ttkbootstrap

Israel Dryer

Dec 20, 2021

CONTENTS:

ttkbootstrap is a collection of modern, flat themes inspired by Bootstrap for tkinter/ttk. There are more than a dozen *built-in dark and light themes*. Even better, you can create your own with *TTK Creator*!

CHAPTER

ONE

INSTALLATION

Installing ttkbootstrap is easy! There are a few options for installing.

1.1 PyPI

Installing from PyPI is the easiest and recommended method. It will contain the most up-to-date *stable* distribution:

```
python -m pip install ttkbootstrap==0.5.3
```

This also installs `pillow` as a required dependency if it is not already installed. This library is used to handle some of the image processing used in ttkbootstrap.

Note: If you are on **Linux**, you may not have a font with emojii support. To prevent the program from crashing, I recommend you also install the *Symbola* font.

```
sudo apt-get install fonts-symbola
```

CHAPTER

TWO

HANDBOOK

2.1 Overview

2.1.1 Why does this project exist?

The purpose of this project is create a set of beautifully designed and easy to apply styles for your tkinter applications. Ttk can be very time-consuming to style if you are just a casual user. This project takes the pain out of getting a modern look and feel so that you can focus on designing your application. This project was created to harness the power of ttk's (and thus Python's) existing built-in theme engine to create modern and professional-looking user interfaces which are inspired by, and in many cases, whole-sale rip-off's of the themes found on Bootswatch. Even better, you have the abilty to *create and use your own custom themes* using TTK Creator.

2.1.2 A bootstrap approach to style

Many people are familiar with bootstrap for web developement. It comes pre-packaged with built-in css style classes that provide a professional and consistent api for quick development. I took a similar approach with this project by pre-defining styles for nearly all ttk widgets. This makes is very easy to apply the theme colors to various widgets by using style declarations. If you want a button in the *secondary* theme color, simply apply the **secondary.TButton** style to the button. Want a blue outlined button? Apply the **info.Outline.TButton** style to the button.

2.1.3 What about the old tkinter widgets?

Some of the ttk widgets utilize existing tkinter widgets. For example: there is a tkinter popdown list in the `ttk.Combobox` and a legacy tkinter widget inside the `ttk.OptionMenu`. To make sure these widgets didn't stick out like a sore thumb, I created a `StyleTK` class to apply the same color and style to these legacy widgets. While these legacy widgets are not necessarily intended to be used (and will probably not look as nice as the ttk versions when they exist), they are available if needed, and shouldn't look completely out-of-place in your ttkbootstrap themed application. *Check out this example* to see for yourself.

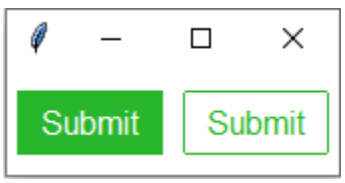
2.2 Tutorial

ttkbootstrap works by generating pre-defined themes at runtime which you can then apply and use as you would any built-in `ttk` theme such as **clam**, **alt**, **classic**, etc… You also have the ability to *create a new theme* using the **ttkcreator** application.

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')
```

Note: Check out the *visual style guide* for each widget. Here you will find an image for each available widget style, as well as information on how to apply and modify styles as needed for each widget.

2.2.2 Choose a theme

ttkbootstrap *light* and *dark* themes are generated at run-time. Generally, the `ttkbootstrap.Style` api is identical to the `ttk.Style` api. See the Python documentation on styling to learn more about this class.

To use a **ttkbootstrap** theme, create a `ttkbootstrap.Style` and pass in the name of the theme you want to use.

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

If you want to load a theme from a specific file (for example, to release an application with a custom theme), you can use the `user_themes` argument:

```
style = Style(theme='custom_name', themes_file='C:/example/my_themes.json')
```

If for some reason you need to change the theme *after* the window has already been created, you will need to use the `Style.theme_use` method, which is what `ttkbootstrap.Style` does internally when instantiated.

To get a list of all available themes:

```
style.theme_names()
```

Currently, the available pre-defined themes include:

light cosmo - flatly - journal - literal - lumen - minty - pulse - sandstone - united - yeti

dark cyborg - darkly - solar - superhero

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).

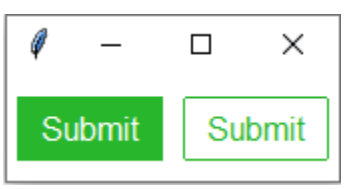
For example, the `ttk.Button` has a widget class of *TButton*. The style patterns available on the button include:

- 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)
```



Note: While all widgets are themed, not all have themed color styles available, such as `ttk.PanedWindow` or the `ttk.Scrollbar`. Instead, these widgets are styled with a default theme color.

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))
```

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):

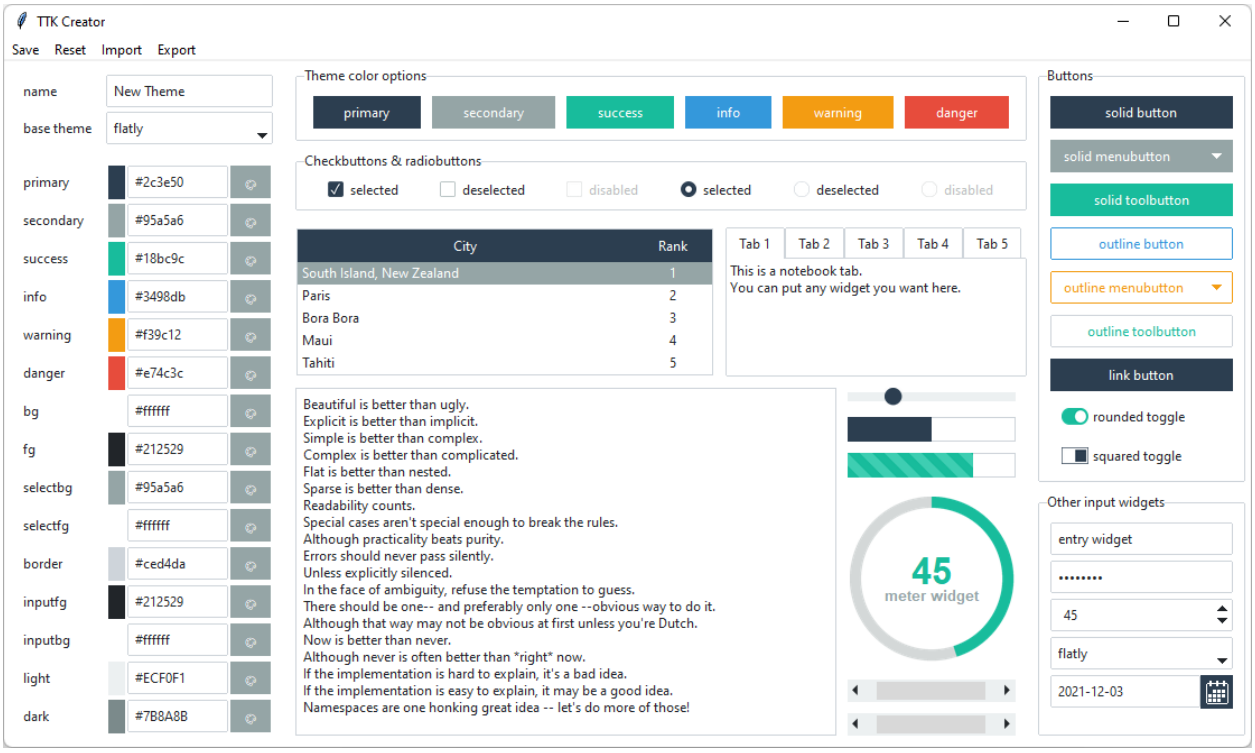
```
for color_label in Colors:
    color = Colors.get(color_label)
    print(color_label, color)
```

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)
```

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
```

2.2.6.2 Create and save your theme

- Name your theme
- Select a base theme to set the initial colors
- Click the color palette to select a color, or input a hex or named color directly
- Click **Save** on the menubar to save your theme
- Click **Reset** to apply the base theme defaults and start from scratch

Theme names must be unique. If you choose a theme name that already exists, you will be prompted to choose another.

You can check your new theme by starting up the **ttkbootstrap** demo application, which will load all available themes. Then, select your new theme from the option menu.

```
python -m ttkbootstrap
```

2.2.6.3 Export or import your user defined themes

- Click **Export** from the menubar to export user-defined themes into a `.py` file.
- Click **Import** to import user-defined themes.

IMPORTANT!!! Importing user-defined themes will overwrite any user-defined themes that are currently saved in the library. Additionally, upgrading to a new version of ttkbootstrap will erase user-defined themes.

2.3 Themes

All of the following themes are available with ttkbootstrap and can be viewed live with the ttkbootstrap demo:

```
python -m ttkbootstrap
```

2.3.1 Light themes

2.3.2 Dark themes

2.3.3 How are themes created?

Imagine being able to take the parts from several existing cars to design the one that you really want… that's basically how ttkbootstrap was created… I used the best parts of the existing themes to craft a brand new theme template.

The base of all widgets in the ttkbootstrap template is the *clam* theme. You may be wondering why the ttkbootstrap theme looks so different than the built-in clam theme… Each ttk widget is created from a collection of elements. These elements, when combined together, create what we see as a ttk widget. Aside from changing colors and state behavior, I constructed new widget layouts using the elements from various themes to give the desired look and feel. There is an old, but excellent reference to widget layouts here.

As an example: the `ttk.Combobox` widget contains a *field* element. In order to get the border effect I wanted, I constructed a new layout for the `ttk.Combobox` using the *field* from the `ttk.Spinbox`.

So, the `ttkbootstrap.StylerTTK` contains the style template for all ttkbootstrap themes. From there, a set of theme definitions (which includes color maps, default font, etc…) are extracted from a json file at runtime and loaded as a new theme by the `ttkbootstrap.Style` class.

