this widget, so not all of these options will be available... for example: `indicatormargin`. However, if you decide to create a new widget, these should be available, depending on the style you are using as a base. Some options are only available in certain styles. See the python style documentation for more information on creating a style.

Create a new theme using TTK Creator if you want to change the default color scheme.

Class names

- TCheckbutton
- Toolbutton
- Outline.Toolbutton
- Roundtoggle.Toolbutton
- Squaretoggle.Toolbutton

Dynamic states

- active
- alternate
- disabled
- pressed
- selected

```
# -*- coding: utf-8 -*-

# Copyright (c) Juliette Monsel 2017
# For license see LICENSE

from ttkwidgets import CheckboxTreeview
import tkinter as tk

root = tk.Tk()

tree = CheckboxTreeview(root)
tree.pack()

tree.insert("", "end", "1", text="1")
tree.insert("1", "end", "11", text="11")
tree.insert("1", "end", "12",  text="12")
tree.insert("11", "end", "111", text="111")
tree.insert("", "end", "2", text="2")
```

(continues on next page)

2.4.4.3 Style configuration

Use the following classes, states, and options when configuring or modifying a new ttk combobox style. See the python style documentation for more information on creating a style.

Create a new theme using TTK Creator if you want to change the default color scheme.

Class names

- TCombobox

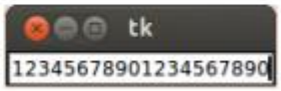
Dynamic states

- disabled
- focus
- pressed
- readonly

Entry

- used to enter or display a single line of text
  - To enter multiple lines of text, use the Text widget.


```
from Tkinter import *
master = Tk()
e = Entry(master)
e.pack()
mainloop()
```



options:

- anchor
- aspect
- background/bg
- borderwidth/bd
- cursor
- font
- foreground/fg
- highlightbackground
- highlightcolor
- highlightthickness
- relief
- takefocus
- text
- textvariable
- width
- validate
- validatecommand
- justify

methods:

- get()
- set()
- delete(first,last/END)
- insert(index)
- insert(index,string)
- icursor(index)
- index(index)

5.2.2.3 DateEntry

class ttkbootstrap.widgets.calendar.**DateEntry**(*master=None*, *dateformat='%Y-%m-%d'*, *firstweekday=6*, *startdate=None*, *style='TCalendar'*, ***kw*)

Bases: tkinter.ttk.Frame

A date entry widget that combines a ttk.Combobox and a ttk.Button`` with a callback attached to the ask_date function.

When pressed, displays a date chooser popup and then inserts the returned value into the combobox.

Optionally set the startdate of the date chooser popup by typing in a date that is consistent with the format that you have specified with the dateformat parameter. By default this is *%Y-%m-%d*.

Change the style of the widget by using the *TCalendar* style, with the colors: 'primary', 'secondary', 'success', 'info', 'warning', 'danger'. By default, the *primary.TCalendar* style is applied.

Change the starting weekday with the firstweekday parameter for geographies that do not start the week on *Sunday*, which is the widget default.

Parameters

- **master** (*Widget*) – The parent widget.
- **dateformat** (*str*) – The format string used to render the text in the entry widget. Default is '%Y-%m-%d'. For more information on date formats, see the python documentation or https://strftime.org/.
- **firstweekday** (*int*) – Specifies the first day of the week. 0 is Monday, 6 is Sunday (the default).
- **startdate** (*datetime*) – The date to be in focus when the calendar is displayed. Current date is default.
- ****kw** – Optional keyword arguments to be passed to containing frame widget.

convert_system_color(*systemcolorname*)

Convert a system color name to a hexadecimal value

Parameters **systemcolorname** (*str*) – a system color name, such as *SystemButtonFace*

convert_system_color(*systemcolorname*)
Convert a system color name to a hexadecimal value

Parameters **systemcolorname** (*str*) – a system color name, such as *SystemButtonFace*

draw_button_image(*color*)
Draw a calendar button image of the specified color

Image reference: https://www.123rf.com/photo_117523637_stock-vector-modern-icon-calendar-button-applications.html

Parameters **color** (*str*) – the color to draw the image foreground.

Returns the image created for the calendar button.

Return type PhotoImage

generate_widget_styles()
Generate all the styles required for this widget from the base_style.

Returns the styles to be used for entry and button widgets.

Return type Tuple[str]

on_date_ask()
A callback for the date push button.

Try to grab the initial date from the entry if possible. However, if this date is not valid, use the current date and print a warning message to the console.

2.4.5.2 How to use

The examples below demonstrate how to *use a style* to create a widget. To learn more about how to *use the widget in ttk*, check out the *References* section for links to documentation and tutorials on this widget.

Create a default **entry**

```
entry = ttk.Entry(parent)

entry.insert('Hello world!')
```

Create an **'info' entry**

```
ttk.Entry(parent, style='info.TEntry')
```

2.4.5.3 Style configuration

Use the following classes, states, and options when configuring or modifying a new `ttk.Entry` style. See the python style documentation for more information on creating a style.

Create a new theme using TTK Creator if you want to change the default color scheme.

Class names

- TEntry

Dynamic states

- disabled
- focus
- readonly

Example: ItemsCanvas

```
# -*- coding: utf-8 -*-

# Copyright (c) RedFantom 2017
# For license see LICENSE
from ttkwidgets import ItemsCanvas
import tkinter as tk
from tkinter import ttk


root = tk.Tk()

canvas = ItemsCanvas(root)
canvas.pack()

canvas.add_item("Example", font=("default", 13, "italic"), backgroundcolor="green",
↪textcolor="darkblue",
                highlightcolor="blue")

root.mainloop()
```

2.4.8.3 Style configuration

Use the following classes, states, and options when configuring or modifying a new ttk label style. See the python style documentation for more information on creating a style.

Create a new theme using TTK Creator if you want to change the default color scheme.

Class names

- TLabel
- Inverse.TLabel

Dynamic states

- disabled
- readonly

2.4.9.3 Style configuration

Use the following classes, states, and options when configuring or modifying a new ttk labelframe style. See the python style documentation for more information on creating a style.

Create a new theme using TTK Creator if you want to change the default color scheme.

Class names

- TLabelframe

Dynamic states

- disabled
- readonly

```
    self.mb = tk.Menubutton(self, text='condiments',
                            relief=RAISED)
    self.mb.grid()

    self.mb.menu = tk.Menu(self.mb, tearoff=0)
    self.mb['menu'] = self.mb.menu

    self.mayoVar  = tk.IntVar()
    self.ketchVar = tk.IntVar()
    self.mb.menu.add_checkbutton(label='mayo',
        variable=self.mayoVar)
    self.mb.menu.add_checkbutton(label='ketchup',
        variable=self.ketchVar)
```

2.4.10.2 How to use

The examples below demonstrate how to *use a style* to create a widget. To learn more about how to *use the widget in ttk*, check out the *References* section for links to documentation and tutorials on this widget.

Create an **info outline menubutton**

```
mb = ttk.Menubutton(parent, text='My widgets', style='info.Outline.TMenubutton')

# create menu
menu = tk.Menu(mb)

# add options
option_var = tk.StringVar()
for option in ['option 1', 'option 2', 'option 3']:
    menu.add_radiobutton(label=option, value=option, variable=option_var)

# associate menu with menubutton
mb['menu'] = menu
```

2.4.10.3 Style configuration

Use the following classes, states, and options when configuring or modifying a new ttk menubutton style. See the python style documentation for more information on creating a style.

Create a new theme using TTK Creator if you want to change the default color scheme.

Class names

- TMenubutton
- Outline.TMenubutton

Dynamic states

- active
- disabled
- readonly

```
# create a new notebook
nb = ttk.Notebook(parent)

# create a new frame
frame = ttk.Frame(nb)

# set the frame as a tab in the notebook
nb.add(frame, text='Tab 1')
```


2.4.12.3 Configuration

Use the following classes, states, and options when configuring or modifying a new ttk notebook style. See the python style documentation for more information on creating a style.

Create a new theme using TTK Creator if you want to change the default color scheme.