```
{
  "name": "cosmo",
  "font": "Helvetica",
  "type": "light",
  "colors": {
                            "primary": "#2780e3",
                            "secondary": "#373a3c",
                            "success": "#3fb618",
                            "info": "#9954bb",
                            "warning": "#ff7518",
                            "danger": "#ff0039",
                            "light": "#f8f9fa",
                            "dark": "#373a3c",
                            "bg": "#ffffff",
                            "fg": "#373a3c",
                            "selectbg": "#373a3c",
```

(continues on next page)

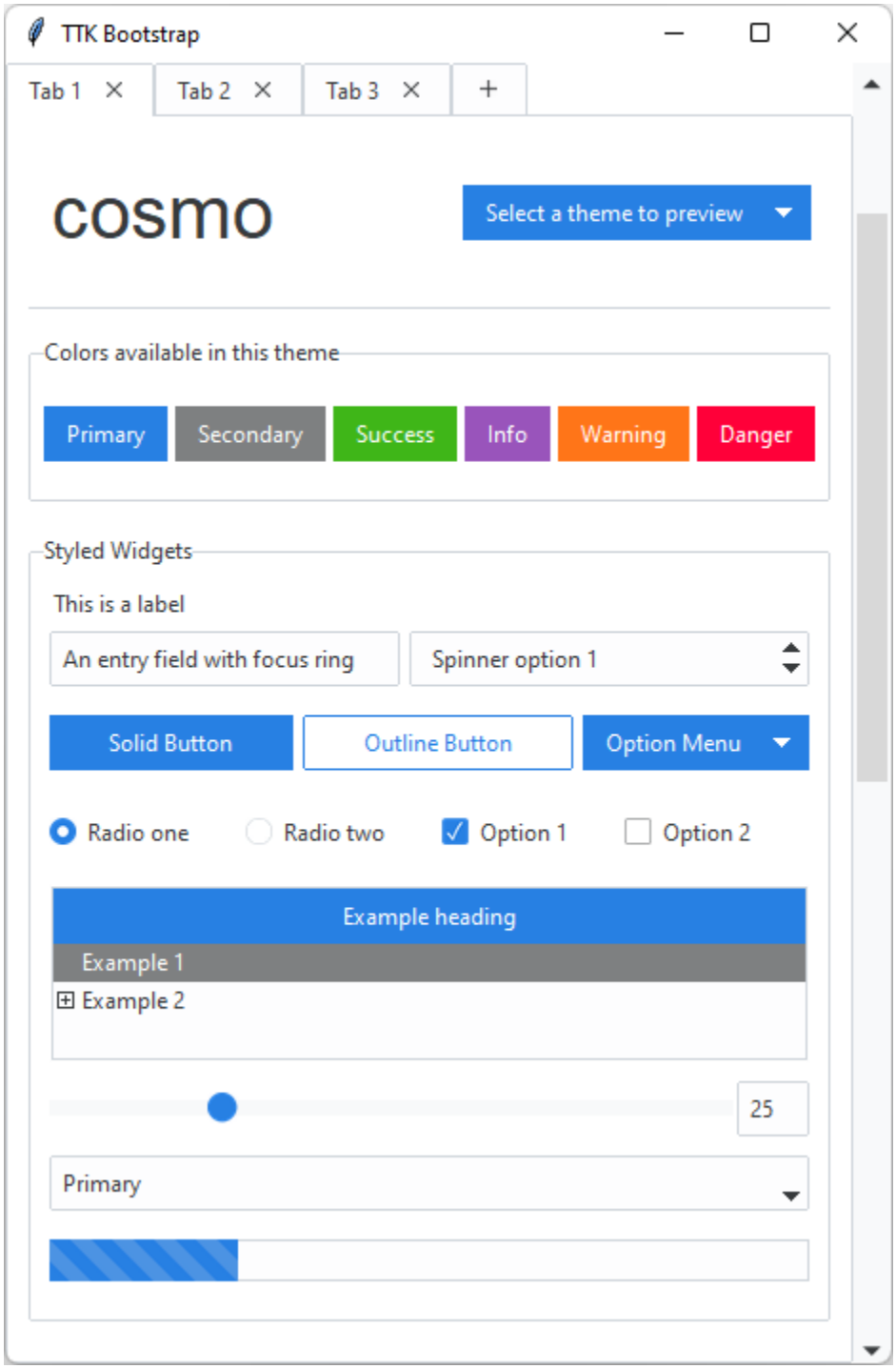


Fig. 1: inspired by https://bootswatch.com/cosmo/

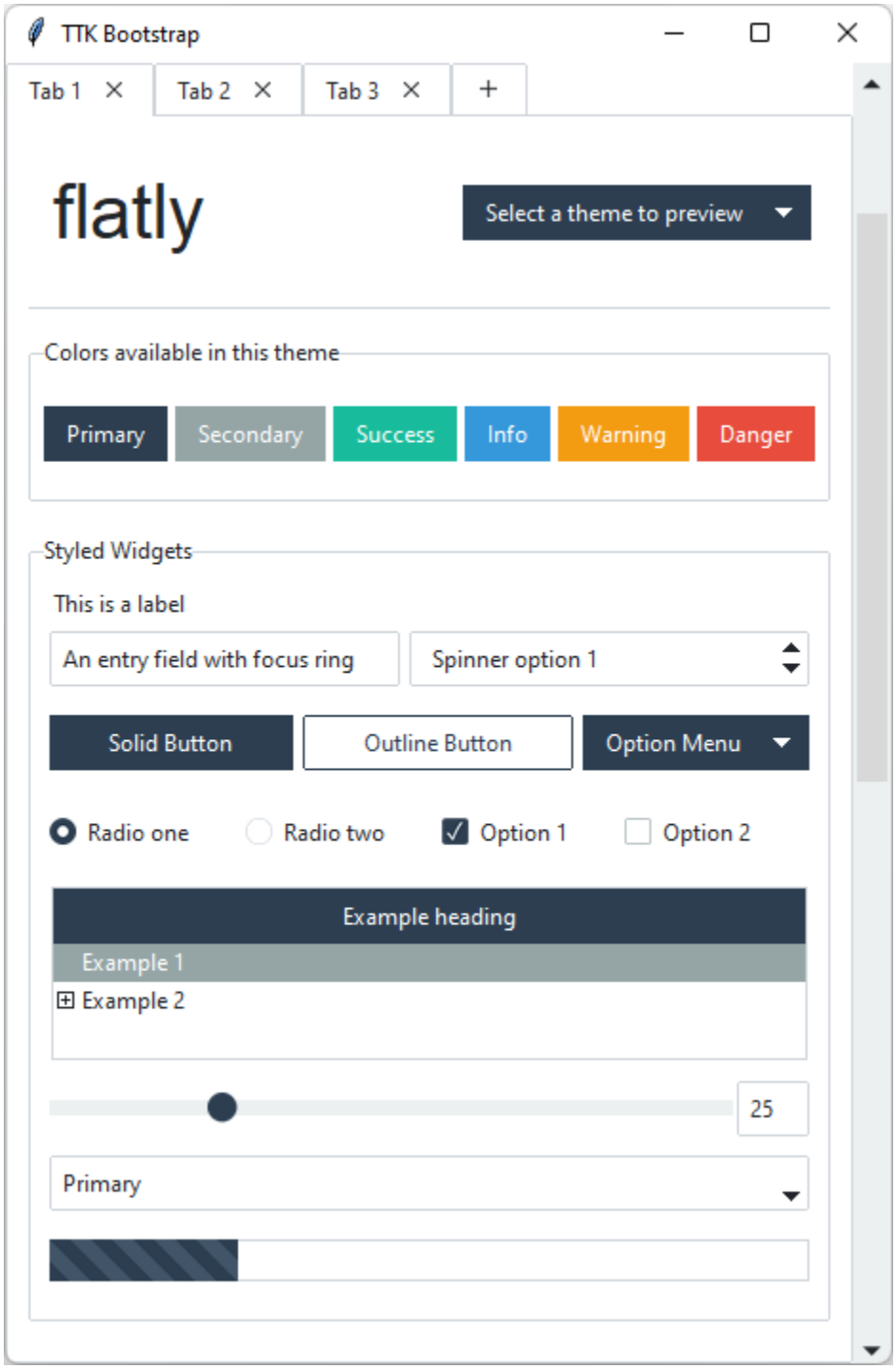


Fig. 2: inspired by https://bootswatch.com/flatly/

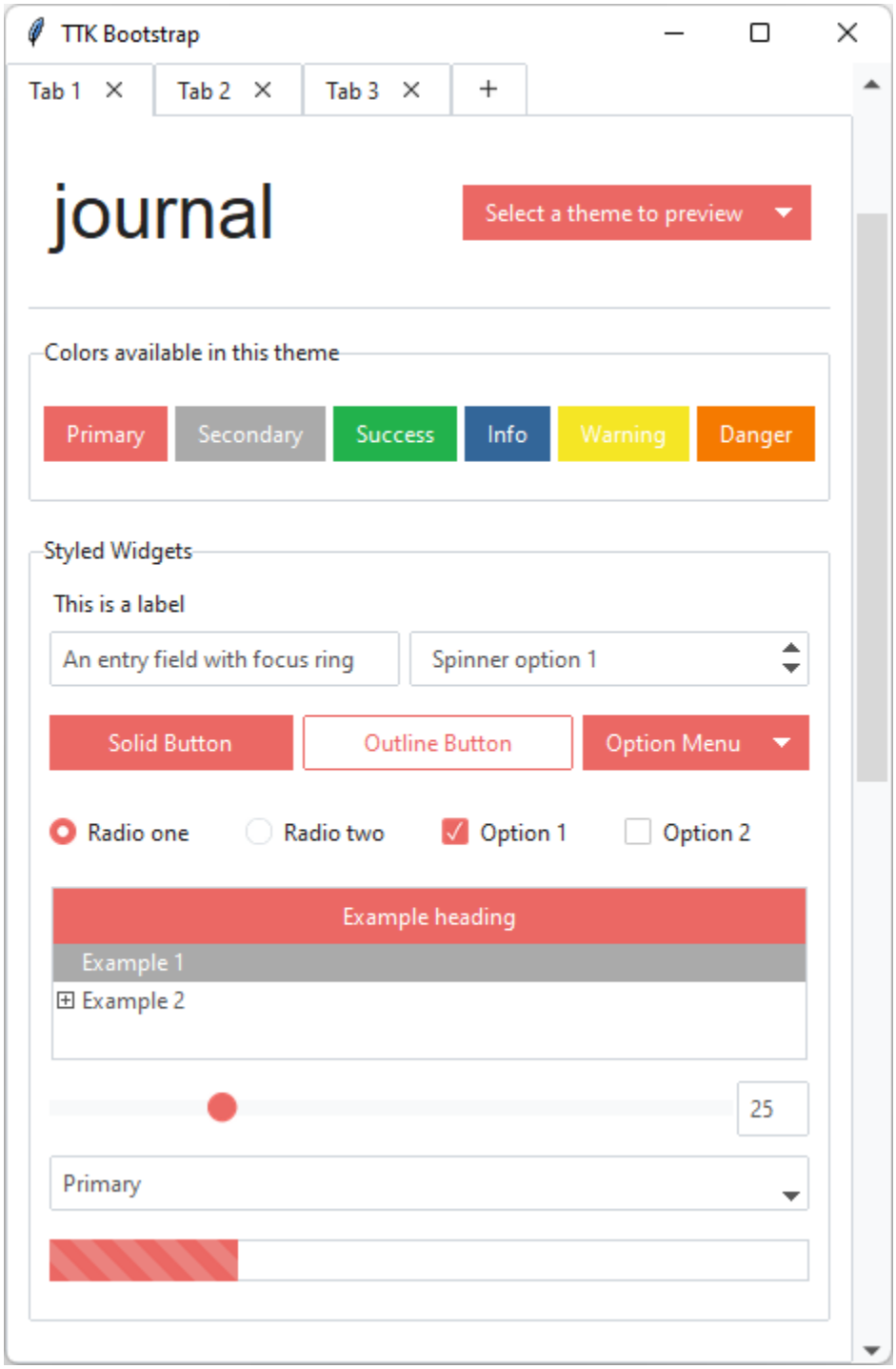


Fig. 3: inspired by https://bootswatch.com/journal/

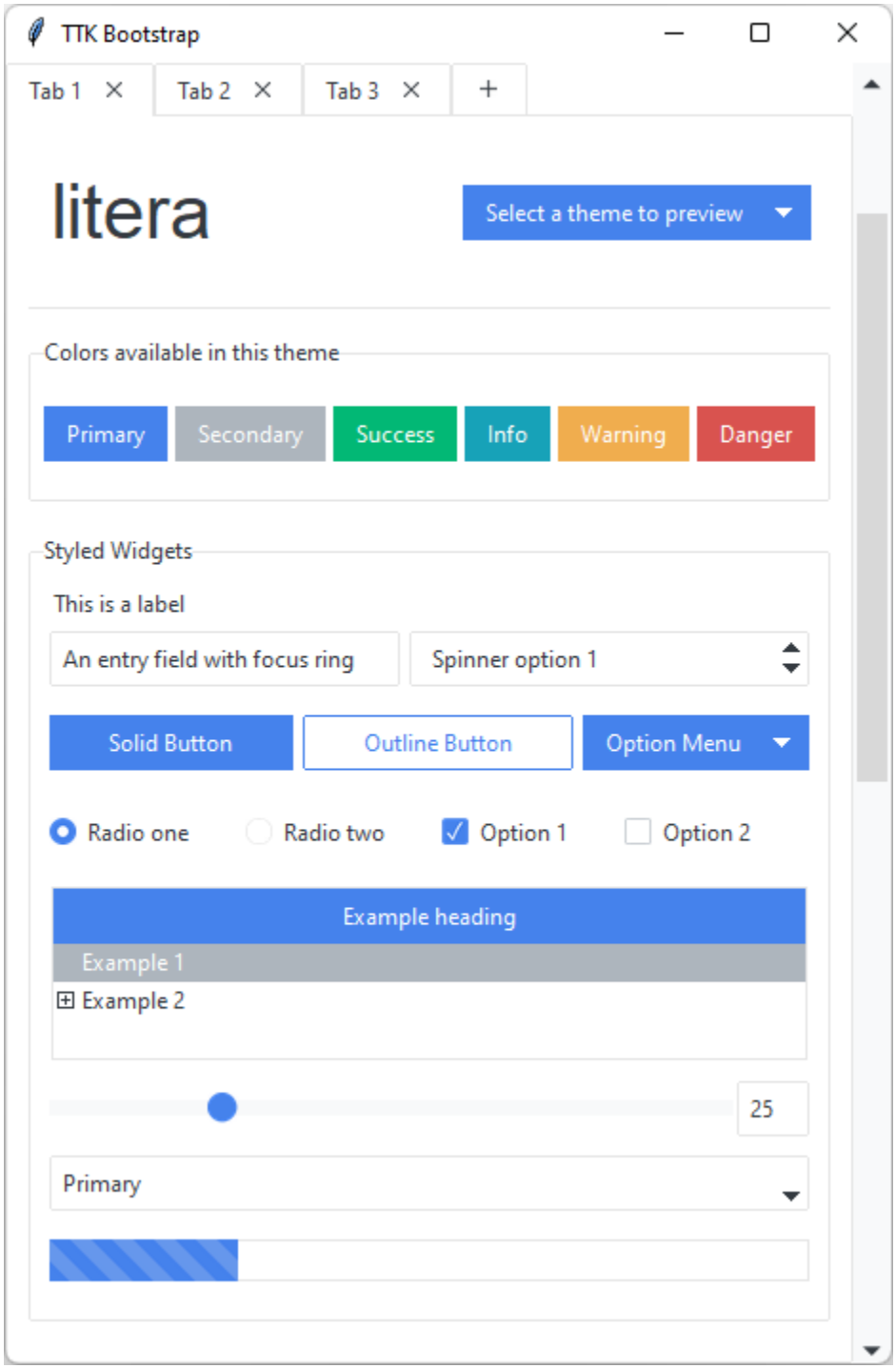


Fig. 4: inspired by https://bootswatch.com/litera/

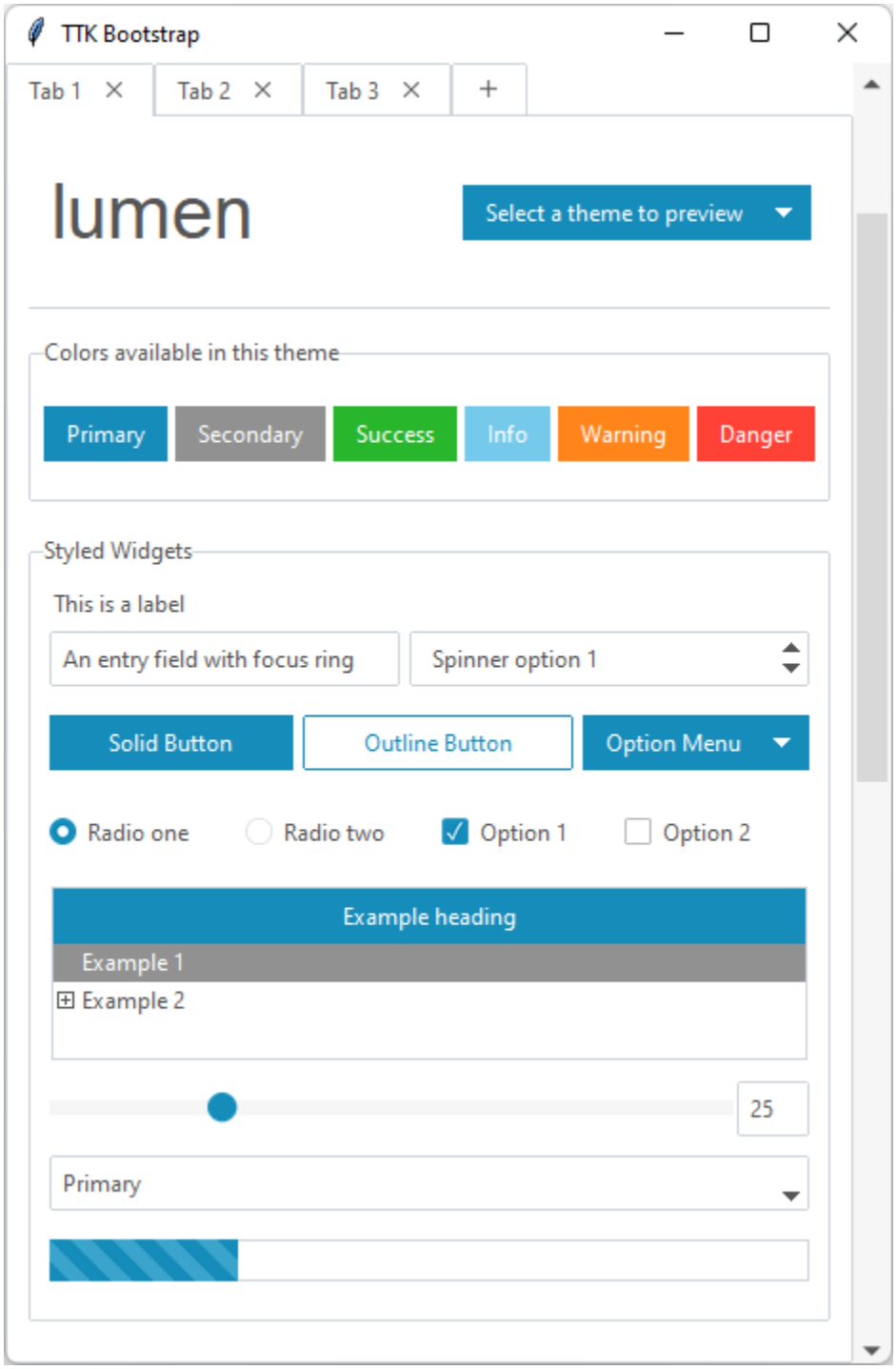


Fig. 5: inspired by https://bootswatch.com/lumen/

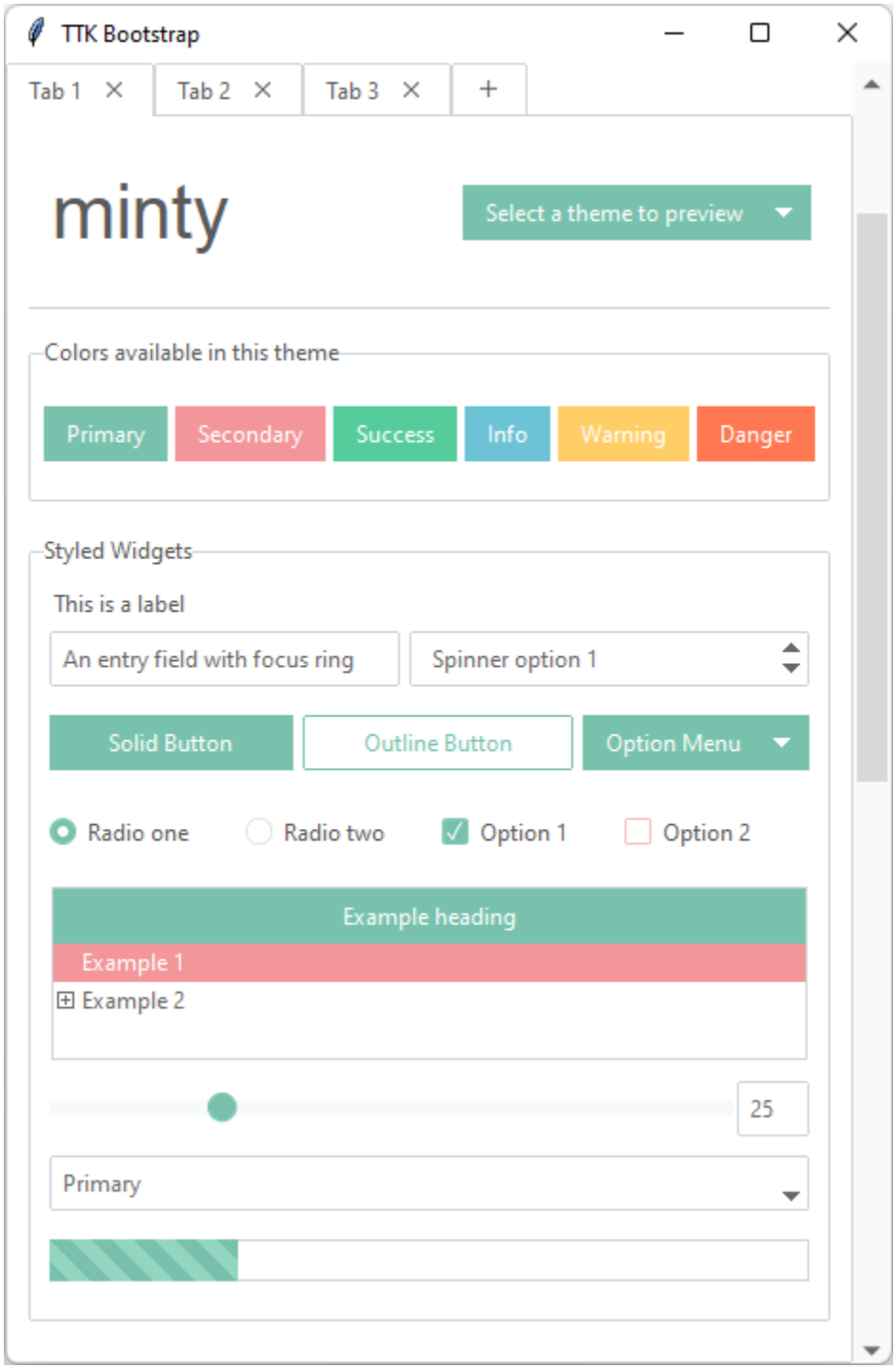


Fig. 6: inspired by https://bootswatch.com/minty/

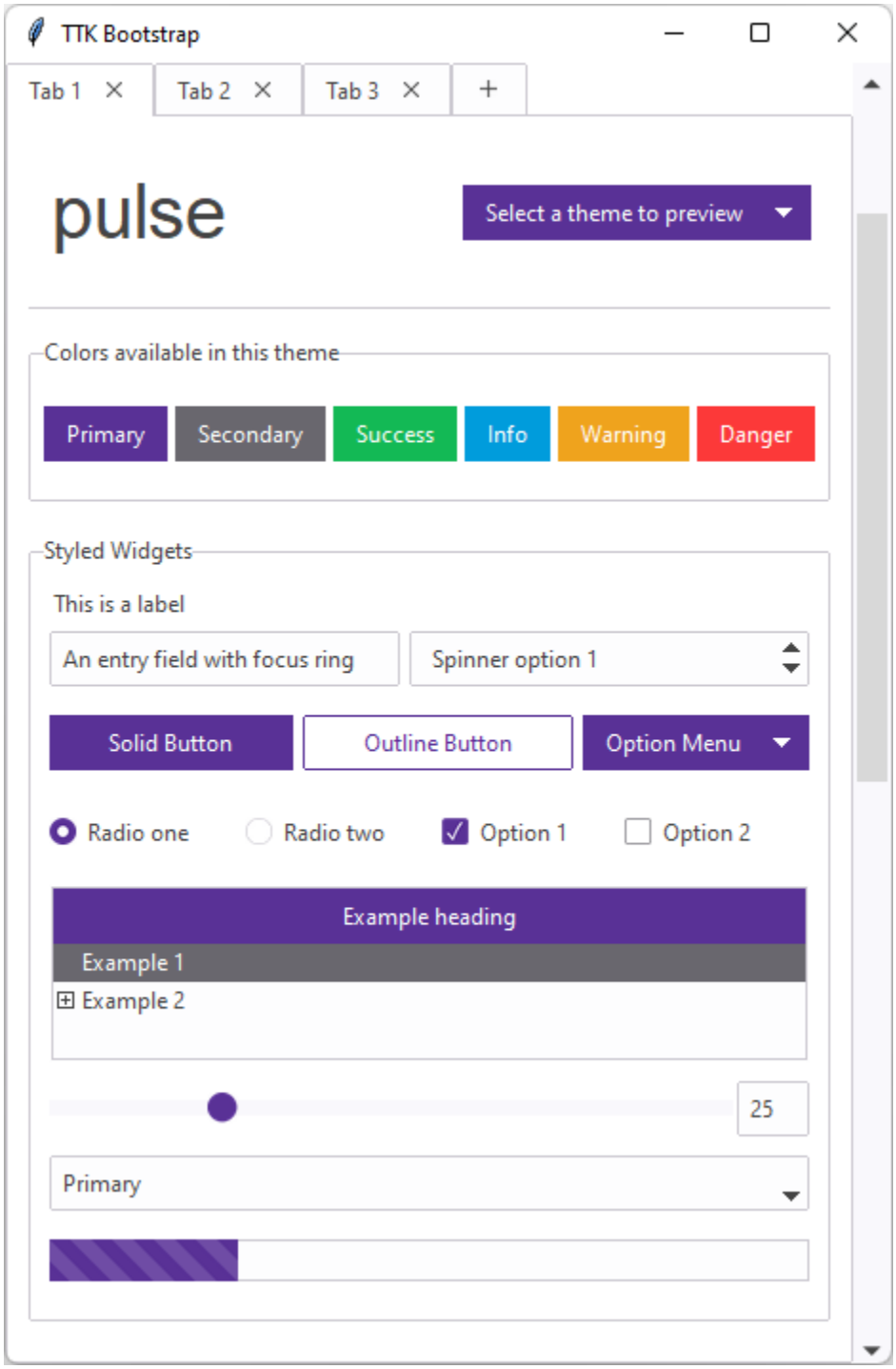


Fig. 7: inspired by https://bootswatch.com/pulse/

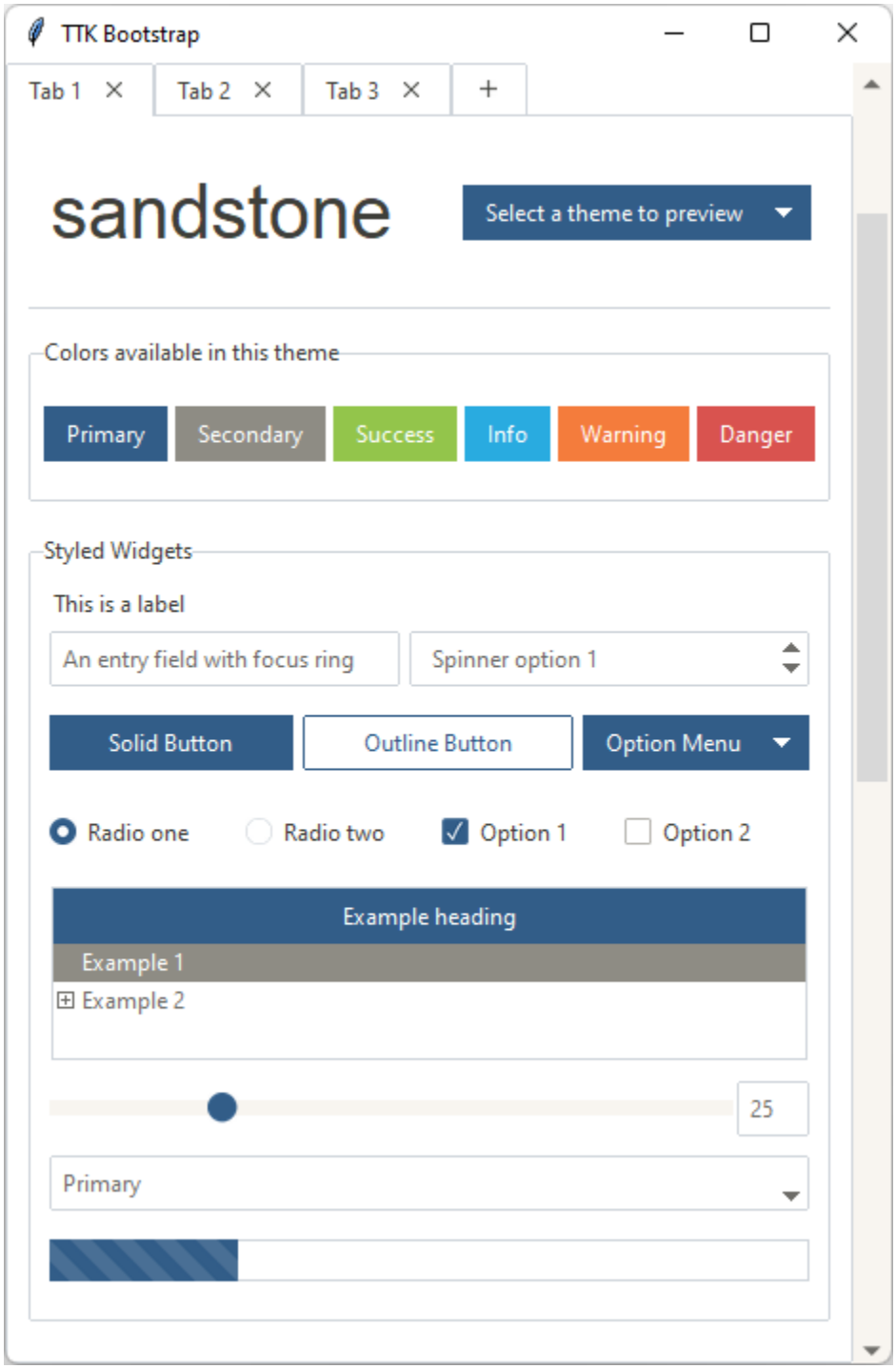


Fig. 8: inspired by https://bootswatch.com/sandstone/

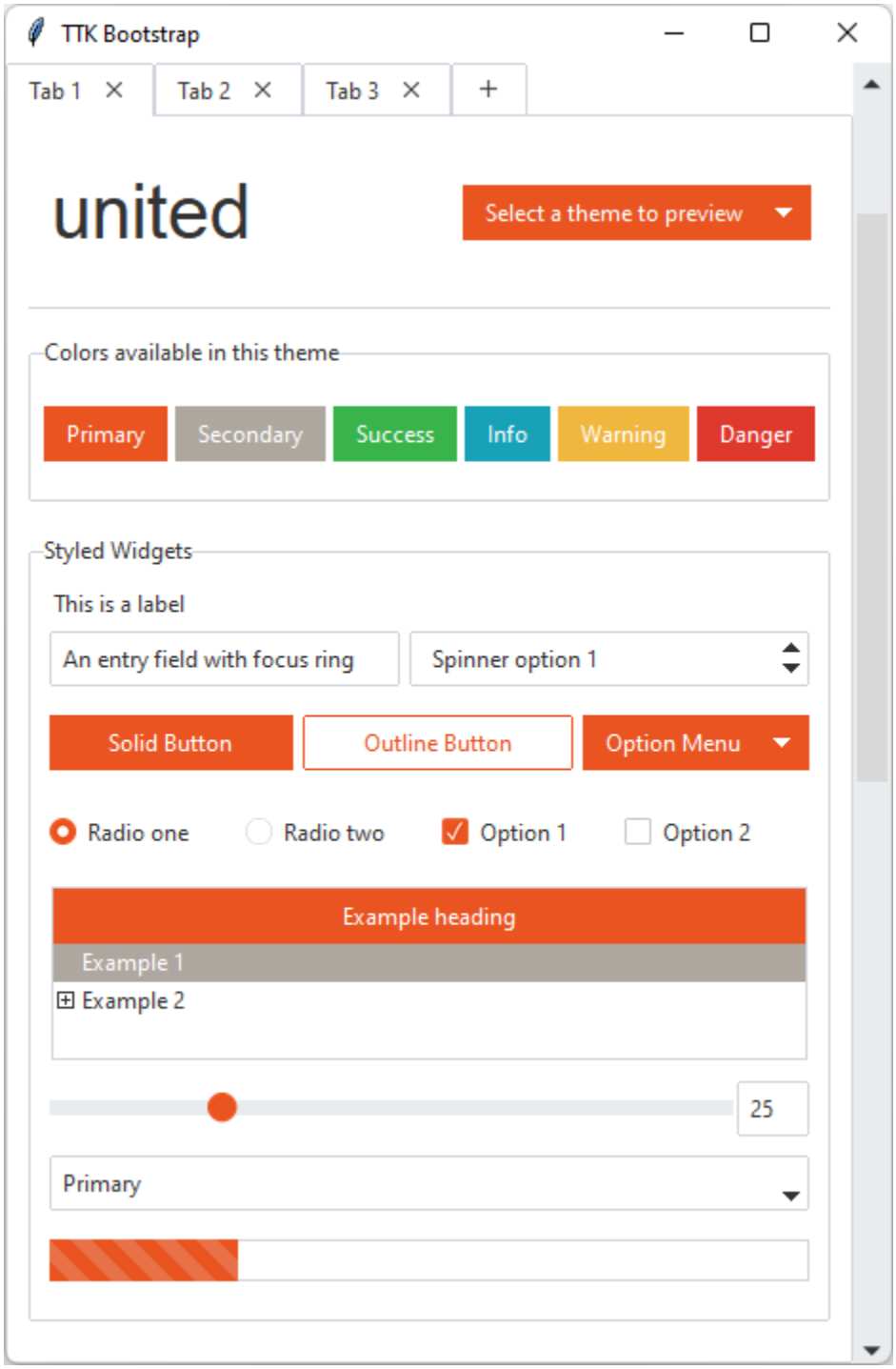


Fig. 9: inspired by https://bootswatch.com/united/

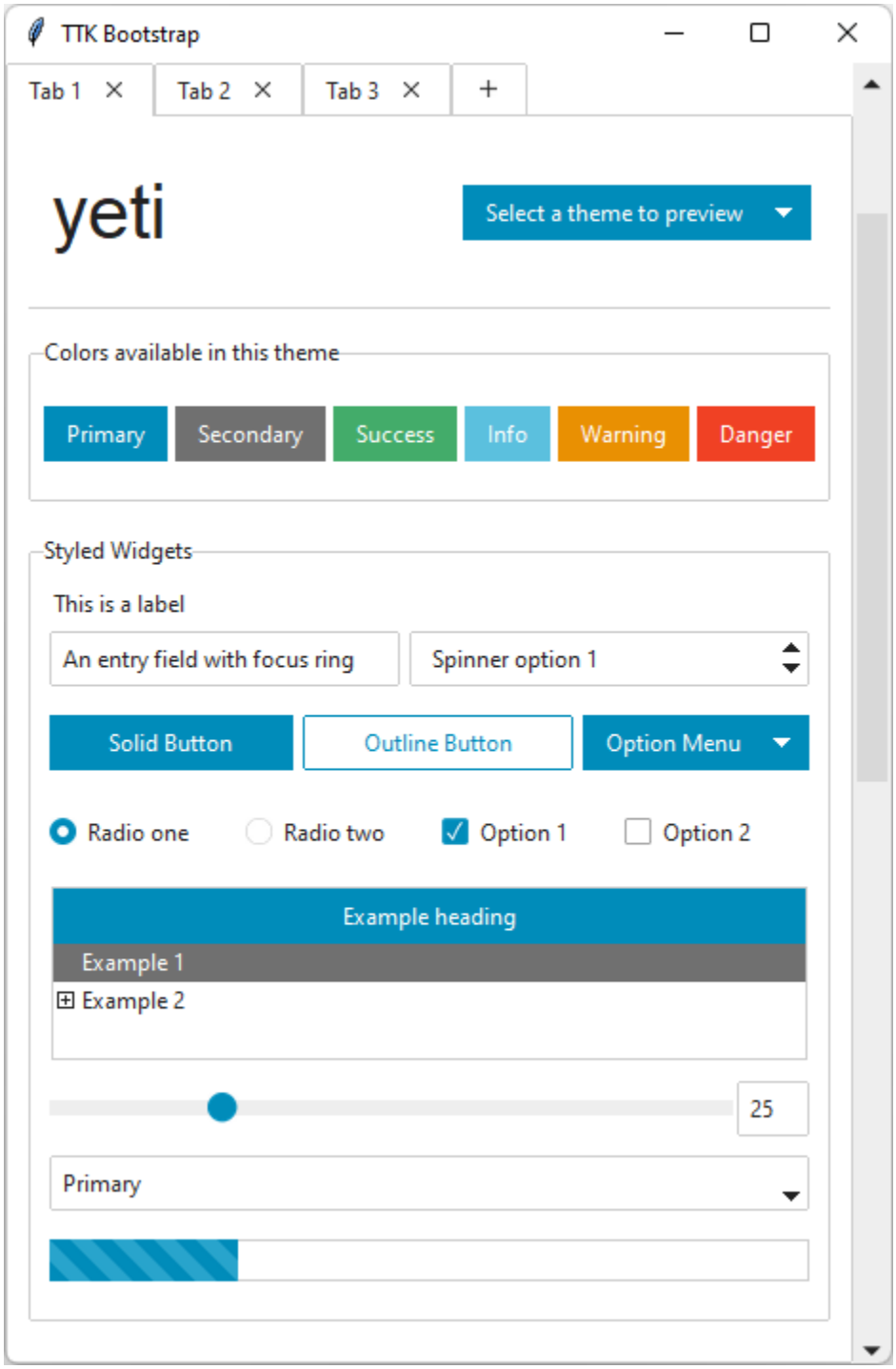


Fig. 10: inspired by https://bootswatch.com/yeti/

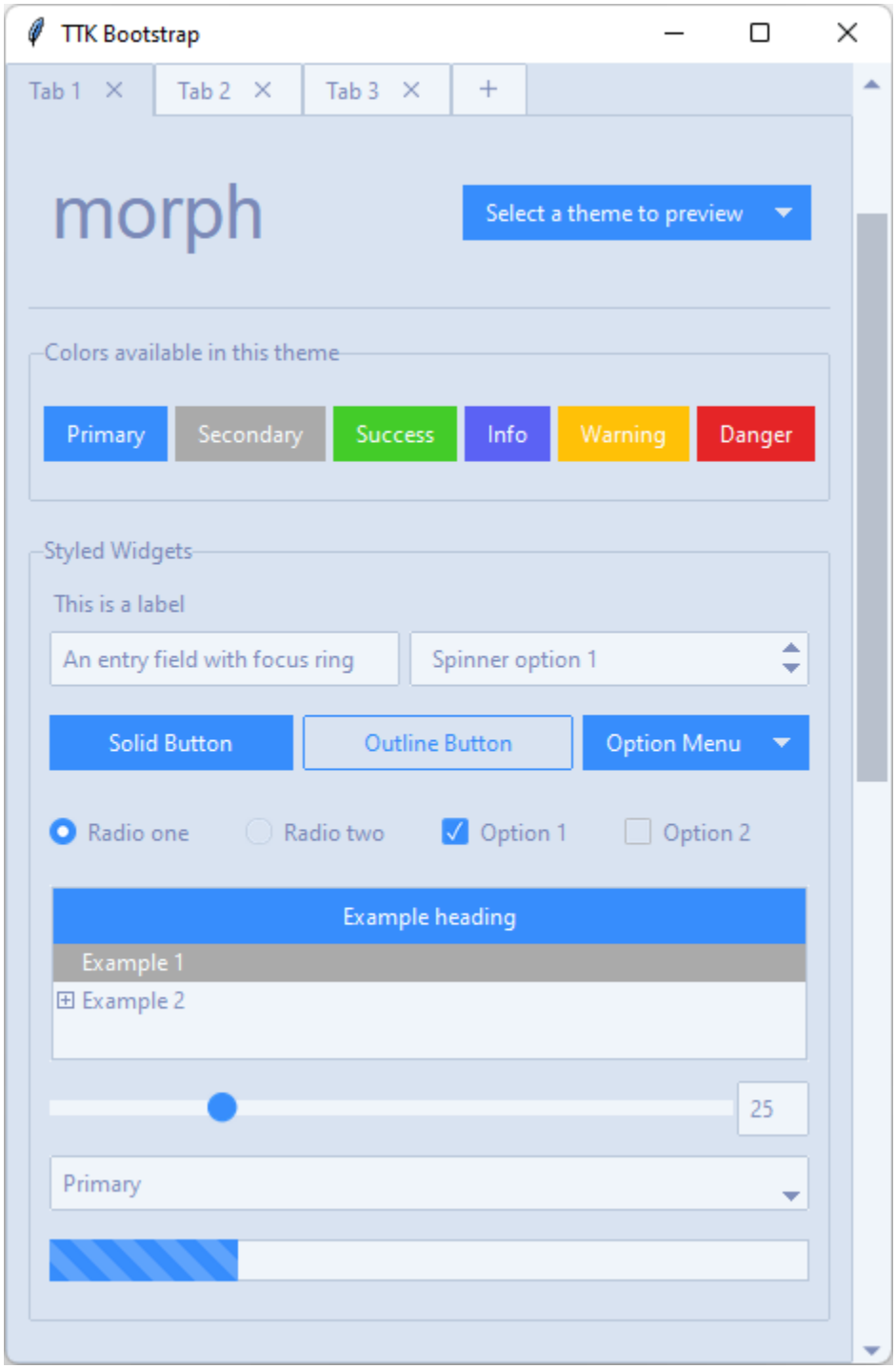


Fig. 11: inspired by https://bootswatch.com/morph/

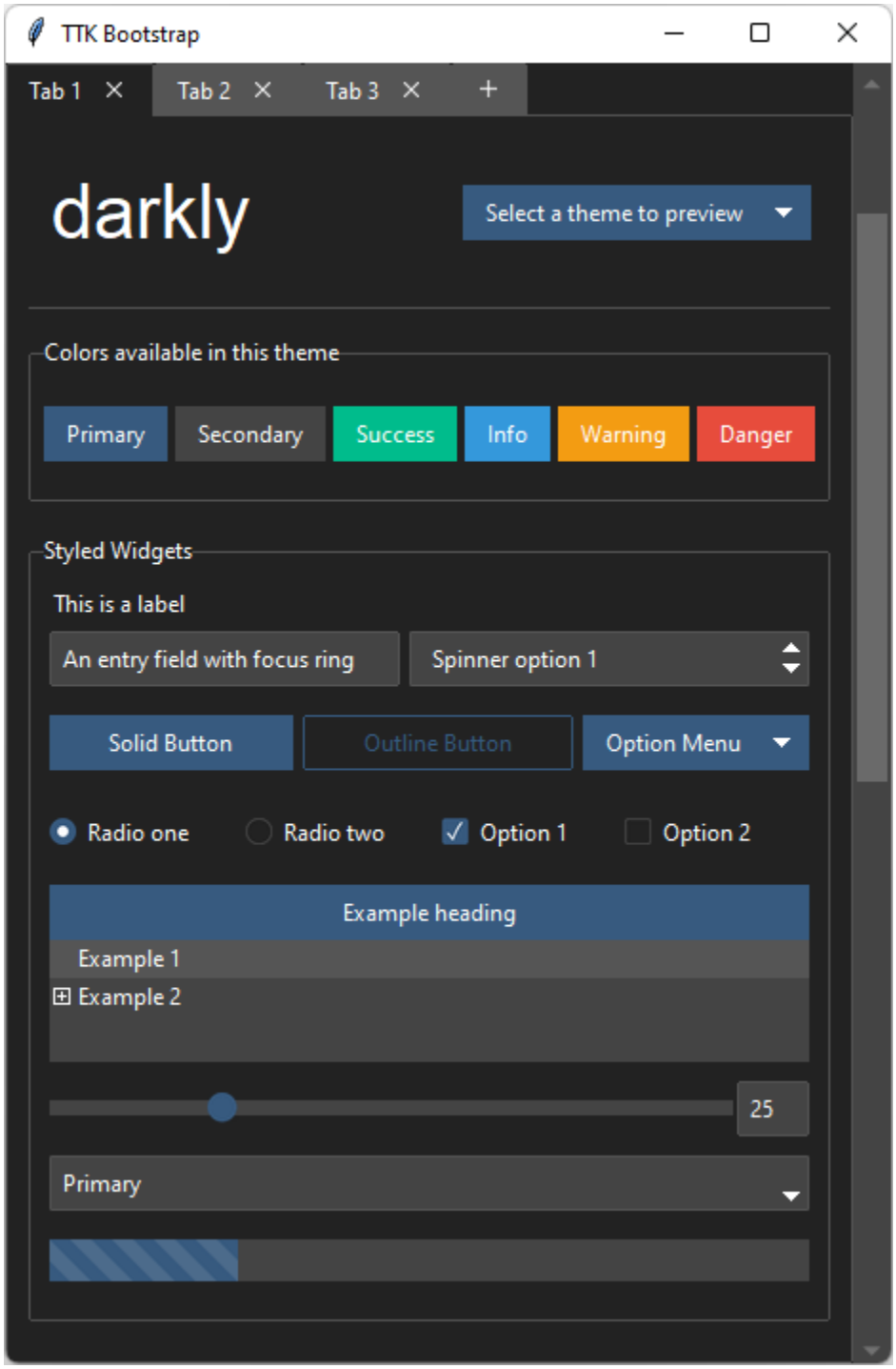


Fig. 12: inspired by https://bootswatch.com/darkly/

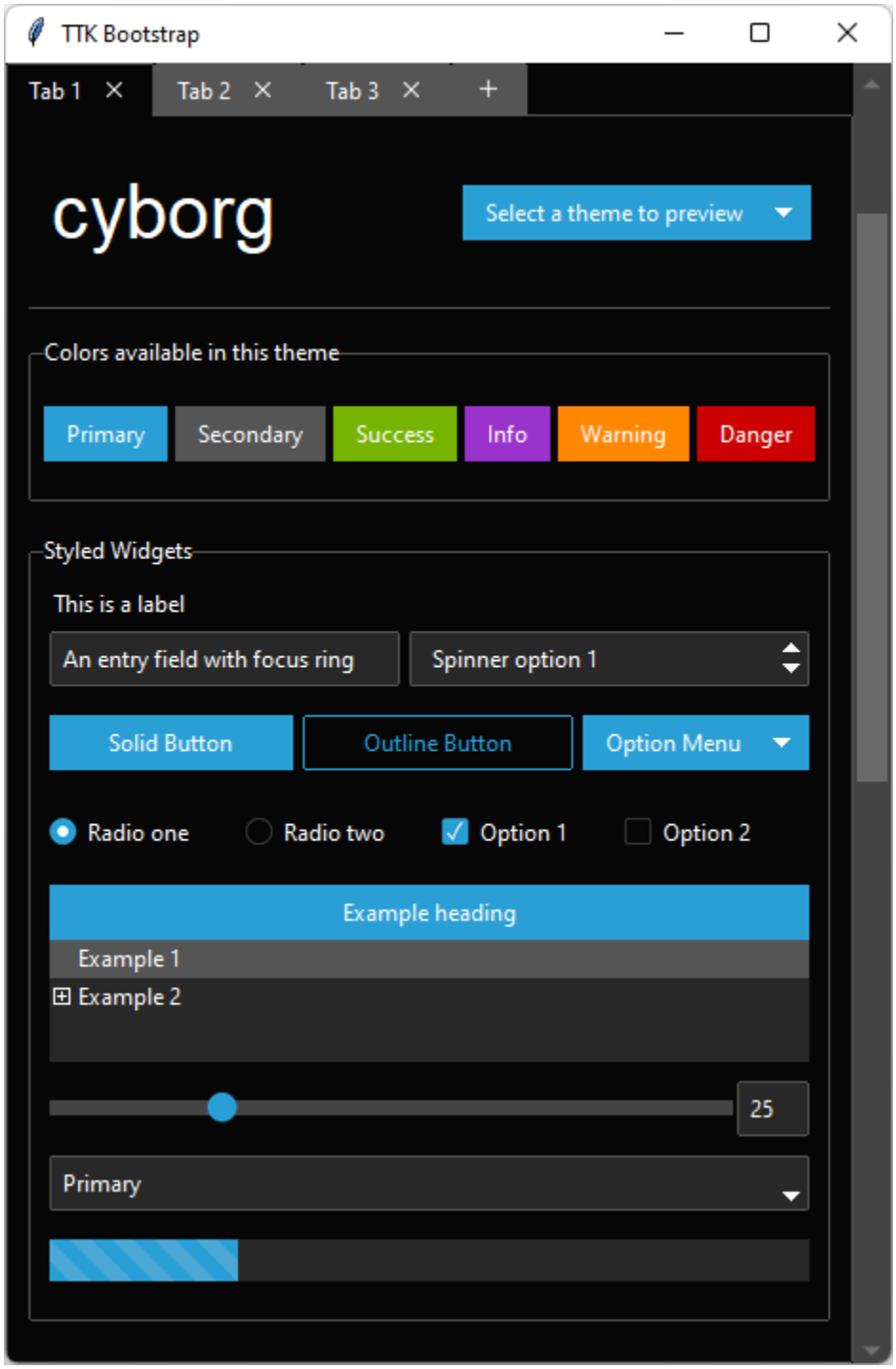


Fig. 13: inspired by https://bootswatch.com/cyborg/

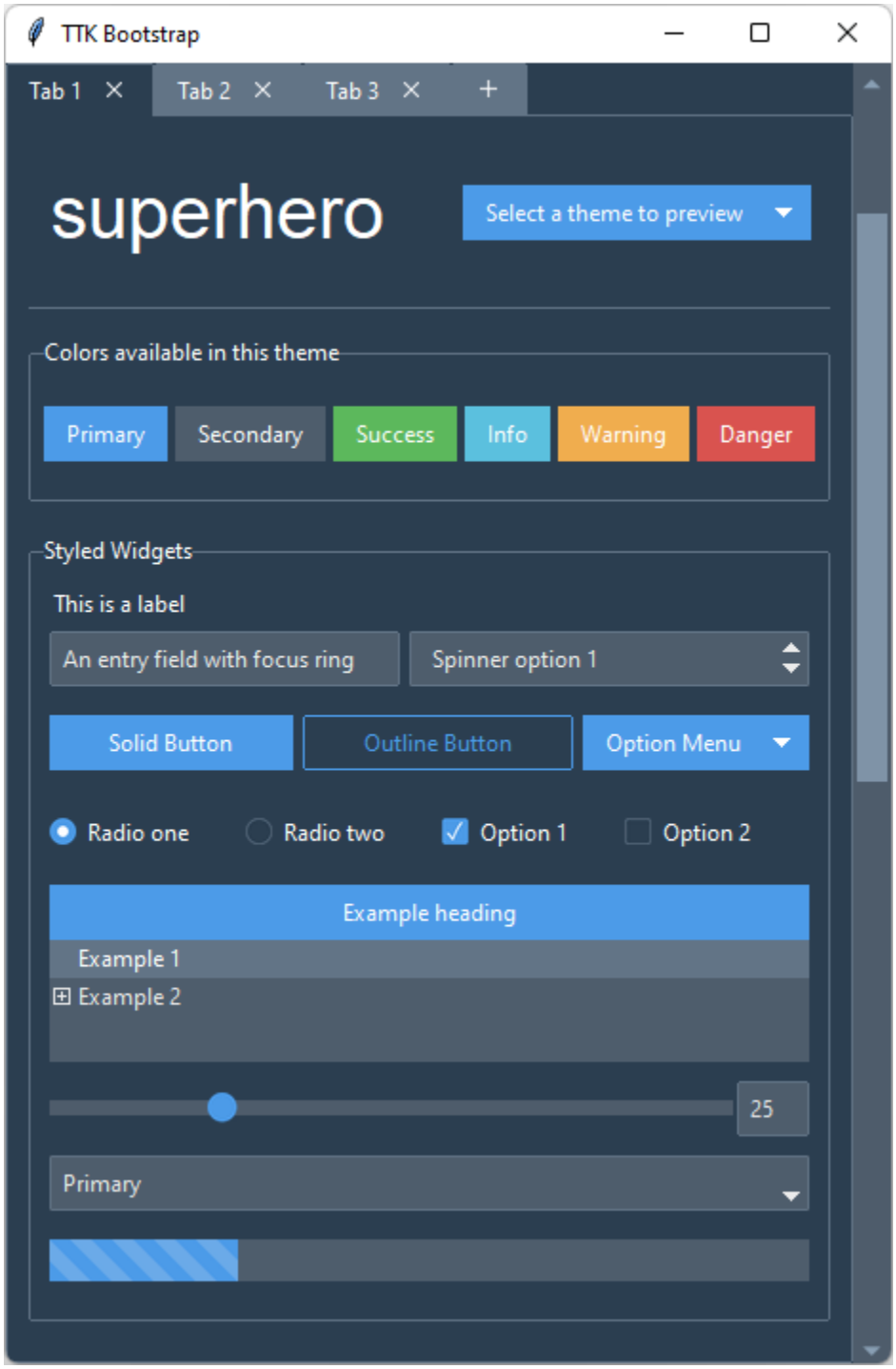


Fig. 14: inspired by https://bootswatch.com/superhero/

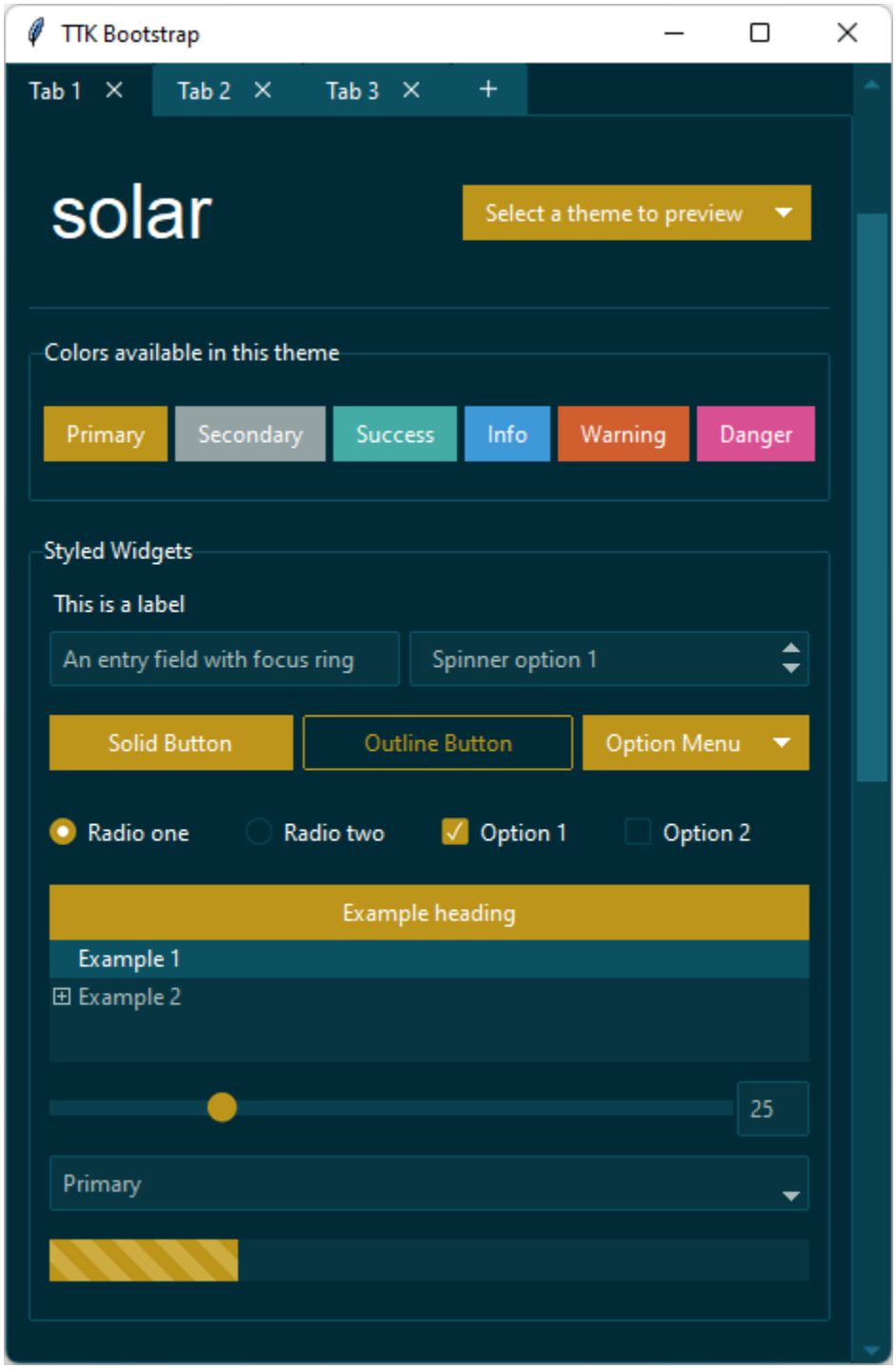


Fig. 15: inspired by https://bootswatch.com/solar/

(continued from previous page)