Class names

- TNotebook
- TNotebook.Tab

Dynamic states

- active
- disabled
- selected

Create and use a **Paned Window**

```
# create a new paned window
pw = ttk.PanedWindow(parent, orient='horizontal')

# add something on the left side
left_frame = ttk.Frame(pw)
left_frame.pack(side='left', fill='both')

# add something on the right side
right_frame = ttk.Frame(pw)
right_frame.pack(side='left', fill='both')

# add the frames to the paned window; a sash will appear between each frame (see image␣
↪above)
```

(continues on next page)

(continued from previous page)

```
pw.add(left_frame)
pw.add(right_frame)
```

2.4.13.3 Configuration

Use the following classes, states, and options when configuring or modifying a new ttk paned window style. See the python style documentation for more information on creating a style.

Create a new theme using TTK Creator if you want to change the default color scheme.

Class names

- TPanedwindow
- Sash

```
ttk.Progressbar(parent, value=75)
```

Create a default **vertical progressbar**

```
ttk.Progressbar(parent, value=75, orient='vertical')
```

Create a default **horizontal striped progressbar**

```
ttk.Progressbar(parent, value=75, style='Striped.Horizontal.TProgressbar')
```

Create a **success horizontal striped progressbar**

```
ttk.Progressbar(parent, value=75, style='success.Striped.Horizontal.TProgressbar')
```


2.4.14.3 Configuration

Use the following classes, states, and options when configuring or modifying a new ttk progressbar style. The *Striped.Horizontal.TProgressbar* is an image-based layout, so the styling options will be limited to those which affect the *trough*. The regular progressbar styles can be configured with all available options. See the python style documentation for more information on creating a style.

Create a new theme using TTK Creator if you want to change the default color scheme.

Class names

- Horizontal.TProgressbar
- Vertical.TProgressbar
- Striped.Horizontal.TProgressbar

Create a **'warning' outline toolbutton**

```
ttk.Radiobutton(parent, text="option 5", style='warning.Outline.Toolbutton')
```

2.4.15.3 Configuration

Use the following classes, states, and options when configuring or modifying a new ttk radiobutton style. TTK Bootstrap uses an image layout for the **TRadiobutton** style on this widget, so not all of these options will be available... for example: `indicatormargin`. However, if you decide to create a new widget, these should be available, depending on the style you are using as a base. Some options are only available in certain styles. See the python style documentation for more information on creating a style.

Create a new theme using TTK Creator if you want to change the default color scheme.

Class names

- TRadiobutton
- Toolbutton
- Outline.Toolbutton

Dynamic states

- active
- alternate
- disabled
- pressed
- selected
- readonly

Example: ScaleEntry

```
# -*- coding: utf-8 -*-

# Copyright (c) Juliette Monsel 2018
# For license see LICENSE

from ttkwidgets import ScaleEntry
import tkinter as tk


window = tk.Tk()
scaleentry = ScaleEntry(window, scalewidth=200, entrywidth=3, from_=0, to=20)
scaleentry.config_entry(justify='center')
scaleentry.pack()
window.mainloop()
```

The `ttk.Scale` includes the **Horizontal.TScale** and **Vertical.TScale** style classes. These styles are further subclassed by each of the theme colors to produce the following color and style combinations:

2.4.16.2 How to use

The examples below demonstrate how to *use a style* to create a widget. To learn more about how to *use the widget in ttk*, check out the *References* section for links to documentation and tutorials on this widget.

Create a default **horizontal scale**

```
ttk.Scale(parent, from_=0, to=100, value=75)
```

Create a default **vertical scale**

```
ttk.Scale(parent, from_=0, to=100, value=75, orient='vertical')
```

22.2. Connecting a `Scrollbar` to another widget

Here is a code fragment showing the creation of a canvas with horizontal and vertical scrollbars. In this fragment, `self` is assumed to be a `Frame` widget.

```
self.canv = tk.Canvas(self, width=600, height=400,
    scrollregion=(0, 0, 1200, 800))
self.canv.grid(row=0, column=0)

self.scrollY = tk.Scrollbar(self, orient=tk.VERTICAL,
    command=self.canv.yview)
self.scrollY.grid(row=0, column=1, sticky=tk.N+tk.S)

self.scrollX = tk.Scrollbar(self, orient=tk.HORIZONTAL,
    command=self.canv.xview)
self.scrollX.grid(row=1, column=0, sticky=tk.E+tk.W)
```



```
self.canv['xscrollcommand'] = self.scrollX.set
self.canv['yscrollcommand'] = self.scrollY.set
```

7.1 Scrolling an `Entry` widget

Making an `Entry` widget scrollable requires a little extra code on your part to adapt the `Scrollbar` widget's callback to the methods available on the `Entry` widget. Here are some code fragments illustrating the setup. First, the creation and linking of the `Entry` and `Scrollbar` widgets:

```
self.entry = Entry ( self, width=10 )
self.entry.grid(row=0, sticky=E+W)

self.entryScroll = Scrollbar ( self, orient=HORIZONTAL,
    command=self.__scrollHandler )
self.entryScroll.grid(row=1, sticky=E+W)
self.entry["xscrollcommand"] = self.entryScroll.set
```

Here's the adapter function referred to above:

```
def __scrollHandler(self, *L):
    op, howMany = L[0], L[1]

    if op == "scroll":
        units  =  L[2]
        self.entry.xview_scroll ( howMany, units )
    elif op == "moveto":
        self.entry.xview_moveto ( howMany )
```

Example: AutoHideScrollbar

```
# -*- coding: utf-8 -*-

# Copyright (c) Juliette Monsel 2018
# For license see LICENSE

from ttkwidgets import AutoHideScrollbar
import tkinter as tk

window = tk.Tk()
listbox = tk.Listbox(window, height=5)
scrollbar = AutoHideScrollbar(window, command=listbox.yview)
listbox.configure(yscrollcommand=scrollbar.set)

for i in range(10):
    listbox.insert('end', 'item %i' % i)
```

(continues on next page)

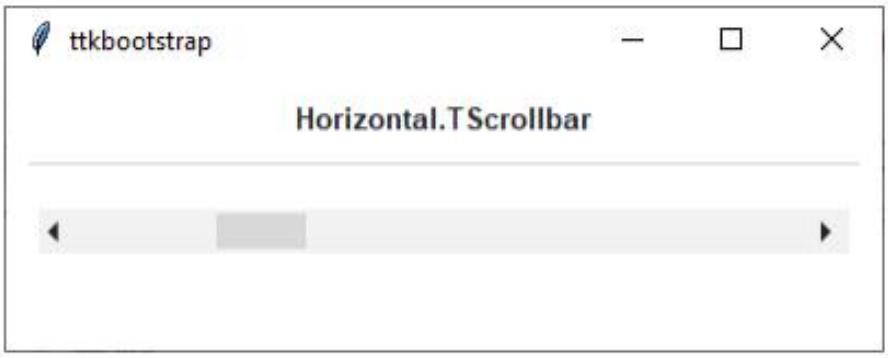
(continued from previous page)

```
tk.Label(window, text="Increase the window's height\nto make the scrollbar vanish.").
↪pack(side='top', padx=4, pady=4)
scrollbar.pack(side='right', fill='y')
listbox.pack(side='left', fill='both', expand=True)

window.mainloop()
```

Example: ScrolledListbox

```
# -*- coding: utf-8 -*-

# Copyright (c) Juliette Monsel 2018
# For license see LICENSE

from ttkwidgets import ScrolledListbox
import tkinter as tk

window = tk.Tk()
listbox = ScrolledListbox(window, height=5)

for i in range(10):
    listbox.listbox.insert('end', 'item {}'.format(i))

listbox.pack(fill='both', expand=True)
window.mainloop()
```

The `ttk.Scrollbar` includes the **Horizontal.TScrollbar** and **Vertical.TScrollbar** style classes. These styles are applied by default to *horizontal* and *vertical* orientations. So there is no need to specify the styles unless you decide to create a new custom style.



2.4.17.2 How to use

The examples below demonstrate how to *use a style* to create a widget. To learn more about how to *use the widget in ttk*, check out the *References* section for links to documentation and tutorials on this widget.