```
                                "selectfg": "#ffffff",
                                "border": "#ced4da",
                                "inputfg": "#373a3c",
                                "inputbg": "#fdfdfe"
  }
}
```

This theme definition is read by the `ttkbootstrap.Style` class and converted into an actual theme by the `ttkbootstrap.StylerTTK` class at runtime. At that point, it is available to use like any other theme. The only information about a theme that is stored (built-in or user-defined) is the theme definition.

2.3.4 Legacy widget styles

While they are not the focus of this package, if you need to use legacy tkinter widgets, they should not look completely out-of-place. Below is an example of the widgets using the **journal** style. Legacy tkinter widgets will have the primary color applied. If you wish to use other theme colors on the widgets, you can override the styles as you would normally when using tkinter widgets. The theme colors are available in the `Style.colors` property.

2.4 Widget Styles

This is a **style guide** for using ttkbootstrap styles. This guide will show you how to **apply visual styles** to change the look and feel of the widget. If you want more information on how to use the widget and what options are available, consult the *reference section on widgets*.

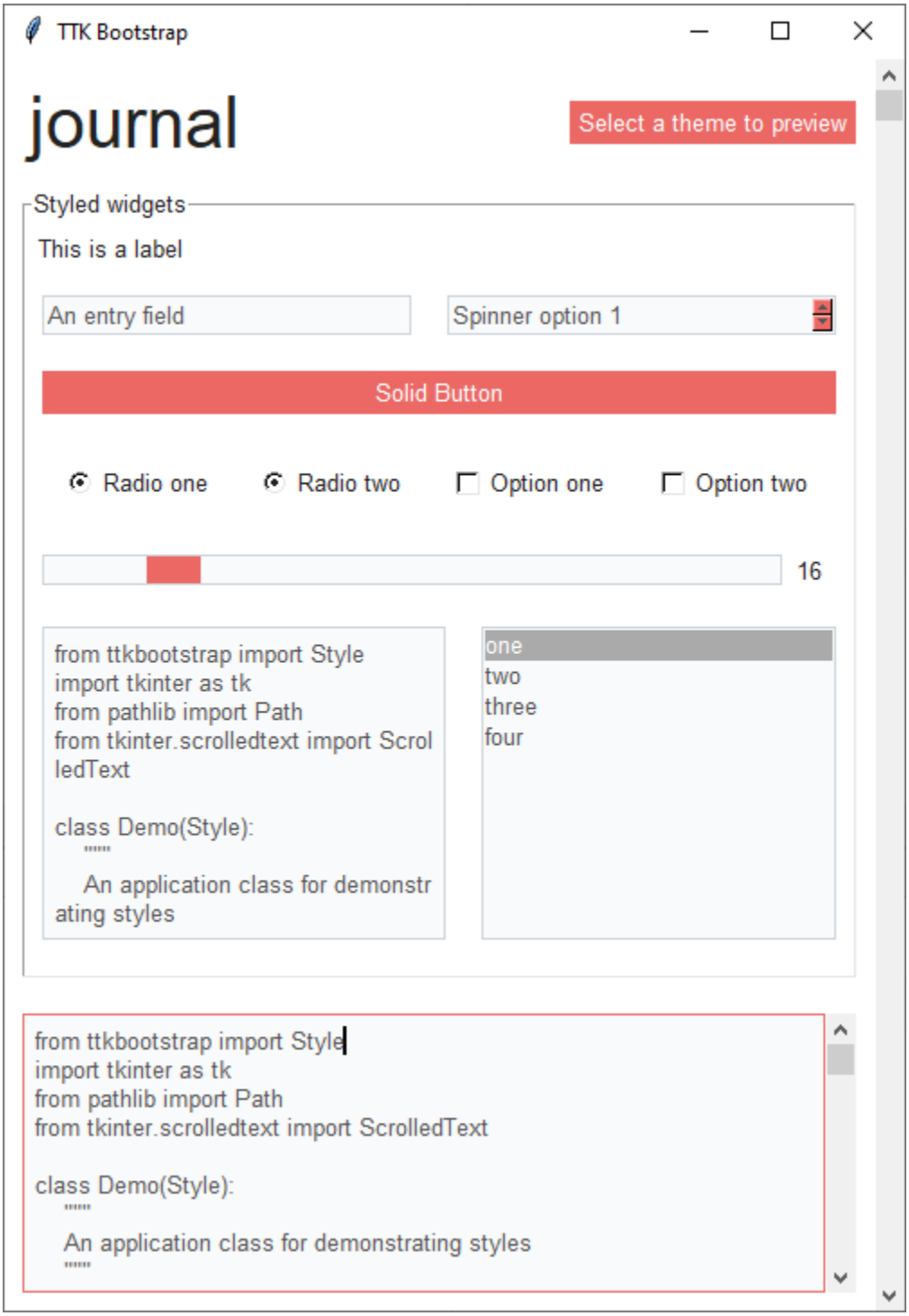
2.4.1 Button

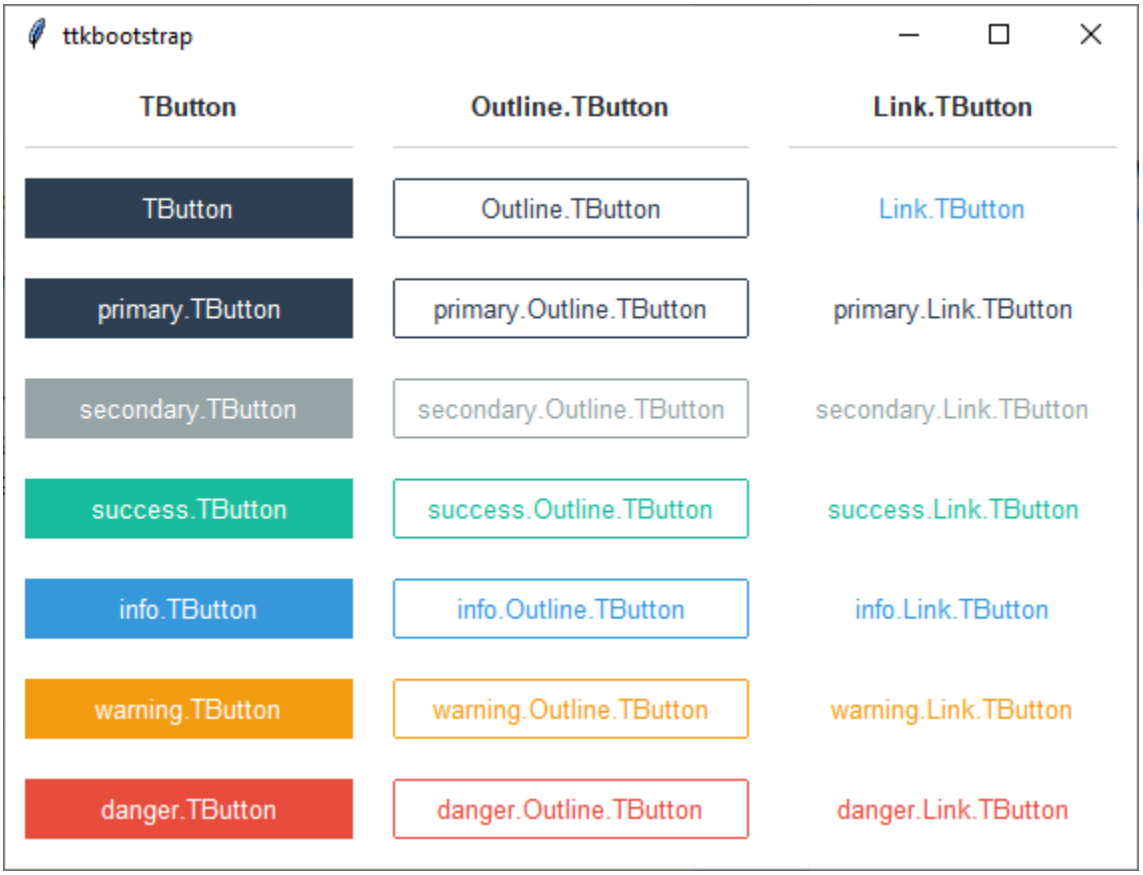
A `ttk.Button` widget displays a textual string, bitmap or image. If text is displayed, it must all be in a single font, but it can occupy multiple lines on the screen (if it contains newlines or if wrapping occurs because of the `wraplength` option) and one of the characters may optionally be underlined using the `underline` option. It can display itself in either of three different ways, according to the `state` option; it can be made to appear raised, sunken, or flat; and it can be made to flash. When a user invokes the button (by pressing mouse button 1 with the cursor over the button), then the command specified in the `command` option is invoked.

Note: This is a **style guide** for using ttkbootstrap styles. This guide will show you how to **apply visual styles** to change the look and feel of the widget. For more information on how to use the widget and what options are available, consult the *reference section on widgets*.

2.4.1.1 Overview

The `ttk.Button` includes the **TButton**, **Outline.TButton**, and **Link.TButton** style classes. The *primary* color is applied to all buttons by default. Other styles must be specified with the `style` option. These styles are further subclassed by each of the theme colors to produce the following color and style combinations:


TTK Bootstrap
journal
Select a theme to preview
Styled widgets
This is a label
An entry field
Spinner option 1
Solid Button
Radio one
Radio two
Option one
Option two
16
from ttkbootstrap import Style
import tkinter as tk
from pathlib import Path
from tkinter.scrolledtext import Scrol
ledText
class Demo(Style):
"""
An application class for demonstr
ating styles
one
two
three
four
from ttkbootstrap import Style
import tkinter as tk
from pathlib import Path
from tkinter.scrolledtext import ScrolledText
class Demo(Style):
"""
An application class for demonstrating styles
"""



The **Link.TButton** has an *info* colored hover effect as well as a slight *shiftrelief* when the button is pressed.

2.4.1.2 How to use

The examples below demonstrate how to *use a style* when creating 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 **solid button**

```
ttk.Button(parent, text='Submit')
```

Create a default **outline button**

```
ttk.Button(parent, text='Submit', style='Outline.TButton')
```

Create an **'info' solid button**

```
ttk.Button(parent, text='Submit', style='info.TButton')
```

Create a **'warning' outline button**

```
ttk.Button(parent, text="Submit", style='warning.Outline.TButton')
```

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

Style options

anchor *e, w, center*

background *color*

bordercolor *color*

compound *top, bottom, left, right*

darkcolor *color*

embossed *amount*

focuscolor *color*

focusthickness *amount*

foreground *color*

font *font*

highlightcolor *color*

highlightthickness *amount*

justify *left, right, center*

lightcolor *color*

padding *padding*

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

shiftrelief *amount*

width *amount*

2.4.1.4 Create a custom style

Change the **font** and **font-size** on all buttons

```
Style.configure('TButton', font=('Helvetica', 12))
```

Change the **foreground color** when the button is active

```
Style.map('TButton', foreground=[
    ('disabled', 'white'),
    ('active', 'yellow')])
```

Subclass an existing style to create a new one, using the pattern 'newstyle.OldStyle'

```
Style.configure('custom.TButton', background='red', foreground='white', font=('Helvetica
↪', 24))
```

Use a custom style

```
ttk.Button(parent, text='Submit', style='custom.TButton')
```

2.4.1.5 References

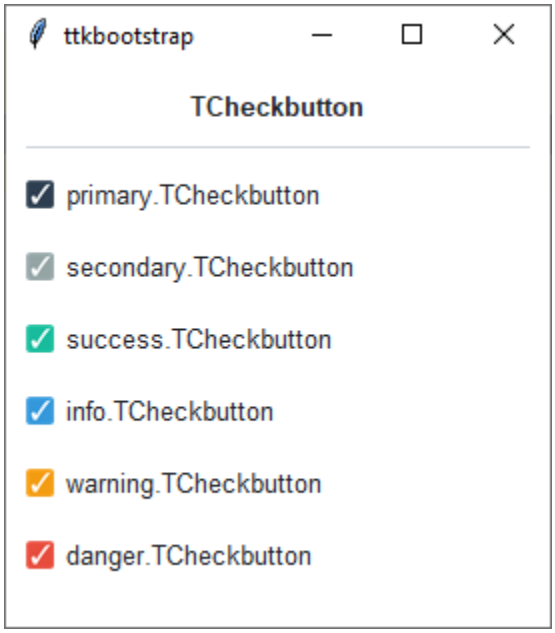
- https://www.pythontutorial.net/tkinter/tkinter-button/
- https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/ttk-Button.html
- https://www.tcl.tk/man/tcl8.6/TkCmd/ttk_button.htm

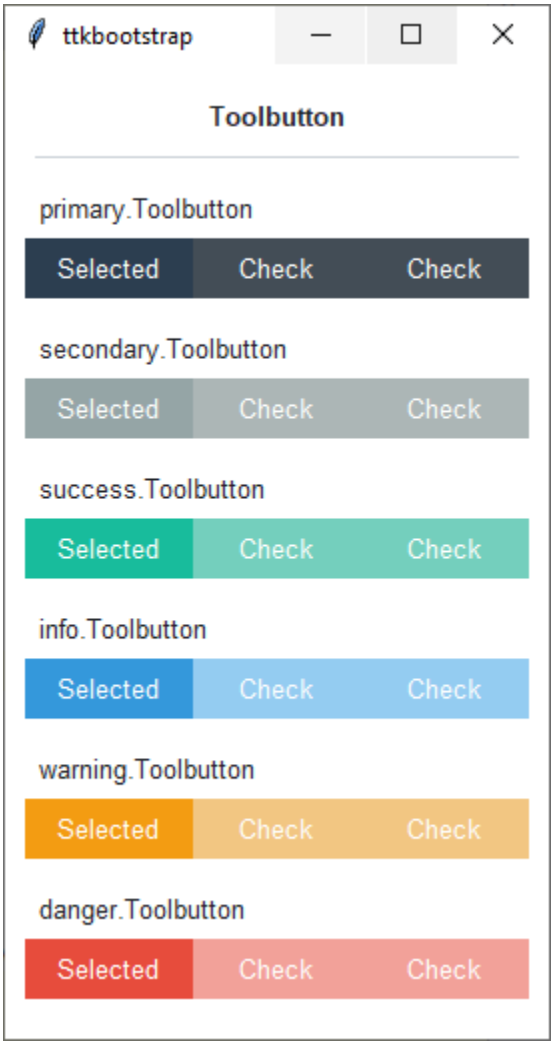
2.4.2 Checkbutton

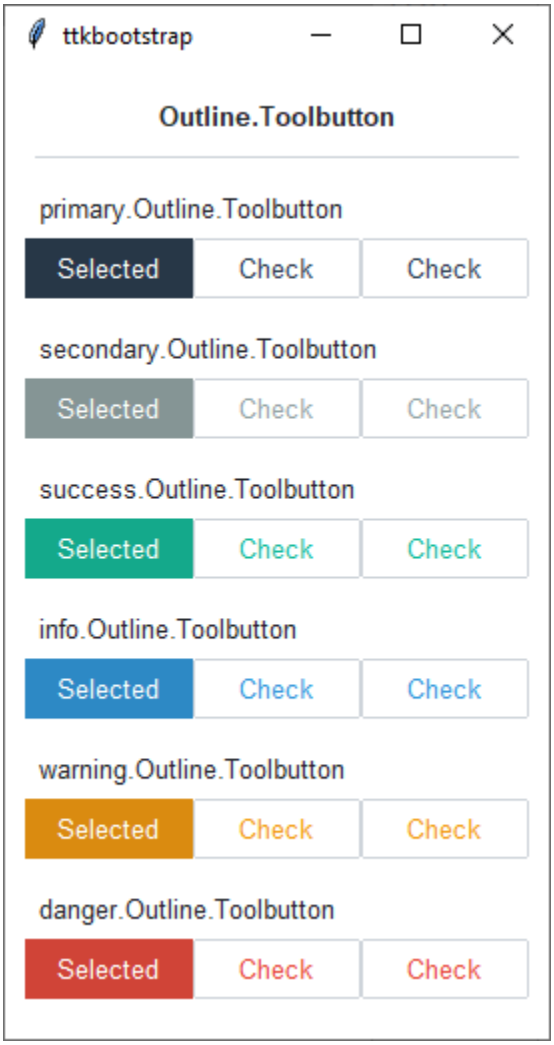
A `ttk.Checkbutton` widget is used to show or change a setting. It has two states, selected and deselected. The state of the checkbutton may be linked to a tkinter variable.

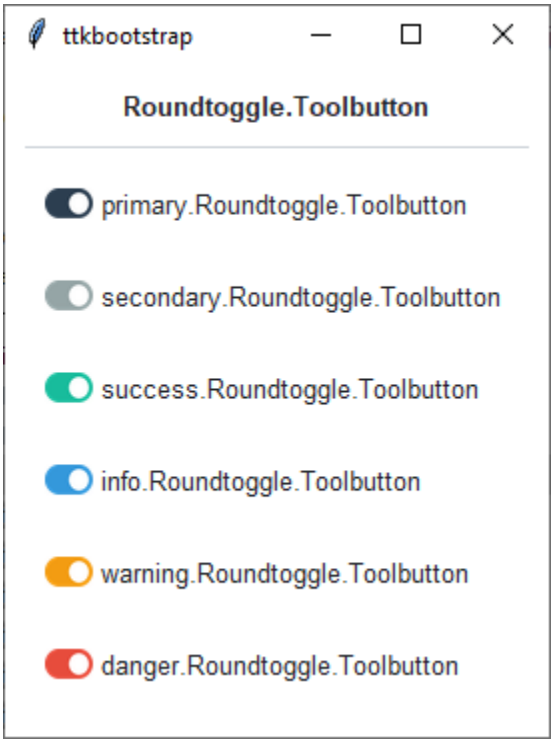
2.4.2.1 Overview

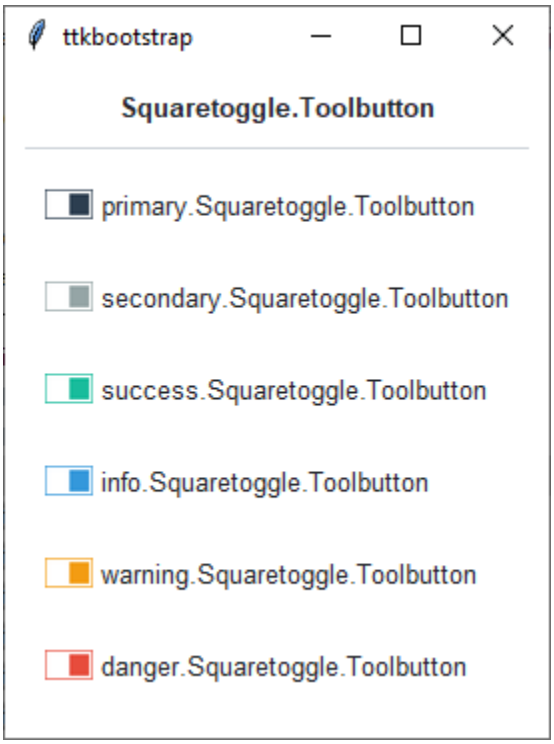
The `ttk.Checkbutton` includes the **TCheckbutton**, **Toolbutton**, **Outline.Toolbutton**, **Roundtoggle.Toolbutton**, and **Squaretoggle.Toolbutton** style classes. The **primary.TCheckbutton** style is applied to all checkbuttons by default. Other styles must be specified with the `style` option. These primary styles are further subclassed by each of the theme colors to produce the following color and style combinations:


ttkbootstrap
TCheckbutton
primary.TCheckbutton
secondary.TCheckbutton
success.TCheckbutton
info.TCheckbutton
warning.TCheckbutton
danger.TCheckbutton


ttkbootstrap
Toolbutton
primary.Toolbutton
Selected
Check
Check
secondary.Toolbutton
Selected
Check
Check
success.Toolbutton
Selected
Check
Check
info.Toolbutton
Selected
Check
Check
warning.Toolbutton
Selected
Check
Check
danger.Toolbutton
Selected
Check
Check


ttkbootstrap
Outline.Toolbutton
primary.Outline.Toolbutton
Selected
Check
Check
secondary.Outline.Toolbutton
Selected
Check
Check
success.Outline.Toolbutton
Selected
Check
Check
info.Outline.Toolbutton
Selected
Check
Check
warning.Outline.Toolbutton
Selected
Check
Check
danger.Outline.Toolbutton
Selected
Check
Check


ttkbootstrap
Roundtoggle.Toolbutton
primary.Roundtoggle.Toolbutton
secondary.Roundtoggle.Toolbutton
success.Roundtoggle.Toolbutton
info.Roundtoggle.Toolbutton
warning.Roundtoggle.Toolbutton
danger.Roundtoggle.Toolbutton


ttkbootstrap
Squaretoggle.Toolbutton
primary.Squaretoggle.Toolbutton
secondary.Squaretoggle.Toolbutton
success.Squaretoggle.Toolbutton
info.Squaretoggle.Toolbutton
warning.Squaretoggle.Toolbutton
danger.Squaretoggle.Toolbutton

2.4.2.2 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 **checkbutton**