Create a default **horizontal scrollbar**

```
ttk.Scrollbar(parent, orient='horizontal')
```

Create a default **vertical scrollbar**

```
ttk.Scrollbar(parent, orient='vertical')
```

Example: ScrolledFrame

```
# -*- coding: utf-8 -*-

# Copyright (c) Juliette Monsel 2018
# For license see LICENSE

from ttkwidgets.frames import ScrolledFrame
import tkinter as tk
from tkinter import ttk

window = tk.Tk()
frame = ScrolledFrame(window, compound=tk.RIGHT, canvasheight=200)
frame.pack(fill='both', expand=True)

for i in range(20):
    ttk.Label(frame.interior, text='Label %i' % i).pack()
window.mainloop()
```

43. *ttk*.Separator

Use this widget to place a horizontal or vertical bar that separates other widgets. The widget is rendered as a 2-pixel wide line. Be sure to use the `sticky` options to the `.grid()` method to stretch the widget, or it will appear as only a single pixel.

To create a *ttk*.`Separator` as the child of a given *`parent`* widget, where the *`option`* values are given in Table 61, "*ttk*.`Separator` options" (p. 137):

```
w = ttk.Separator(parent, option=value, ...)
```

Table 61. *ttk*.Separator options

`class_`	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, "Standardizing appearance" (p. 105).
`orient`	Set `orient=tk.HORIZONTAL` for a horizontal separator, `orient=tk.VERTICAL` for a vertical one (the default orientation).
`style`	The style to be used in rendering this scrollbar; see Section 49, "Using and customizing *ttk* styles" (p. 147). The only style feature you can configure is `background`, which specifies the color of the separator bar; the default color is a dark gray.

The only methods available on a *ttk*.`Separator` widgets are the ones listed in Section 46, "Methods common to all *ttk* widgets" (p. 145).

2.4.18.2 How to use

The examples below demonstrate how to *use a style* to create a widget. To learn more about how to *use the widget in ttk*, check out the *References* section for links to documentation and tutorials on this widget.

Create a default **horizontal separator**

```
ttk.Separator(parent, orient='horizontal')
```

Create a default **vertical separator**

```
ttk.Separator(parent, orient='vertical')
```

Create an **info vertical separator**

2.4.19.2 How to use

The examples below demonstrate how to *use a style* to create a widget. To learn more about how to *use the widget in ttk*, check out the *References* section for links to documentation and tutorials on this widget.

Create a default **sizegrip**

```
ttk.Sizegrip(parent)
```

Create a **success sizegrip**

```
ttk.Sizegrip(parent, style='success.TSizegrip')
```

2.4.20.2 How to use

The examples below demonstrate how to *use a style* to create a widget. To learn more about how to *use the widget in ttk*, check out the *References* section for links to documentation and tutorials on this widget.

Create a default **spinbox**

```
cb = ttk.Spinbox(parent, from_=1, to=100)
```

Create an **'info' spinbox**

```
ttk.Spinbox(parent, from_=1, to=100, style='info.TSpinbox')
```

2.4.20.3 Configuration

Use the following classes, states, and options when configuring or modifying a new ttk spinbox style. Or, See the python style documentation for more information on creating a style.

create a new theme using TTK Creator if you want to change the default color scheme.

Class names

- TSpinbox

Dynamic states

- active
- disabled
- focus
- readonly

register_theme(*definition*)

Registers a theme definition for use by the Style class.

This makes the definition and name available at run-time so that the assets and styles can be created.

Parameters **definition** (ThemeDefinition) – an instance of the ThemeDefinition class

theme_create(*themename*, *parent=None*, *settings=None*)

Creates 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 specified parent theme. If settings are present, they are expected to have the same syntax used for theme_settings.

theme_names()

Returns a list of all known themes.

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 configure, map, layout and element_create respectively.

```
from ttkwidgets import Table
import tkinter as tk
from tkinter import ttk

root = tk.Tk()

root.columnconfigure(0, weight=1)
root.rowconfigure(0, weight=1)

style = ttk.Style(root)
style.theme_use('alt')
sortable = tk.BooleanVar(root, False)
drag_row = tk.BooleanVar(root, False)
drag_col = tk.BooleanVar(root, False)

columns = ["A", "B", "C", "D", "E", "F", "G"]
table = Table(root, columns=columns, sortable=sortable.get(), drag_cols=drag_col.
↪get(),
              drag_rows=drag_row.get(), height=6)
for col in columns:
    table.heading(col, text=col)
    table.column(col, width=100, stretch=False)

# sort column A content as int instead of strings
table.column('A', type=int)

for i in range(12):
    table.insert('', 'end', iid=i,
                 values=(i, i) + tuple(i + 10 * j for j in range(2, 7)))

# add scrollbars
sx = tk.Scrollbar(root, orient='horizontal', command=table.xview)
sy = tk.Scrollbar(root, orient='vertical', command=table.yview)
table.configure(yscrollcommand=sy.set, xscrollcommand=sx.set)

table.grid(sticky='ewns')
sx.grid(row=1, column=0, sticky='ew')
sy.grid(row=0, column=1, sticky='ns')
root.update_idletasks()


# toggle table properties
def toggle_sort():
    table.config(sortable=sortable.get())
```

```
def toggle_drag_col():
    table.config(drag_cols=drag_col.get())


def toggle_drag_row():
    table.config(drag_rows=drag_row.get())


frame = tk.Frame(root)
```

(continues on next page)

(continued from previous page)

```
tk.Checkbutton(frame, text='sortable', variable=sortable, command=toggle_sort).
↪pack(side='left')
tk.Checkbutton(frame, text='drag columns', variable=drag_col, command=toggle_drag_
↪col).pack(side='left')
tk.Checkbutton(frame, text='drag rows', variable=drag_row, command=toggle_drag_row).
↪pack(side='left')
frame.grid()
root.geometry('400x200')

root.mainloop()
```

15.2. Top-level menus

Especially under MacOS, it is sometimes desirable to create menus that are shown as part of the top-level window. To do this, follow these steps.

1. Using any widget `W`, obtain the top-level window by using the `W.winfo_toplevel()` method.
2. Create a `Menu` widget, using the top-level window as the first argument.
3. Items added to this `Menu` widget will be displayed across the top of the application.

Here is a brief example. Assume that `self` is the application instance, an instance of a class that inherits from `Frame`. This code would create a top-level menu choice named *"Help"* with one choice named *"About"* that calls a handler named `self.__aboutHandler`:

```
top = self.winfo_toplevel()
self.menuBar = tk.Menu(top)
top['menu'] = self.menuBar

self.subMenu = tk.Menu(self.menuBar)
self.menuBar.add_cascade(label='Help', menu=self.subMenu)
self.subMenu.add_command(label='About', command=self.__aboutHandler)
```

There is some variation in behavior depending on your platform.

- Under Windows or Unix systems, the top-level menu choices appear at the top of your application's main window.
- Under MacOS X, the top-level menu choices appear at the top of the screen when the application is active, right where Mac users expect to see them.

  You must use the `.add_cascade()` method for all the items you want on the top menu bar. Calls to `.add_checkbutton()`, `.add_command()`, or `.add_radiobutton()` will be ignored.

2.4.21.2 How to use

The examples below demonstrate how to *use a style* to create a widget. To learn more about how to *use the widget in ttk*, check out the *References* section for links to documentation and tutorials on this widget.

Create a default **treeview**

```
cb = ttk.Treeview(parent, columns=[1, 2, 3], show='headings')
```

Create an **'info' treeview**

```
ttk.Treeview(parent, columns=[1, 2, 3], show='headings', style='info.Treeview')
```

2.4.21.3 Configuration

Use the following classes, states, and options when configuring or modifying a new ttk separator style. See the python style documentation for more information on creating a style.

Create a new theme using TTK Creator if you want to change the default color scheme.

Class names

- Treeview
- Heading
- Item
- Cell

55.1. The tkMessageBox dialogs module

Once you import the tkMessageBox module, you can create any of these seven common types of pop-up menu by calling functions from this table.

	`.askokcancel(title, message, options)`
	`.askquestion(title, message, options)`
	`.askretrycancel(title, message, options)`
	`.askyesno(title, message, options)`

	`.showerror(`*`title, message, options`*`)`
	`.showinfo(`*`title, message, options`*`)`
	`.showwarning(`*`title, message, options`*`)`

In each case, the *`title`* is a string to be displayed in the top of the window decoration. The *`message`* argument is a string that appears in the body of the pop-up window; within this string, lines are broken at newline (`'\n'`) characters.