```
ttk.Checkbutton(parent, text='include', value=1)
```

Create a default **toolbutton**

```
ttk.Checkbutton(parent, text='include', style='Toolbutton')
```

Create a default **outline toolbutton**

```
ttk.Checkbutton(parent, text='include', style='Outline.Toolbutton')
```

Create a default **round toggle** toolbutton

```
ttk.Checkbutton(parent, text='include', style='Roundtoggle.Toolbutton')
```

Create a default **square toggle** toolbutton

```
ttk.Checkbutton(parent, text='include', style='Squaretoggle.Toolbutton')
```

Create an **'info' checkbutton**

```
ttk.Checkbutton(parent, text='include', style='info.TCheckbutton')
```

Create a **'warning' outline toolbutton**

```
ttk.Checkbutton(parent, text="include", style='warning.Outline.Toolbutton')
```

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
- readonly

Style options

background *color*

compound *compound*

foreground *foreground*

focuscolor *color*

focusthickness *amount*

font *font*

padding *padding*

2.4.2.4 Create a custom style

Change the **font** and **font-size** on all checkbuttons

```
Style.configure('TCheckbutton', font=('Helvetica', 12))
```

Change the **foreground color** when the checkbutton is **selected**

```
Style.map('TCheckbutton', foreground=[
    ('disabled', 'white'),
    ('selected', 'yellow'),
    ('!selected', 'gray')])
```

Subclass an existing style to create a new one, using the pattern 'newstyle.OldStyle'

```
Style.configure('custom.TCheckbutton', foreground='white', font=('Helvetica', 24))
```

Use a custom style

```
ttk.Checkbutton(parent, text='include', style='custom.TCheckbutton')
```

2.4.2.5 References

- https://www.pythontutorial.net/tkinter/tkinter-checkbox/
- https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/ttk-Checkbutton.html
- https://www.tcl.tk/man/tcl8.6/TkCmd/ttk_checkbutton.htm

2.4.3 Calendar

The calendar module contains several classes and functions that enable the user to select a date.

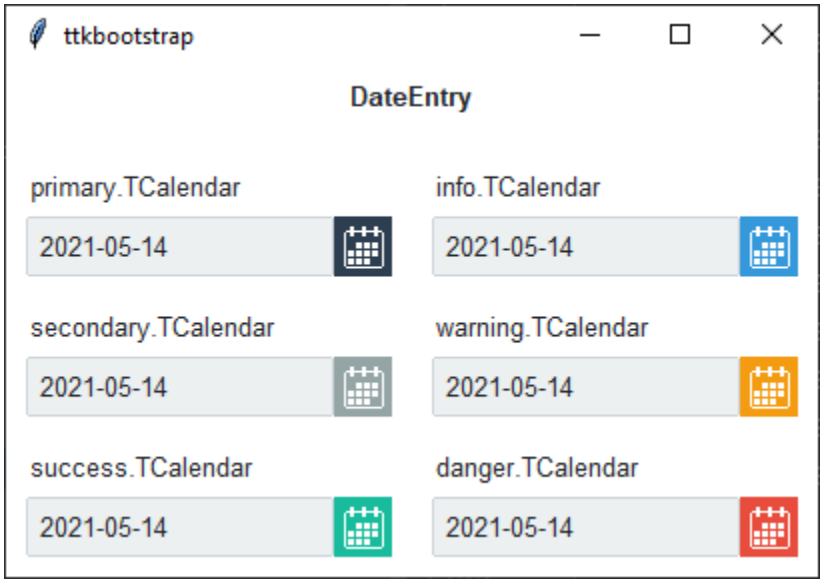
Note: This is a **style guide** for using ttkbootstrap styles. This guide will show you how to **apply visual styles** to change the look and feel of the widget. For more information on how to use the widget and what options are available, consult the *reference section on widgets*.

2.4.3.1 Overview

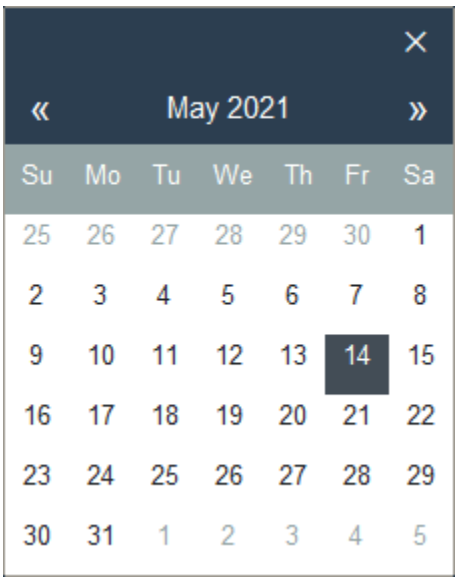
The `DateEntry` and `DateChooserPopup` are the two classes that you will use along with the `calendar.ask_date()` helper function.

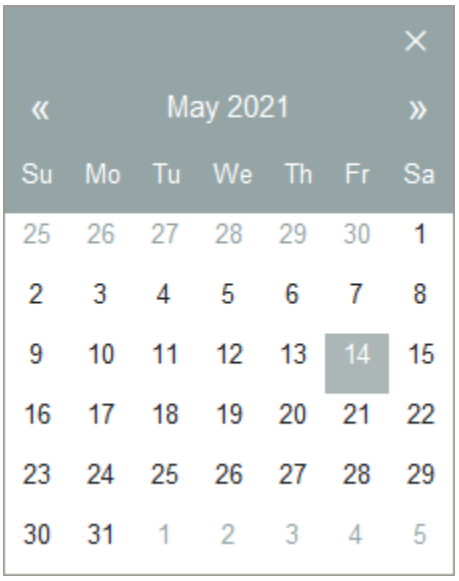
The `DateEntry` widget is a `ttk.Entry` widget combined with a `ttk.Button` widget that opens up a `DateChooserPopup` when pressed. It is recommended to *not use* the `DateChooserPopup` directly, unless you want to subclass it, but rather to use it via the `calendar.ask_date()` method, which opens up a `DateChooserPopup` and returns the selected value as a `datetime` object.

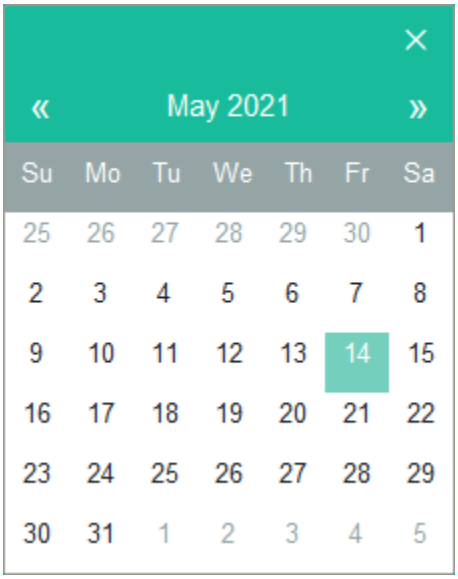
All of these objects have a `style` parameter that accept a **TCalendar** style. By default, the *primary* color is applied to the widget. However, the base style is further subclassed by each of the theme colors to produce the following color and style combinations for `DateEntry` and `DateChooserPopup`.

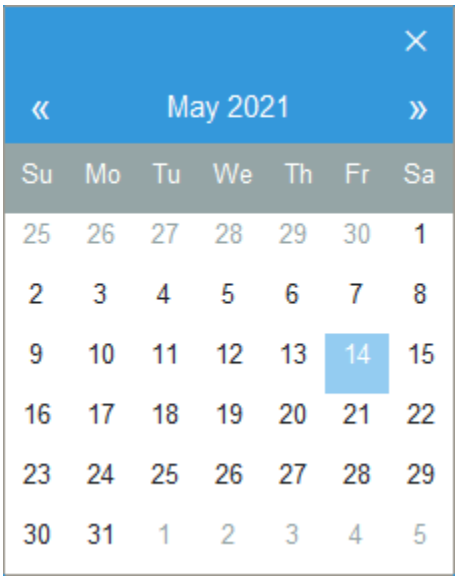


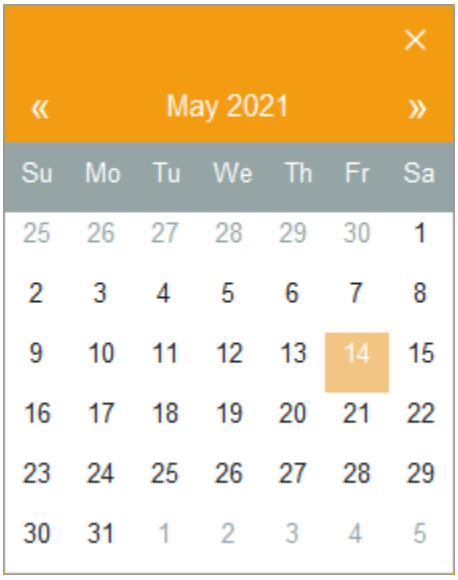
The styles above correspond to the same colored `DateChooserPopup` below:

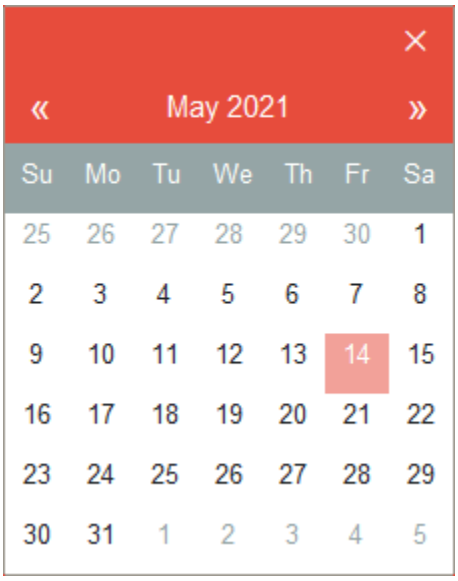

May 2021
Su Mo Tu We Th Fr Sa
25 26 27 28 29 30 1
2 3 4 5 6 7 8
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31 1 2 3 4 5


May 2021
Su Mo Tu We Th Fr Sa
25 26 27 28 29 30 1
2 3 4 5 6 7 8
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31 1 2 3 4 5


«
May 2021
»
Su Mo Tu We Th Fr Sa
25 26 27 28 29 30 1
2 3 4 5 6 7 8
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31 1 2 3 4 5


«
May 2021
»
Su Mo Tu We Th Fr Sa
25 26 27 28 29 30 1
2 3 4 5 6 7 8
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31 1 2 3 4 5





2.4.3.2 How to use

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

Create a default **date entry**

```
DateEntry(parent)
```

Create a **success date entry**

```
DateEntry(parent, style='success.TCalendar')
```

Create a button that calls the calendar popup by assigning a callback.

```
def callback():
    return ask_date()
```

(continues on next page)

(continued from previous page)

```
btn = ttk.Button(parent, text='Get Date', command=callback)
```

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

Style options

background *color*

compound *compound*

foreground *foreground*

focuscolor *color*

focusthickness *amount*

font *font*

padding *padding*

2.4.3.4 Create a custom style

Change the **font** and **font-size** on all calendar buttons

```
Style.configure('TCalendar', font=('helvetica', 12))
```

Change the **foreground color** when the calendar date is **selected**

```
Style.map('TCalendar', foreground=[
    ('disabled', 'white'),
    ('selected', 'yellow'),
    ('!selected', 'gray')])
```

Subclass an existing style to create a new one, using the pattern 'newstyle.OldStyle'

```
Style.configure('custom.TCalendar', foreground='tan', font=('Helvetica', 10))
```

Use a custom style

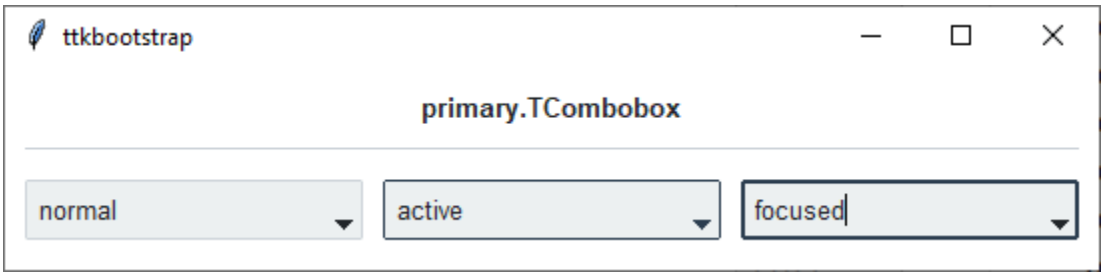
```
DateEntry(parent, style='custom.TCalendar')
```

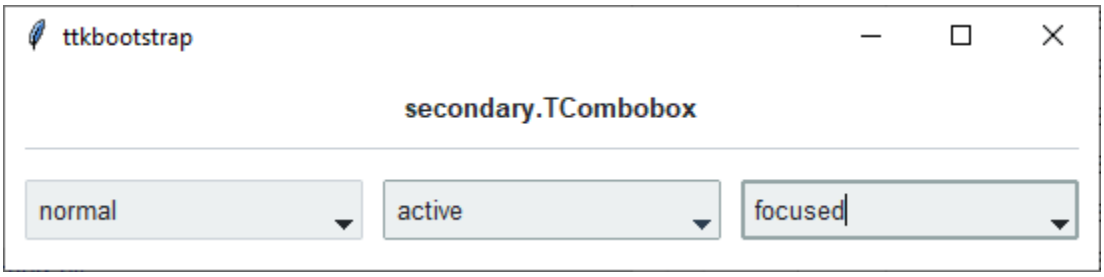
2.4.4 Combobox

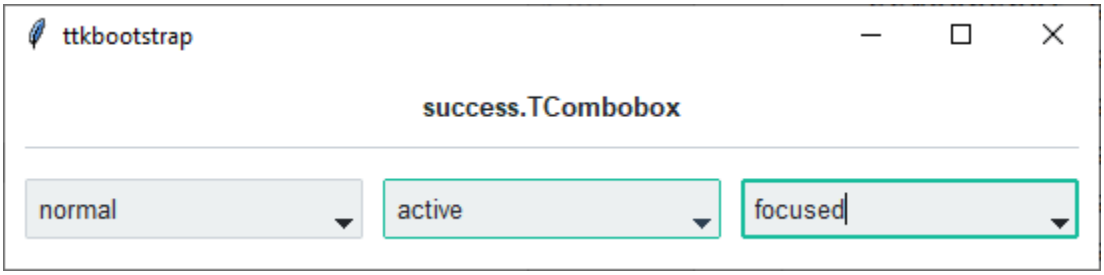
A `ttk.Combobox` widget is a combination of an Entry and a drop-down menu. In your application, you will see the usual text entry area, with a downward-pointing arrow. When the user clicks on the arrow, a drop-down menu appears. If the user clicks on one, that choice replaces the current contents of the entry. However, the user may still type text directly into the entry (when it has focus), or edit the current text.

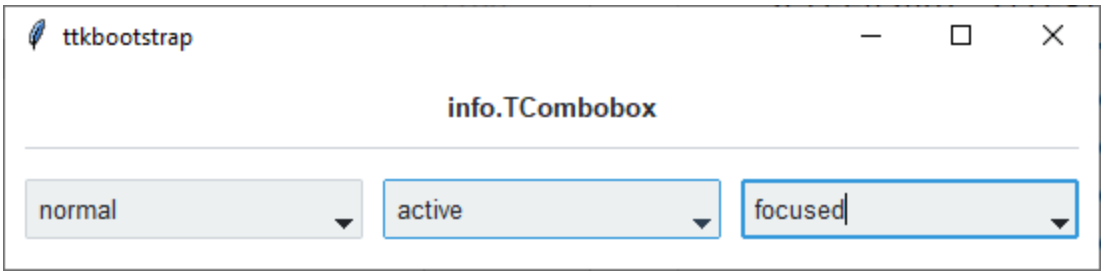
2.4.4.1 Overview

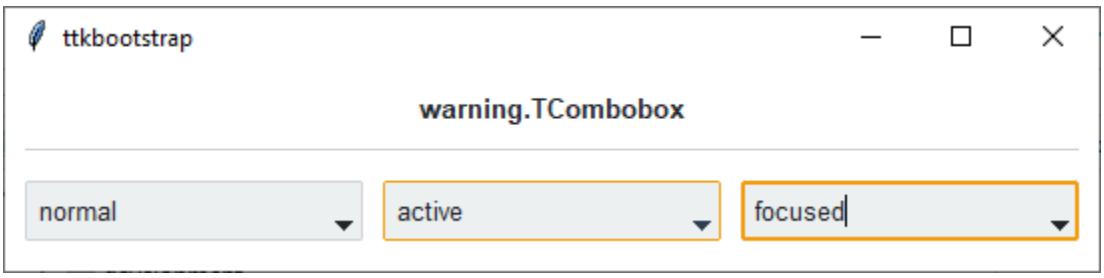
The `ttk.Combobox` includes the **TCombobox** class. The *primary* color is applied by default. This style is further subclassed by each of the theme colors to produce the following color and style combinations.











As you can see, in a *normal* state, all styles look the same. What distinguishes them are the colors that are used for the **active** and **focused** states.

2.4.4.2 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 **combobox**

```
cb = ttk.Combobox(parent)

for option in ['option 1', 'option 2', 'option 3']:
    cb.insert('end', option)
```

Create an **'info' combobox**

```
ttk.Combobox(parent, style='info.TCombobox')
```

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

Style options

anchor *e, w, center*

arrowcolor *color*

arrowsize *amount*

background *color*

bordercolor *color*

borderwidth *amount*

darkcolor *color*

fieldbackground *color*

foreground *color*

font *font*

lightcolor *color*

padding *padding*

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

width *amount*

Note: The popdown list cannot be configured using the `Style` class. Instead, you must use the tk option database.

- `tk.option_add('*TCombobox*Listbox.background', color)`
- `tk.option_add('*TCombobox*Listbox.font', font)`
- `tk.option_add('*TCombobox*Listbox.foreground', color)`
- `tk.option_add('*TCombobox*Listbox.selectBackground', color)`

- `tk.option_add('*TCombobox*Listbox.selectForeground', color)`

2.4.4.4 Create a custom style

Change the **font** and **font-size** on all comboboxes

```
Style.configure('TCombobox', font=('Helvetica', 12))
```

Change the **arrow color** when in different states

```
Style.map('TCombobox', arrowcolor=[
    ('disabled', 'gray'),
    ('pressed !disabled', 'blue'),
    ('focus !disabled', 'green'),
    ('hover !disabled', 'yellow')])
```

Subclass an existing style to create a new one, using the pattern 'newstyle.OldStyle'

```
Style.configure('custom.TCombobox', background='green', foreground='white', font=(
↪'Helvetica', 24))
```

Use a custom style

```
ttk.Combobox(parent, style='custom.TCombobox')
```

2.4.4.5 References

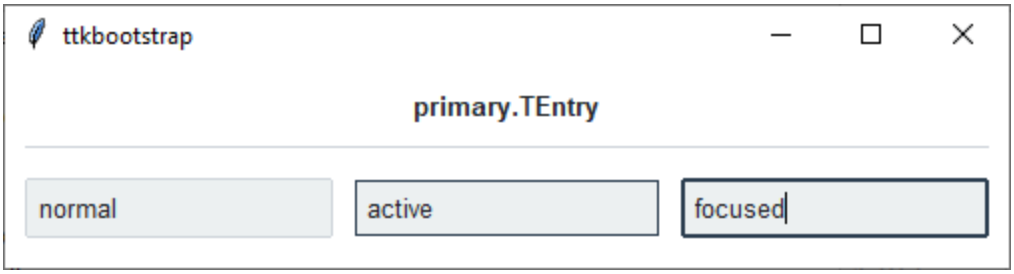
- https://www.pythontutorial.net/tkinter/tkinter-combobox/
- https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/ttk-Combobox.html
- https://tcl.tk/man/tcl8.6/TkCmd/ttk_combobox.htm

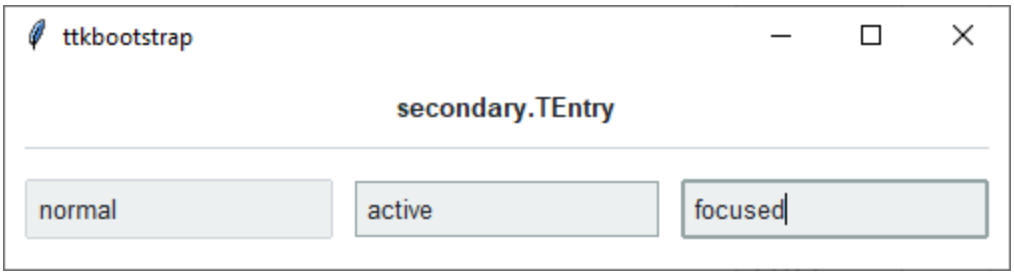
2.4.5 Entry

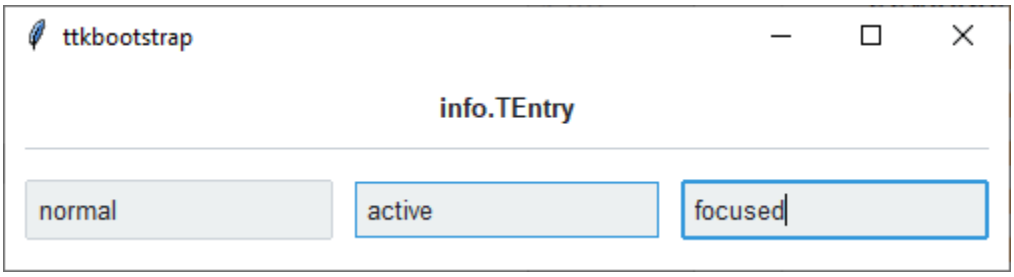
The *ttk.Entry* widget displays a one-line text string and allows that string to be edited by the user. The value of the string may be linked to a tkinter variable with the `textvariable` option. Entry widgets support horizontal scrolling with the standard `xscrollcommand` option and `xview` widget command.

2.4.5.1 Overview

The `ttk.Entry` includes the **TEntry** class. The *primary* color is applied by default. This style is further subclassed by each of the theme colors to produce the following color and style combinations.







As you can see, in a *normal* state, all styles look the same. What distinguishes them are the colors that are used for the **active** and **focused** states.

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

Style options

background *color*

bordercolor *color*

borderwidth *amount*

darkcolor *color*

fieldbackground *color*

foreground *color*

font *font*

lightcolor *color*

padding *padding*

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

selectbackground *color*

selectborderwidth *amount*

selectforeground *color*

2.4.5.4 Create a custom style

Change the **font** and **font-size** on all entry widgets

```
Style.configure('TEntry', font=('Helvetica', 12))
```

Change the **foreground color** when in different states

```
Style.map('TEntry', foreground=[
    ('disabled', 'gray'),
    ('focus !disabled', 'green'),
    ('hover !disabled', 'yellow')])
```

Subclass an existing style to create a new one, using the pattern 'newstyle.OldStyle'

```
Style.configure('custom.TEntry', background='green', foreground='white', font=('Helvetica
↪', 24))
```

Use a custom style

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

2.4.5.5 References

- https://www.pythontutorial.net/tkinter/tkinter-entry/
- https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/ttk-Entry.html
- https://tcl.tk/man/tcl8.6/TkCmd/ttk_entry.htm

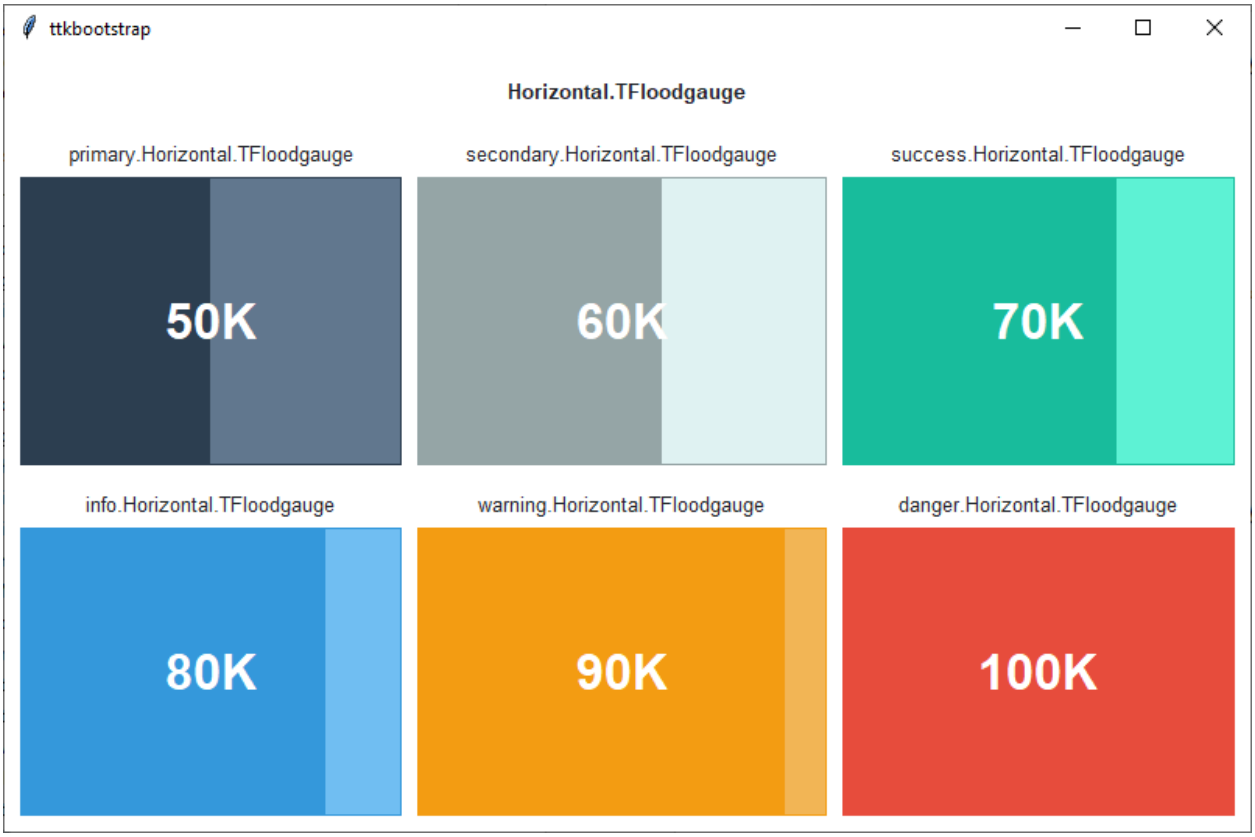
2.4.6 Floodgauge

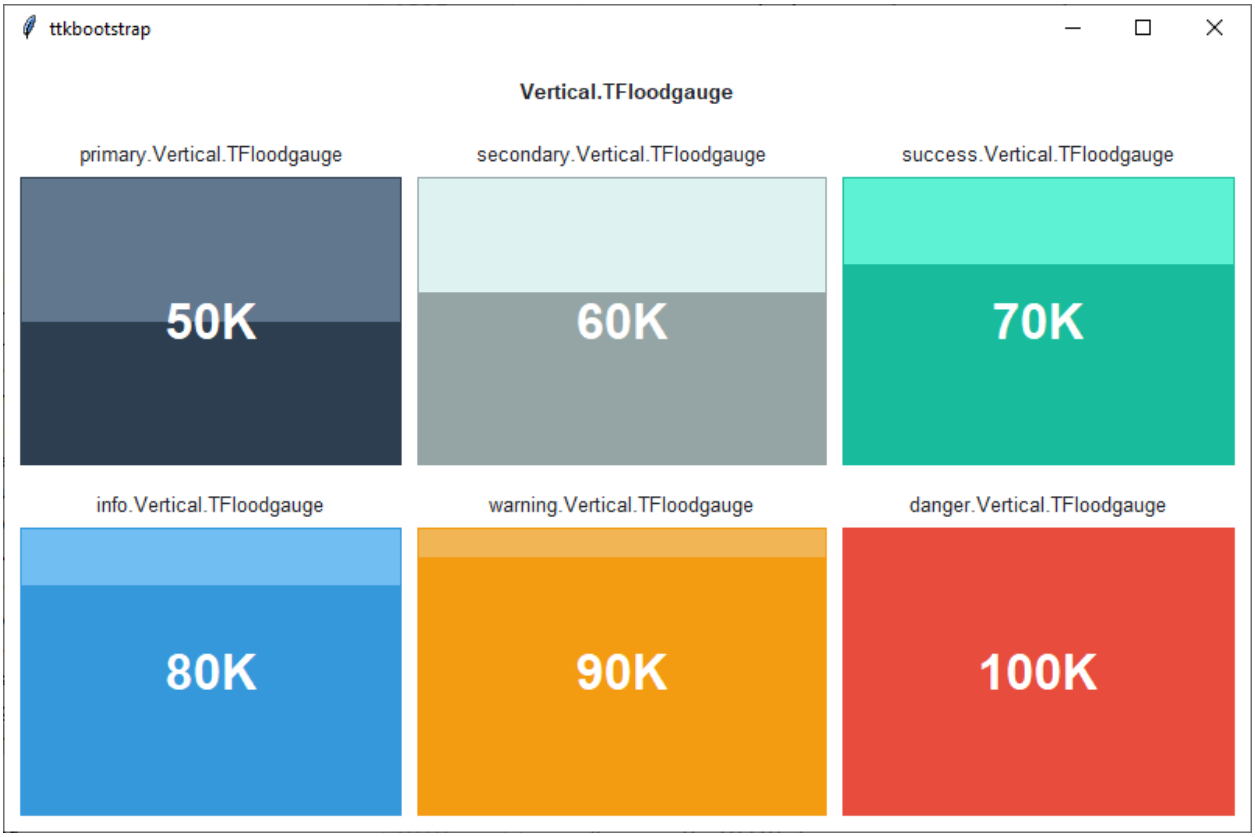
A `Floodgauge` widget is a custom **ttkbootstrap** widget that shows the status of a long-running operation with an optional text indicator. Similar to the `ttk.Progressbar`, this widget can operate in two modes: **determinate** mode shows the amount completed relative to the total amount of work to be done, and **indeterminate** mode provides an animated display to let the user know that something is happening.

Note: This is a **style guide** for using ttkbootstrap styles. This guide will show you how to **apply visual styles** to change the look and feel of the widget. For more information on how to use the widget and what options are available, consult the *reference section on widgets*.

2.4.6.1 Overview

The `Floodgauge` includes the **Horizontal.TFloodgauge** and **Vertical.TFloodgauge** styles. These styles are further subclassed by each of the theme colors to produce the following color and style combinations (the *primary* color is the default for all floodgauge widgets:





2.4.6.2 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*.

Create a default **horizontal floodgauge**

```
Floodgauge(parent, value=75)
```

Create a default **vertical floodgauge**

```
Floodgauge(parent, value=75, orient='vertical')
```

Create a **success colored horizontal floodgauge**

```
Floodgauge(parent, value=75, style='success.Horizontal.TFloodgauge')
```

Create an **info colored vertical floodgauge**

```
Floodgauge(parent, value=75, style='info.Vertical.TFloodgauge', orient='vertical')
```

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*

2.4.6.4 Create a custom style

Change the **thickness** and **relief** of all floodgauges

```
Style.configure('TFloodgauge', thickness=20, pbarrelief='flat')
```

Subclass an existing style to create a new one, using the pattern 'newstyle.OldStyle'