The *`option`* arguments may be any of these choices.

`default`
: Which button should be the default choice? If you do not specify this option, the first button ("OK", "Yes", or "Retry") will be the default choice.

  To specify which button is the default choice, use `default=`*`C`*, where *`C`* is one of these constants defined in `tkMessageBox`: `CANCEL`, `IGNORE`, `OK`, `NO`, `RETRY`, or `YES`.

`icon`
: Selects which icon appears in the pop-up. Use an argument of the form `icon=`*`I`* where *`I`* is one of these constants defined in `tkMessageBox`: `ERROR`, `INFO`, `QUESTION`, or `WARNING`.

`parent`
: If you don't specify this option, the pop-up appears above your root window. To make the pop-up appear above some child window *`W`*, use the argument `parent=`*`W`*.

Each of the "`ask...`" pop-up functions returns a value that depends on which button the user pushed to remove the pop-up.

Example: AutocompleteCombobox

```
# -*- coding: utf-8 -*-

# Copyright (c) Juliette Monsel 2018
# For license see LICENSE

from ttkwidgets.autocomplete import AutocompleteCombobox
import tkinter as tk

window = tk.Tk()
tk.Label(window, text="Combobox with autocompletion for the Tk instance's methods:").
↪pack(side='left')
entry = AutocompleteCombobox(window, width=20, completevalues=dir(window))
entry.pack(side='right')
window.mainloop()
```

Example: AutocompleteEntry

```
# -*- coding: utf-8 -*-

# Copyright (c) Juliette Monsel 2018
# For license see LICENSE

from ttkwidgets.autocomplete import AutocompleteEntry
import tkinter as tk

window = tk.Tk()
tk.Label(window, text="Entry with autocompletion for the Tk instance's methods:").
↪pack(side='left')
entry = AutocompleteEntry(window, width=20, completevalues=dir(window))
entry.pack(side='right')
window.mainloop()
```

Example: AutocompleteEntryListbox

```
# -*- coding: utf-8 -*-

# Copyright (c) Juliette Monsel 2019
# For license see LICENSE
import tkinter as tk
from ttkwidgets.autocomplete import AutocompleteEntryListbox

window = tk.Tk()
tk.Label(window, text="Entry + Listbox with autocompletion for the Tk instance's␣
↪methods:").pack()
entry = AutocompleteEntryListbox(window, width=20, completevalues=dir(window))
entry.pack()
window.mainloop()
```

5.1 Module

5.1.1 Colors

class ttkbootstrap.**Colors**(*primary*, *secondary*, *success*, *info*, *warning*, *danger*, *bg*, *fg*, *selectbg*, *selectfg*, *border*, *inputfg*, *inputbg*)

Bases: `object`

A class that contains the theme colors as well as several helper methods for manipulating colors.

This class is attached to the `Style` object at run-time for the selected theme, and so is available to use with `Style.colors`. The colors can be accessed via dot notation or get method:

```
# dot-notation
Colors.primary

# get method
Colors.get('primary')
```

This class is an iterator, so you can iterate over the main style color labels (primary, secondary, success, info, warning, danger):

```
for color_label in Colors:
    color = Colors.get(color_label)
    print(color_label, color)
```

If, for some reason, you need to iterate over all theme color labels, then you can use the `Colors.label_iter` method. This will include all theme colors, including border, fg, bg, etc...

```
for color_label in Colors.label_iter():
    color = Colors.get(color_label)
    print(color_label, color)
```

Parameters

- **primary** (*str*) – the primary theme color; used by default for all widgets.
- **secondary** (*str*) – an accent color; commonly of a *grey* hue.
- **success** (*str*) – an accent color; commonly of a *green* hue.
- **info** (*str*) – an accent color; commonly of a *blue* hue.

- **warning** (*str*) – an accent color; commonly of an *orange* hue.
- **danger** (*str*) – an accent color; commonly of a *red* hue.
- **bg** (*str*) – background color.
- **fg** (*str*) – default text color.
- **selectfg** (*str*) – the color of selected text.
- **selectbg** (*str*) – the background color of selected text.
- **border** (*str*) – the color used for widget borders.
- **inputfg** (*str*) – the text color for input widgets: ie. `Entry`, `Combobox`, etc...
- **inputbg** (*str*) – the text background color for input widgets.

get(*color_label*)

Lookup a color property

Parameters **color_label** (*str*) – a color label corresponding to a class propery (primary, secondary, success, etc...)

ondary, success, etc...)

Returns a hexadecimal color value.

Return type str

static hex_to_rgb(*color*)

Convert hexadecimal color to rgb color value

Parameters **color** (*str*) – param str color: hexadecimal color value

Returns rgb color value.

Return type tuple[int, int, int]

static label_iter()

Iterate over all color label properties in the Color class

Returns an iterator representing the name of the color properties

Return type iter

static rgb_to_hex(*r*, *g*, *b*)

Convert rgb to hexadecimal color value

Parameters

- **r** (*int*) – red
- **g** (*int*) – green
- **b** (*int*) – blue

Returns a hexadecimal colorl value

Return type str

set(*color_label*, *color_value*)

Set a color property

Parameters

- **color_label** (*str*) – the name of the color to be set (key)
- **color_value** (*str*) – a hexadecimal color value

Example

static update_hsv(*color*, *hd=0*, *sd=0*, *vd=0*)

Modify the hue, saturation, and/or value of a given hex color value.

Parameters

- **color** (*str*) – the hexadecimal color value that is the target of hsv changes.
- **hd** (*float*) – % change in hue
- **sd** (*float*) – % change in saturation
- **vd** (*float*) – % change in value

Returns a new hexadecimal color value that results from the hsv arguments passed into the function

Return type str

4.1 Dials & Meters

This example demonstrates the versitility of the `Meter` widget. All of the example below were created using the same class. All of the examples below include a supplemental label using the `labeltext` parameter, and all but the first example use the `textappend` parameter to add the 'gb', '%', and degrees symbol. Finally, all of the examples use the parameter `interactive=True` which turns the meter into a dial that can be manipulated directly with a mouse-click or drag. The theme used for the examples below is *cosmo*.

top-left the `metertype` is *semi* which gives the meter a semi-circle arc. The `meterstyle` is *primary.TLabel*.

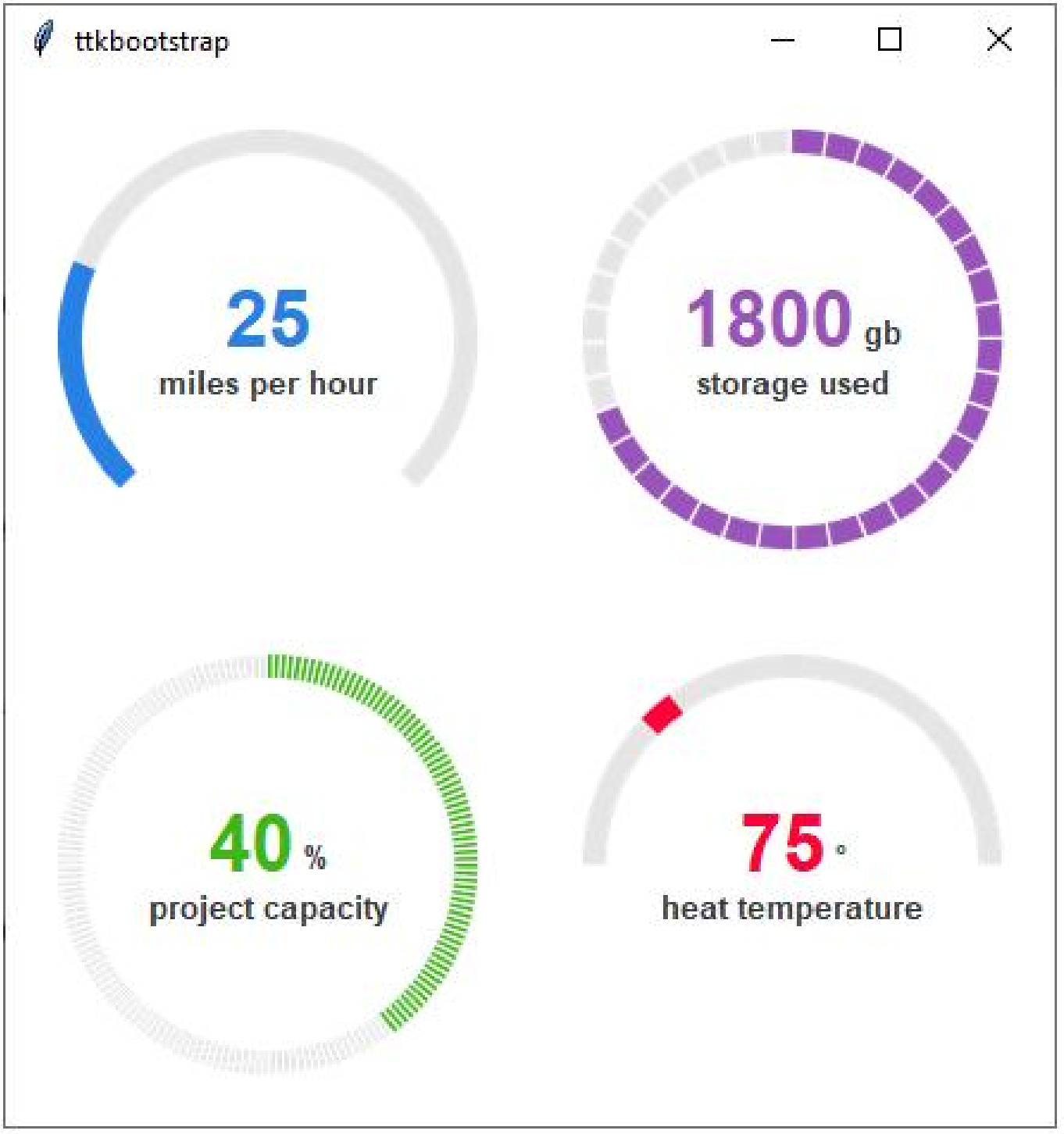
top-right the `stripethickness` is *10* pixels to give it a segmented appearance. The `meterstyle` is *info.TLabel*.

bottom-left the `stripethickness` is *2* pixels to give it a very thin segmented appearance. The `meterstyle` is *success.TLabel*.

bottom-right this example has a custom arc, with the `arcrange` at *180*, the `arcoffset` at *-180* and the `wedgethickness` at *5* pixels in order to create a wedge style indicator that rests at the meter value. The `meterstyle` is *danger.TLabel*.

```
"""
    Author: Israel Dryer
    Modified: 2021-05-09
"""

from ttkbootstrap import Style
from ttkbootstrap.widgets import Meter

style = Style('cosmo')
root = style.master
root.title('ttkbootstrap')

m1 = Meter(metersize=180, padding=20, amountused=25, metertype='semi', labeltext='miles␣
↪per hour', interactive=True)
m1.grid(row=0, column=0)

m2 = Meter(metersize=180, padding=20, amountused=1800, amounttotal=2600, labeltext=
↪'storage used', textappend='gb',
```

(continues on next page)


ttkbootstrap
25
miles per hour
1800 gb
storage used
40 %
project capacity
75 °
heat temperature

(continued from previous page)

```
            meterstyle='info.TMeter', stripethickness=10, interactive=True)
m2.grid(row=0, column=1)

m3 = Meter(metersize=180, padding=20, stripethickness=2, amountused=40, labeltext=
↪'project capacity', textappend='%',
            meterstyle='success.TMeter', interactive=True)
m3.grid(row=1, column=0)

m4 = Meter(metersize=180, padding=20, amounttotal=280, arcrange=180, arcoffset=-180,␣
↪amountused=75, textappend='°',
            labeltext='heat temperature', wedgesize=5, meterstyle='danger.TMeter',␣
↪interactive=True)
m4.grid(row=1, column=1)

root.mainloop()
```

54.4. Event modifiers

The modifier names that you can use in event sequences include:

`Alt`	True when the user is holding the *alt* key down.
`Any`	This modifier generalizes an event type. For example, the event pattern `'<Any-KeyPress>'` applies to the pressing of any key.
`Control`	True when the user is holding the *control* key down.
`Double`	Specifies two events happening close together in time. For example, `<Double-Button-1>` describes two presses of button 1 in rapid succession.
`Lock`	True when the user has pressed *shift lock*.
`Shift`	True when the user is holding down the *shift* key.
`Triple`	Like `Double`, but specifies three events in rapid succession.

You can use shorter forms of the events. Here are some examples:

- `'<1>'` is the same as `'<Button-1>'`.
- `'x'` is the same as `'<KeyPress-x>'`.

Note that you can leave out the enclosing `'<...>'` for most single-character keypresses, but you can't do that for the space character (whose name is `'<space>'`) or the less-than (`<`) character (whose name is `'<less>'`).

`<Button-1>`	The user pressed the first mouse button.
`<KeyPress-H>`	The user pressed the H key.
`<Control-Shift-KeyPress-H>`	The user pressed *control-shift*-H.

54.3. Event types

The full set of event types is rather large, but a lot of them are not commonly used. Here are most of the ones you'll need:

Type	Name	Description
36	`Activate`	A widget is changing from being inactive to being active. This refers to changes in the `state` option of a widget such as a button changing from inactive (grayed out) to active.

Type	Name	Description
4	`Button`	The user pressed one of the mouse buttons. The `detail` part specifies which button. For mouse wheel support under Linux, use `Button-4` (scroll up) and `Button-5` (scroll down). Under Linux, your handler for mouse wheel bindings will distinguish between scroll-up and scroll-down by examining the `.num` field of the `Event` instance; see Section 54.6, "Writing your handler: The `Event` class" (p. 162).
5	`ButtonRelease`	The user let up on a mouse button. This is probably a better choice in most cases than the `Button` event, because if the user accidentally presses the button, they can move the mouse off the widget to avoid setting off the event.
22	`Configure`	The user changed the size of a widget, for example by dragging a corner or side of the window

37	`Deactivate`	A widget is changing from being active to being inactive. This refers to changes in the `state` option of a widget such as a radiobutton changing from active to inactive (grayed out).
17	`Destroy`	A widget is being destroyed.
7	`Enter`	The user moved the mouse pointer into a visible part of a widget. (This is different than the *enter* key, which is a `KeyPress` event for a key whose name is actually `'return'`.)
12	`Expose`	This event occurs whenever at least some part of your application or widget becomes visible after having been covered up by another window.
9	`FocusIn`	A widget got the input focus (see Section 53, "Focus: routing keyboard input" (p. 155) for a general introduction to input focus.) This can happen either in response to a user event (like using the *tab* key to move focus between widgets) or programmatically (for example, your program calls the `.focus_set()` on a widget).
10	`FocusOut`	The input focus was moved out of a widget. As with `FocusIn`, the user can cause this event, or your program can cause it.
2	`KeyPress`	The user pressed a key on the keyboard. The `detail` part specifies which key. This keyword may be abbreviated `Key`.
3	`KeyRelease`	The user let up on a key.
8	`Leave`	The user moved the mouse pointer out of a widget.
19	`Map`	A widget is being mapped, that is, made visible in the application. This will happen, for example, when you call the widget's `.grid()` method.
6	`Motion`	The user moved the mouse pointer entirely within a widget.
38	`MouseWheel`	The user moved the mouse wheel up or down. At present, this binding works on Windows and MacOS, but not under Linux. For Windows and MacOS, see the discussion of the `.delta` field of the `Event` instance in Section 54.6, "Writing your handler: The `Event` class" (p. 162). For Linux, see the note above under `Button`.
18	`Unmap`	A widget is being unmapped and is no longer visible. This happens, for example, when you use the widget's `.grid_remove()` method.
15	`Visibility`	Happens when at least some part of the application window becomes visible on the screen.

55.2. The `tkFileDialog` module

The `tkFileDialog` module provides two different pop-up windows you can use to give the user the ability to find existing files or create new files.

`.askopenfilename(`*`option`*`=`*`value`*`, ...)`
Intended for cases where the user wants to select an existing file. If the user selects a nonexistent file, a popup will appear informing them that the selected file does not exist.

`.asksaveasfilename(`*`option`*`=`*`value`*`, ...)`
Intended for cases where the user wants to create a new file or replace an existing file. If the user selects an existing file, a pop-up will appear informing that the file already exists, and asking if they really want to replace it.

The arguments to both functions are the same:

`defaultextension=`*`s`*
The default file extension, a string starting with a period (`'.'`). If the user's reply contains a period, this argument has no effect. It is appended to the user's reply in case there are no periods.

For example, if you supply a `defaultextension='.jpg'` argument and the user enters `'gojiro'`, the returned file name will be `'gojiro.jpg'`.

`filetypes=[(`*`label`*$_1$`, `*`pattern`*$_1$`), (`*`label`*$_2$`, `*`pattern`*$_2$`), ...]`
A list of two-element tuples containing file type names and patterns that will select what appears in the file listing. In the screen picture below, note the pull-down menu labeled "Files of type:". The `filetypes` argument you supply will populate this pull-down list. Each *`pattern`* is a file type name ("PNG" in the example) and a pattern that selects files of a given type ("(*.png)" in the example).

`initialdir=`*`D`*
The path name of the directory to be displayed initially. The default directory is the current working directory.

`initialfile=`*`F`*
The file name to be displayed initially in the "File name:" field, if any.

parent=*W*

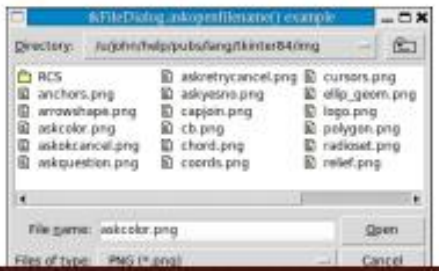
To make the pop-up appear over some window *W*, supply this argument. The default behavior is that the pop-up will appear over your application's root window.

title=*T*

If specified, *T* is a string to be displayed as the pop-up window's title.

If the user selects a file, the returned value is the complete path name of the selected file. If the user uses the *Cancel* button, the function returns an empty string.

Here is an example:



5.2.3 Floodgauge

class ttkbootstrap.widgets.**Floodgauge**(*master=None*, *cursor=None*, *font=None*, *length=None*, *maximum=100*, *mode='determinate'*, *orient='horizontal'*, *style='TFloodgauge'*, *takefocus=False*, *text=None*, *value=0*, ***kw*)

Bases: tkinter.ttk.Progressbar

A Floodgauge widget shows the status of a long-running operation with an optional text indicator.

Similar to the ttk.Progressbar, this widget can operate in two modes: **determinate** mode shows the amount completed relative to the total amount of work to be done, and **indeterminate** mode provides an animated display to let the user know that something is happening.

Variable are generated automatically for this widget and can be linked to other widgets by referencing them via the textvariable and variable attributes.

The text and value properties allow you to easily get and set the value of these variables without the need to call the get and set methods of the related tkinter variables. For example: Floodgauge.value or Floodgauge.value = 55 will get or set the amount used on the widget.

Parameters

- **master** (*Widget*) – Parent widget
- **cursor** (*str*) – The cursor that will appear when the mouse is over the progress bar.
- **font** (*Font or str*) – The font to use for the progress bar label.
- **length** (*int*) – Specifies the length of the long axis of the progress bar (width if horizontal, height if vertical); defaults to 300.
- **maximum** (*float*) – A floating point number specifying the maximum value. Defaults to 100.

100.

- **mode** (*str*) – One of **determinate** or **indeterminate**. Use *indeterminate* if you cannot accurately measure the relative progress of the underlying process. In this mode, a rectangle bounces back and forth between the ends of the widget once you use the `.start()` method. Otherwise, use *determinate* if the relative progress can be calculated in advance. This is the default mode.
- **orient** (*str*) – Specifies the orientation of the widget; either *horizontal* or *vertical*.
- **style** (*str*) – The style used to render the widget; *TFloodgauge* by default.
- **takefocus** (*bool*) – This widget is not included in focus traversal by default. To add the widget to focus traversal, use `takefocus=True`.
- **text** (*str*) – A string of text to be displayed in the progress bar. This is assigned to the `textvariable` `StringVar` which is automatically generated on instantiation. This value can be get and set using the `Floodgauge.text` property without having to directly call the `textvariable`.
- **value** – The current value of the progressbar. In *determinate* mode, this represents the amount of work completed. In *indeterminate* mode, it is interpreted modulo `maximum`; that is, the progress bar completes one "cycle" when the `value` increases by `maximum`.
- ****kw** – Other configuration options from the option database.

ttkbootstrap

start(*interval=None*)
Begin autoincrement mode: schedules a recurring timer event that calls method step every interval milliseconds.

interval defaults to 50 milliseconds (20 steps/second) if omitted.

step(*amount=None*)
Increments the value option by amount.

amount defaults to 1.0 if omitted.

stop()
Stop autoincrement mode: cancels any recurring timer event initiated by start.

2.4.6.3 Configuration

Use the following classes, states, and options when configuring or modifying a new ttk floodgauge style. See the python style documentation for more information on creating a style.

Create a new theme using TTK Creator if you want to change the default color scheme.

Class names

- Horizontal.TFloodgauge
- Vertical.TFloodgauge

Style options

background *color*

barsize *amount*

bordercolor *color*

borderwidth *amount*

darkcolor *color*

lightcolor *color*

pbarrelief *flat, groove, raised, ridge, solid, sunken*

thickness *amount*

troughcolor *color*

troughrelief *flat, groove, raised, ridge, solid, sunken*

5.2.4 Meter

class ttkbootstrap.widgets.**Meter**(*master=None, arcrange=None, arcoffset=None, amounttotal=100, amountused=0, interactive=False, labelfont='Helvetica 10 bold', labelstyle='secondary.TLabel', labeltext=None, metersize=200, meterstyle='TMeter', metertype='full', meterthickness=10, showvalue=True, stripethickness=0, textappend=None, textfont='Helvetica 25 bold', textprepend=None, wedgesize=0, **kw*)

Bases: `tkinter.ttk.Frame`

A radial meter that can be used to show progress of long running operations or the amount of work completed; can also be used as a *Dial* when set to `interactive=True`.

This widget is very flexible. There are two primary meter types which can be set with the `metertype` parameter: 'full' and 'semi', which show the arc of the meter in a full or semi-circle. You can also customize the arc of the circle with the `arcrange` and `arcoffset` parameters.

The progress bar indicator can be displayed as a solid color or with stripes using the `stripethickness` parameter. By default, the `stripethickness` is 0, which results in a solid progress bar. A higher `stripethickness` results in larger wedges around the arc of the meter.

Various text and label options exist. The center text and progressbar is formatted with the `meterstyle` parameter and uses the *TMeter* styles. You can prepend or append text to the center text using the `textappend` and `textprepend` parameters. This is most commonly used for '$', '%', or other such symbols.

Variable are generated automatically for this widget and can be linked to other widgets by referencing them via the `amountusedvariable` and `amounttotalvariable` attributes.

The variable properties allow you to easily get and set the value of these variables. For example: `Meter.amountused` or `Meter.amountused = 55` will get or set the amount used on the widget without having to call the `get` or `set` methods of the tkinter variable.

Parameters

- **master** (*Widget*) – Parent widget
- **arcoffset** (*int*) – The amount to offset the arc's starting position in degrees; 0 is at 3 o'clock.
- **arcrange** (*int*) – The range of the arc in degrees from start to end.
- **amounttotal** (*int*) – The maximum value of the meter.
- **amountused** (*int*) – The current value of the meter; displayed if `showvalue=True`.

- **interactive** (*bool*) – Enables the meter to be adjusted with mouse interaction.
- **labelfont** (*Font or str*) – The font of the supplemental label.

- **labelstyle** (*str*) – The ttk style used to render the supplemental label.
- **labeltext** (*str*) – Supplemental label text that appears *below* the center text.
- **metersize** (*int*) – The size of the meter; represented by one side length of a square.
- **meterstyle** (*str*) – The ttk style used to render the meter and center text.
- **metertype** (*str*) – One of **full** or **semi**; displays a full-circle or semi-circle.
- **meterthickness** (*int*) – The thickness of the meter's progress bar.
- **showvalue** (*bool*) – Show the meter's value in the center text; default = True.
- **stripethickness** (*int*) – The meter's progress bar can be displayed in solid or striped form. If the value is greater than 0, the meter's progress bar changes from a solid to striped, where the value is the thickness of the stripes.
- **textappend** (*str*) – A short string appended to the center text.
- **textfont** (*Font or str*) – The font of the center text.
- **textprepend** (*str*) – A short string prepended to the center text.
- **wedgesize** (*int*) – If greater than zero, the width of the wedge on either side of the current meter value.

convert_system_color(*systemcolorname*)
Convert a system color name to a hexadecimal value

Parameters **systemcolorname** (*str*) – a system color name, such as *SystemButtonFace*

draw_base_image()
Draw the base image to be used for subsequent updates

draw_meter(**args*)
Draw a meter

Parameters ***args** – if triggered by a trace, will be *variable*, *index*, *mode*.

draw_solid_meter(*draw*)
Draw a solid meter

Parameters **draw** (*ImageDraw.Draw*) – an object used to draw an arc on the meter

draw_striped_meter(*draw*)
Draw a striped meter

Parameters **draw** (*ImageDraw.Draw*) – an object used to draw an arc on the meter

lookup(*style*, *option*)
Wrapper around the tcl style lookup command

Parameters

- **style** (*str*) – the name of the style used for rendering the widget.
- **option** (*str*) – the option to lookup from the style option database.

Returns the value of the option looked up.

Return type any

meter_value()
Calculate the meter value

Returns the value to be used to draw the arc length of the progress meter

Return type int

`on_dial_interact`(*e*)

Callback for mouse drag motion on indicator

Parameters **e** (*Event*) – event callback for drag motion.

`step`(*delta=1*)

Increase the indicator value by `delta`.

The default increment is 1. The indicator will reverse direction and count down once it reaches the maximum value.

Keyword Arguments **delta** (*int*) – the amount to change the indicator.

2.4.11.2 How to use

The examples below demonstrate how to *use a style* when creating a meter widget.

Create a default **meter**

```
Meter(parent, amountused=25, labeltext='miles per hour')
```

Create a **danger meter**

```
Meter(parent, amountused=25, labeltext='miles per hour', meterstyle='danger.TLabel')
```

Create an **info meter** with an **success label**

```
Meter(parent, amountused=25, labeltext='miles per hour', meterstyle='info.TLabel',␣
↪labelstyle='success.TLabel')
```

5.1.2 Style

class ttkbootstrap.**Style**(*theme='flatly', themes_file=None, *args, **kwargs*)

Bases: tkinter.ttk.Style

A class for setting the application style.

Sets the theme of the tkinter.Tk instance and supports all ttkbootstrap and ttk themes provided. This class is meant to be a drop-in replacement for ttk.Style and inherits all of it's methods and properties. Creating a Style object will instantiate the tkinter.Tk instance in the Style.master property, and so it is not necessary to explicitly create an instance of tkinter.Tk. For more details on the ttk.Style class, see the python documentation.

```
# instantiate the style with default theme *flatly*
style = Style()

# instantiate the style with another theme
style = Style(theme='superhero')

# instantiate the style with a theme from a specific themes file
style = Style(theme='custom_name', themes_file='C:/example/my_themes.json')

# available themes
for theme in style.theme_names():
    print(theme)
```

Parameters

- **theme** (*str*) – the name of the theme to use at runtime; *flatly* by default.
- **themes_file** (*str*) – Path to a user-defined themes file. Defaults to the themes file set in ttkcreator.

configure(*style*, *query_opt=None*, ***kw*)
Query or sets the default value of the specified option(s) in style.

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

:bootstrap

element_create(*elementname*, *etype*, **args*, ***kw*)
Create a new element in the current theme of given etype.

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

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

layout(*style*, *layoutspec=None*)

Define the widget layout for given style. If layoutspec is omitted, return the layout specification for given style.

layoutspec is expected to be a list or an object different than None that evaluates to False if you want to "turn off" that style. If it is a list (or tuple, or something else), each item should be a tuple where the first item is the layout name and the second item should have the format described below:

LAYOUTS

A layout can contain the value None, if 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.

children: [sublayout...] Specifies a list of elements to place inside the element. Each element is a tuple (or other sequence) where the first item is the layout name, and the other is a LAYOUT.

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.

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, or list, or something else of your preference. A statespec is compound of one or more states and then a value.

register_theme(*definition*)

Registers a theme definition for use by the Style class.

This makes the definition and name available at run-time so that the assets and styles can be created.

Parameters **definition** (ThemeDefinition) – an instance of the ThemeDefinition class

theme_create(*themename*, *parent=None*, *settings=None*)

Creates 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 specified parent theme. If settings are present, they are expected to have the same syntax used for theme_settings.

theme_names()

Returns a list of all known themes.

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 configure, map, layout and element_create respectively.

theme_use(*themename=None*)

Changes the theme used in rendering the application widgets.

If themename is None, returns the theme in use, otherwise, set the current theme to themename, refreshes all widgets and emits a <<ThemeChanged>> event.

Only use this method if you are changing the theme *during* runtime. Otherwise, pass the theme name into the Style constructor to instantiate the style with a theme.

Keyword Arguments **themename** (`str`) – the theme to apply when creating new widgets

5.1.4 StylerTK

class ttkbootstrap.**StylerTK**(*styler_ttk*)

Bases: object

A class for styling tkinter widgets (not ttk).

Several ttk widgets utilize tkinter widgets in some capacity, such as the *popdownlist* on the ttk.Combobox. To create a consistent user experience, standard tkinter widgets are themed as much as possible with the look and feel of the **ttkbootstrap** theme applied. Tkinter widgets are not the primary target of this project; however, they can be used without looking entirely out-of-place in most cases.

master

the root window.

Type Tk

theme

the color settings defined in the *themes.json* file.

Type *ThemeDefinition*

Parameters **styler_ttk** (StylerTTK) – an instance of the StylerTTK class.

style_tkinter_widgets()

A wrapper on all widget style methods. Applies current theme to all standard tkinter widgets

Example: DebugWindow

```
# -*- coding: utf-8 -*-

# Copyright (c) RedFantom 2017
# Copyright (c) Juliette Monsel 2017
# For license see LICENSE

from ttkwidgets import DebugWindow
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
ttk.Button(root, text="Print ok", command=lambda: print('ok')).pack()
DebugWindow(root)
root.mainloop()
```

Example: askcolor

```
# -*- coding: utf-8 -*-

# Copyright (c) Juliette Monsel 2018
# For license see LICENSE

from ttkwidgets.color import askcolor
import tkinter as tk
from tkinter import ttk
from PIL import Image, ImageTk


def pick(alpha=False):
    global im  # to avoid garbage collection of image
    res = askcolor('sky blue', parent=window, title='Pick a color', alpha=alpha)
    canvas.delete('image')
    if res[1] is not None:
        im = ImageTk.PhotoImage(Image.new('RGBA', (100, 100), res[1]), master=window)
        canvas.create_image(60, 60, image=im, tags='image', anchor='center')
    print(res)


window = tk.Tk()
canvas = tk.Canvas(window, width=120, height=120)
canvas.create_text(60, 60, text='Background', anchor='center')
canvas.pack()
ttk.Button(window, text="Pick a color (No alpha channel)", command=pick).pack(fill='x
↪')
ttk.Button(window, text="Pick a color (With alpha channel)", command=lambda:␣
↪pick(True)).pack(fill='x')
window.mainloop()
```

Example: askfont

```
# -*- coding: utf-8 -*-

# Copyright (c) Juliette Monsel 2018
# For license see LICENSE

from ttkwidgets.font import askfont
import tkinter as tk
from tkinter import ttk

def font():
    res = askfont()
    if res[0] is not None:
        label.configure(font=res[0])
    print(res)

window = tk.Tk()
label = ttk.Label(window, text='Sample text rendered in the chosen font.')
label.pack(padx=10, pady=10)
ttk.Button(window, text="Pick a font", command=font).pack()
window.mainloop()
```

Example: FontSelectFrame

```
# -*- coding: utf-8 -*-

# Copyright (c) Juliette Monsel 2018
# For license see LICENSE

from ttkwidgets.font import FontSelectFrame
import tkinter as tk
from tkinter import ttk

def update_preview(font_tuple):
    print(font_tuple)
    font = font_selection.font[0]
    if font is not None:
        label.configure(font=font)

window = tk.Tk()
label = ttk.Label(window, text='Sample text rendered in the chosen font.')
label.pack(padx=10, pady=10)
```

(continues on next page)

(continued from previous page)

```
font_selection = FontSelectFrame(window, callback=update_preview)
font_selection.pack()
window.mainloop()
```

2.4.7.3 Style configuration

Use the following classes, states, and options when configuring or modifying a new ttk button style. See the python style documentation for more information on creating a style.

Create a new theme using TTK Creator if you want to change the default color scheme.

Class names

- TFrame

Dynamic states

- disabled
- focus
- pressed
- readonly

Style options

background *color*

relief *flat, groove, raised, ridge, solid, sunken*

Example: Tooltip

```
# -*- coding: utf-8 -*-

# Copyright (c) RedFantom 2017
# For license see LICENSE
from ttkwidgets.frames import Tooltip
import tkinter as tk


window = tk.Tk()
button = tk.Button(window, text="Button", command=window.destroy)
button.pack()
balloon = Tooltip(button)
window.mainloop()
```

```
# -*- coding: utf-8 -*-

# Copyright (c) RedFantom 2021
# For license see LICENSE
from tkinter import ttk
import tkinter as tk
from ttkwidgets.hook import hook_ttk_widgets

if __name__ == '__main__':
    hook_ttk_widgets(lambda s, o, v: print(s, o, v), {"tooltip": "Default Value"})
    hook_ttk_widgets(lambda s, o, v: print(s, o, v), {"hello_world": "second_hook"})

    original_init = ttk.Button.__init__
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, *args, **kwargs):
        print("User custom hook")
        original_init(self, *args, **kwargs)

    ttk.Button.__init__ = __init__

    window = tk.Tk()
    button = ttk.Button(window, text="Destroy", command=window.destroy, tooltip=
↪"Destroys Window")
    button.pack()
    print([name for name in dir(button) if name.startswith("WidgetHook")])
    window.after(1000, lambda: button.configure(tooltip="Does not destroy window",␣
↪command=lambda: None))
    window.mainloop()
```

Example: LinkLabel

```
# -*- coding: utf-8 -*-

# Copyright (c) RedFantom 2017
# Copyright (c) Juliette Monsel 2018
# For license see LICENSE

from ttkwidgets import LinkLabel
```

(continues on next page)

(continued from previous page)

```
import tkinter as tk

window = tk.Tk()
LinkLabel(window, text="ttkwidgets repository",
          link="https://github.com/RedFantom/ttkwidgets",
          normal_color='royal blue',
          hover_color='blue',
          clicked_color='purple').pack()
window.mainloop()
```

5.1.5 ThemeDefinition

class ttkbootstrap.**ThemeDefinition**(*name='default'*, *themetype='light'*, *font='helvetica'*, *colors=None*)

Bases: object

A class to provide defined name, colors, and font settings for a ttkbootstrap theme.

Parameters

- **name** (str) – the name of the theme; default is 'default'.
- **themetype** (str) – the type of theme: *light* or *dark*; default is 'light'.
- **font** (str) – the default font to use for the application; default is 'helvetica'.
- **colors** (Colors) – an instance of the *Colors* class. One is provided by default.

Example: TickScale

```
# -*- coding: utf-8 -*-

# Copyright (c) Juliette Monsel 2017
# For license see LICENSE
from ttkwidgets import TickScale
import tkinter as tk
from tkinter import ttk


root = tk.Tk()
style = ttk.Style(root)
style.theme_use('clam')
style.configure('my.Vertical.TScale', sliderlength=50, background='white',
                foreground='red')
style.configure('my.Horizontal.TScale', sliderlength=10,
                font='TkDefaultFont 20 italic')
s1 = TickScale(root, orient='vertical', style='my.Vertical.TScale',
               tickinterval=0.2, from_=-1, to=1, showvalue=True, digits=2,
               length=400, labelpos='e')
s2 = TickScale(root, orient='horizontal', style='my.Horizontal.TScale',
               from_=0, to=10, tickinterval=2, resolution=1,
               showvalue=True, length=400)
s3 = TickScale(root, orient='horizontal', from_=0.25, to=1, tickinterval=0.1,
               resolution=0.1)

s1.pack(fill='y')
s2.pack(fill='x')
s3.pack(fill='x')

root.mainloop()
```

```
import tkinter as tk
from ttkwidgets import TimeLine
```

(continues on next page)

(continued from previous page)

```
window = tk.Tk()
timeline = TimeLine(
    window,
    categories={str(key): {"text": "Category {}".format(key)} for key in range(0, 5)},
    height=100, extend=True
)
menu = tk.Menu(window, tearoff=False)
menu.add_command(label="Some Action", command=lambda: print("Command Executed"))
timeline.tag_configure("1", right_callback=lambda *args: print(args), menu=menu,␣
↪foreground="green",
                       active_background="yellow", hover_border=2, move_
↪callback=lambda *args: print(args))
timeline.create_marker("1", 1.0, 2.0, background="white", text="Change Color", tags=(
↪"1",), iid="1")
timeline.create_marker("2", 2.0, 3.0, background="green", text="Change Category",␣
↪foreground="white", iid="2",
                       change_category=True)
timeline.create_marker("3", 1.0, 2.0, text="Show Menu", tags=("1",))
timeline.create_marker("4", 4.0, 5.0, text="Do nothing", move=False)
timeline.draw_timeline()
timeline.grid()
window.after(2500, lambda: timeline.configure(marker_background="cyan"))
window.after(5000, lambda: timeline.update_marker("1", background="red"))
window.after(5000, lambda: print(timeline.time))
window.mainloop()
```

55.3. The `tkColorChooser` module

To give your application's user a popup they can use to select a color, import the `tkColorChooser` module and call this function:

```
result = tkColorChooser.askcolor(color, option=value, ...)
```

Arguments are:

color
The initial color to be displayed. The default initial color is a light gray.

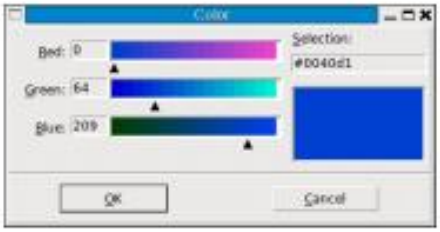
`title=`*`text`*
The specified *`text`* appears in the pop-up window's title area. The default title is "Color".

`parent=`*`W`*
Make the popup appear over window *`W`*. The default behavior is that it appears over your root window.

If the user clicks the *OK* button on the pop-up, the returned value will be a tuple `(`*`triple`*`, color)`, where *`triple`* is a tuple `(`*`R, G, B`*`)` containing red, green, and blue values in the range [0,255] respectively, and *`color`* is the selected color as a regular *Tkinter* color object.

If the users clicks *Cancel*, this function will return `(None, None)`.

Here's what the popup looks like on the author's system:



Example: ToggledFrame

```
# -*- coding: utf-8 -*-

# Copyright (c) RedFantom 2017
# For license see LICENSE

from ttkwidgets.frames import ToggledFrame
import tkinter as tk
from tkinter import ttk

window = tk.Tk()
frame = ToggledFrame(window, text="Value", width=10)
frame.pack()
button = ttk.Button(frame.interior, text="Button", command=window.destroy)
button.grid()
frame.toggle()
window.mainloop()
```

Example: tooltips

```
"""
Author: RedFantom
License: GNU GPLv3
Source: The ttkwidgets repository
"""
import tkinter as tk
from tkinter import ttk
# Import once, use everywhere
from ttkwidgets import tooltips


window = tk.Tk()
button = ttk.Button(window, text="Destroy", command=window.destroy, tooltip="This␣
↪button destroys the window.")
button.pack()
x = lambda: button.configure(tooltip="This button no longer destroys the window",␣
↪command=lambda: print("Behaviour changed!"))
window.after(5000, x)
window.mainloop()
```