```
Style.configure('custom.Horizontal.TFloodgauge', background='green', troughcolor='gray')
```

Use a custom style

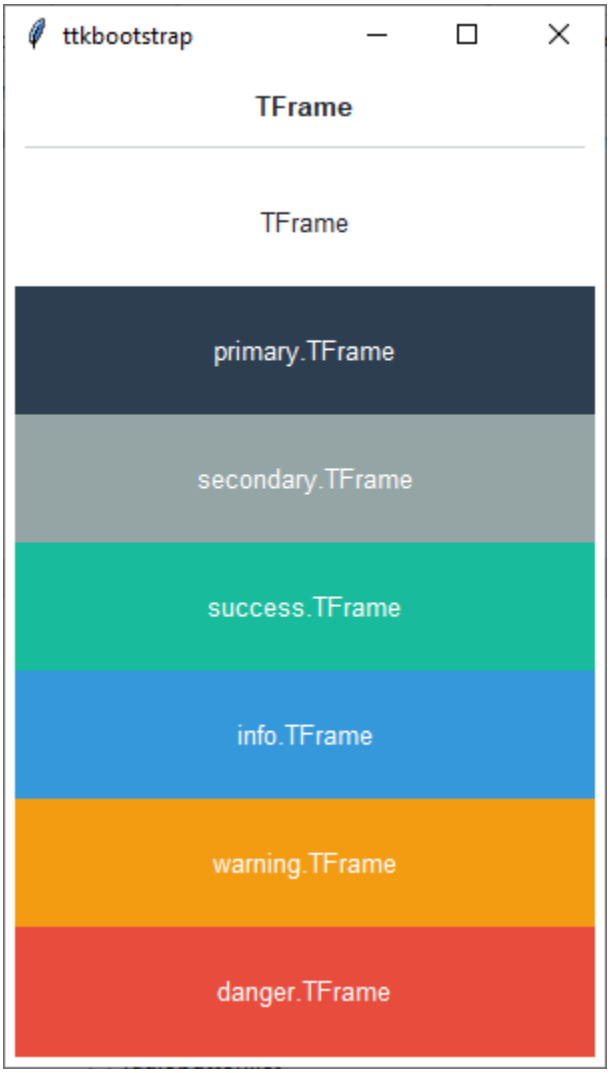
```
Floodgauge(parent, value=25, orient='horizontal', style='custom.Horizontal.TFloodgauge')
```

2.4.7 Frame

A `ttk.Frame` widget is a container, used to group other widgets together.

2.4.7.1 Overview

The `ttk.Frame` includes the **TFrame** class. This class is further subclassed by each of the theme colors to produce the following color and style combinations. The **TFrame** style is applied to all frame widgets by default and shares the same color as the theme background.



2.4.7.2 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 **frame**

```
ttk.Frame(parent)
```

Create an **'info' frame**

```
ttk.Frame(parent, style='info.TFrame')
```

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*

2.4.7.4 Create a custom style

Subclass an existing style to create a new one, using the pattern 'newstyle.OldStyle'

```
Style.configure('custom.TFrame', background='green', relief='sunken')
```

Use a custom style

```
ttk.Frame(parent, style='custom.TFrame')
```

2.4.7.5 Tips & tricks

If you use a themed **Frame** widget, then you will likely want to use a **Label** widget with an *Inverse.TLabel* style. This will create the effect that is presented in the *Overview*, with the the label background matching the background color of its parent.

```
frm = ttk.Frame(parent, style='danger.TFrame')
lbl = ttk.Label(f, text='Hello world!', style='danger.Inverse.TLabel')
```

2.4.7.6 References

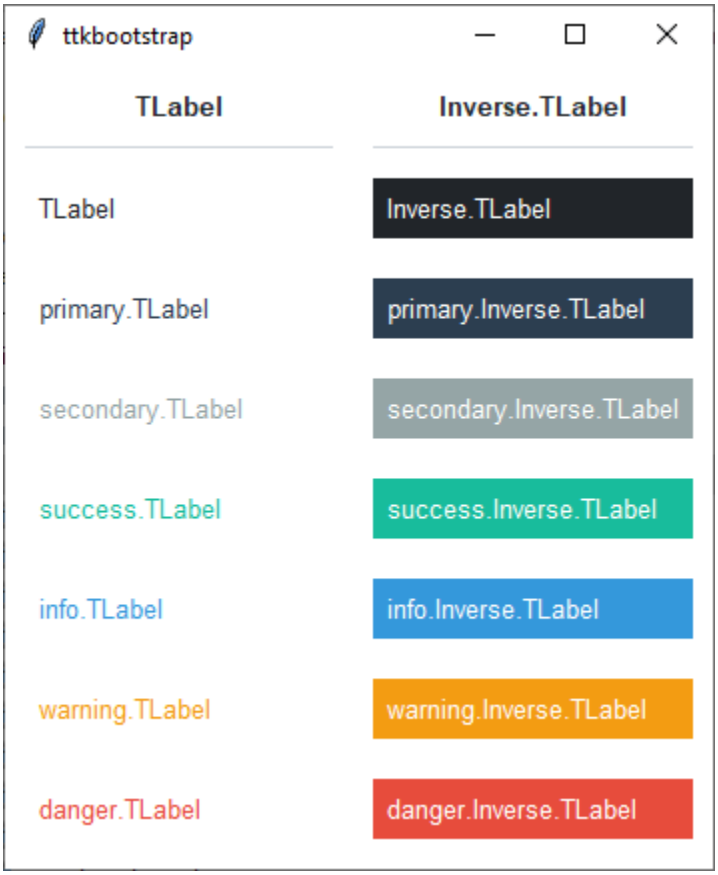
- https://www.pythontutorial.net/tkinter/tkinter-frame/
- https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/ttk-Frame.html
- https://tcl.tk/man/tcl8.6/TkCmd/ttk_frame.htm

2.4.8 Label

A `ttk.Label` widget displays a textual label and/or image. The label may be linked to a tkinter variable to automatically change the displayed text.

2.4.8.1 Overview

The `ttk.Label` includes the **TLabel** and **Inverse.TLabel** style classes. The **TLabel** style is applied to all labels by default and uses the theme's *inputfg* color for the foreground and the *background* color for the background. Other styles must be specified with the `style` option. These two primary styles are further subclassed by each of the theme colors to produce the following color and style combinations:



The theme colors can be *inverted* by using the **Inverse.TLabel** style, which causes the *background* and *foreground* colors to reverse.

2.4.8.2 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 **label**

```
ttk.Label(parent, text='python is great')
```

Create a default **inverse label**

```
ttk.Label(parent, text='python is great', style='Inverse.TLabel')
```

Create an **'info' label**

```
ttk.Label(parent, text='python is great', style='info.TLabel')
```

Create a **'warning' inverse label**

```
ttk.Label(parent, text="python is great", style='warning.Inverse.TLabel')
```

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

Style options

anchor *e, w, center*

background *color*

bordercolor *color*

compound *top, bottom, left, right*

darkcolor *color*

embossed *amount*

foreground *color*

font *font*

justify *left, right, center*

lightcolor *color*

padding *padding*

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

width *amount*

2.4.8.4 Create a custom style

Change the **font** and **font-size** on all labels

```
Style.configure('TLabel', font=('Helvetica', 12))
```

Subclass an existing style to create a new one, using the pattern 'newstyle.OldStyle'

```
Style.configure('custom.TLabel', background='red', foreground='white', font=('Helvetica',
↪ 24))
```

Use a custom style

```
ttk.Label(parent, text='what a great label', style='custom.TLabel')
```

2.4.8.5 Tips & tricks

You can apply a **TButton** style to a label to inherit the colors and hover effects of the button.

2.4.8.6 References

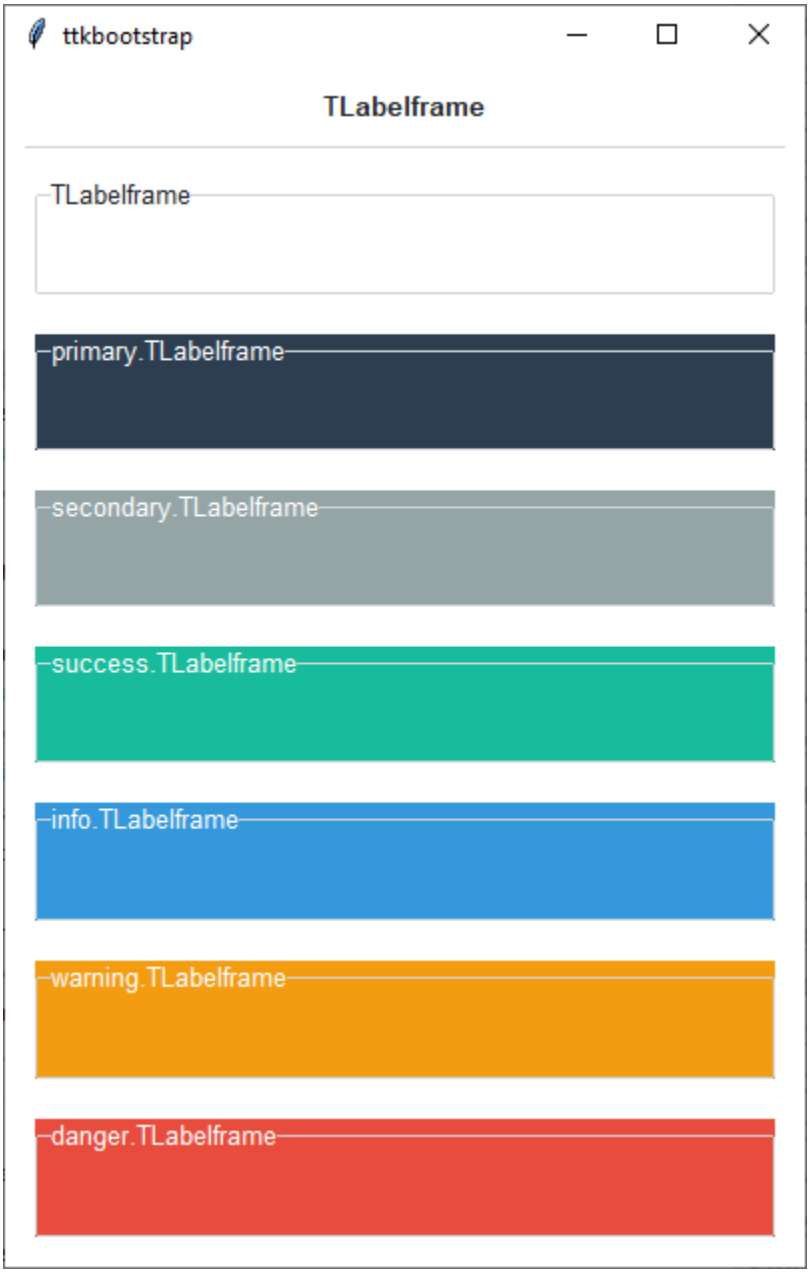
- https://www.pythontutorial.net/tkinter/tkinter-label/
- https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/ttk-Label.html
- https://tcl.tk/man/tcl8.6/TkCmd/ttk_label.htm

2.4.9 Labelframe

A `ttk.Labelframe` widget is a container used to group other widgets together. It has an optional label, which may be a plain text string or another widget.

2.4.9.1 Overview

The `ttk.Labelframe` includes the **TLabelframe** style class. The **TLabelframe** style is applied to all Labelframes by default and uses the theme *border* color for the frame and *background* color for the background. Other styles must be specified with the `style` option. This style is further subclassed by each of the theme colors to produce the following color and style combinations:



2.4.9.2 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 **labelframe**

```
ttk.Labelframe(parent, text='My widgets')
```

Create an **'info' labelframe**

```
ttk.Labelframe(parent, text='My widgets', style='info.TLabelframe')
```

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

Style options

anchor *e, w, center*

background *color*

bordercolor *color*

borderwidth *amount*

darkcolor *color*

labelmargins *amount*

labeloutside *boolean*

lightcolor *color*

padding *padding*

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

width *amount*

TLabelframe.Label styling options include:

background *color*

darkcolor *color*

font *font*

foreground *color*

lightcolor *color*

2.4.9.4 References

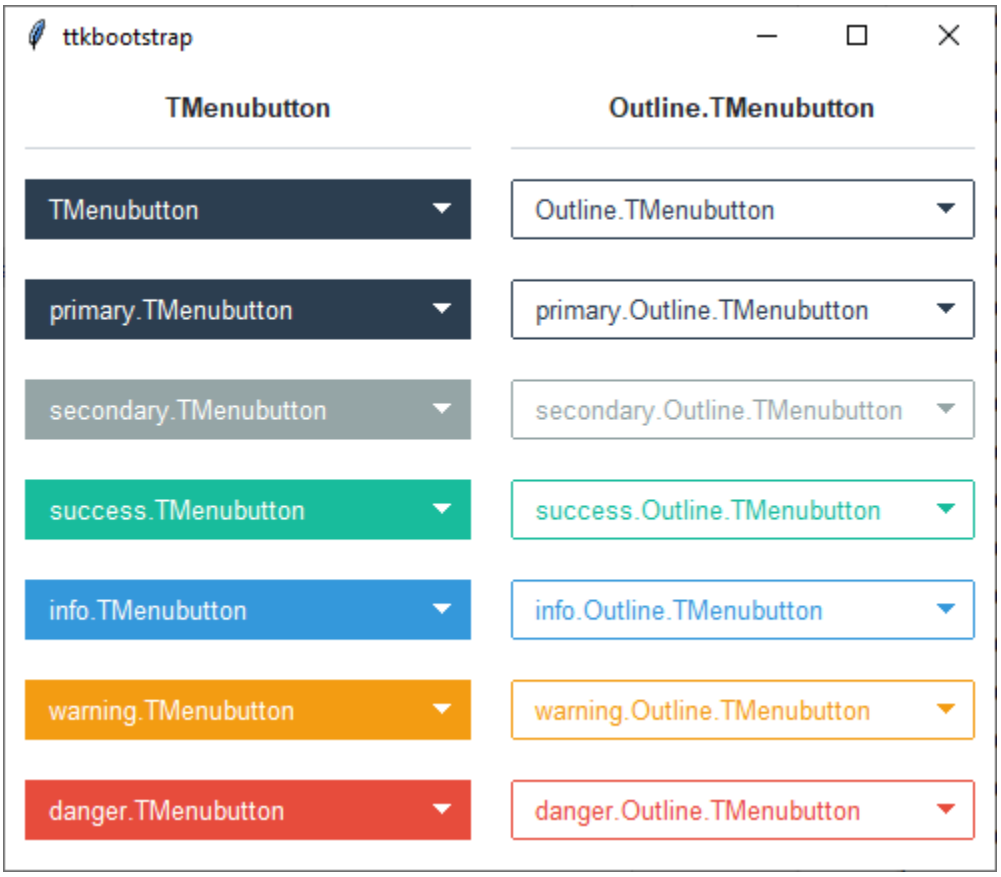
- https://www.pythontutorial.net/tkinter/tkinter-labelframe/
- https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/ttk-LabelFrame.html
- https://tcl.tk/man/tcl8.6/TkCmd/ttk_labelframe.htm
- https://tkdocs.com/tutorial/complex.html#labelframe

2.4.10 Menubutton

A `ttk.Menubutton` widget displays a textual label and/or image, and displays a menu when pressed.

2.4.10.1 Overview

The `ttk.Menubutton` includes the **TMenubutton** and **Outline.TMenubutton** style classes. The **TMenubutton** style is applied to all Menubuttons by default and uses the theme *primary* color as the background. These two primary styles are further subclassed by each of the theme colors to produce the following color and style combinations:



The **Outline.TMenubutton** style has a solid fill color (matching the regular *TMenubutton*) when hovered or pressed.

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

Style options

arrowsize *amount*

arrowcolor *color*

arrowpadding *amount*

background *color*

compound *top, bottom, left, right*

bordercolor *color*

borderwidth *amount*

darkcolor *color*

focusthickness *amount*

focuscolor *color*

foreground *color*

font *font*

lightcolor *color*

padding *padding*

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

2.4.10.4 Create a custom style

Change the **font** and **font-size** on all menubuttons

```
Style.configure('TMenubutton', font=('Helvetica', 12))
```

Subclass an existing style to create a new one, using the pattern 'newstyle.OldStyle'

```
Style.configure('custom.TMenubutton', background='red', foreground='white', font=(
↪'Helvetica', 24))
```

Use a custom style

```
ttk.Menubutton(parent, text='My widgets', style='custom.TMenubutton')
```

Note: The *Menu* object cannot be configured with `Style`. Instead, use the tk option database.

- `tk.option_add('*Menu.tearoff', 0)`
- `tk.option_add('*Menu.foreground', 'white')`
- `tk.option_add('*Menu.selectColor', 'blue')`
- `tk.option_add('*Menu.font', 'Helvetica 12')`
- `tk.option_add('*Menu.background', 'black')`
- `tk.option_add('*Menu.activeBackground', 'yellow')`
- `tk.option_add('*Menu.activeForegorund', 'blue')`

2.4.10.5 References

- https://www.pythontutorial.net/tkinter/tkinter-menubutton/
- https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/ttk-Menubutton.html
- https://tcl.tk/man/tcl8.6/TkCmd/ttk_menubutton.htm

2.4.11 Meter

The `Meter` is a custom **ttkbootstrap** widget that can be used to show progress of long-running operations or the amount of work completed. It can also be used as a *Dial* when *interactive* mode is set to True.

Note: This is a **style guide** for using ttkbootstrap styles. This guide will show you how to **apply visual styles** to change the look and feel of the widget. For more information on how to use the widget and what options are available, consult the *reference section on widgets*.

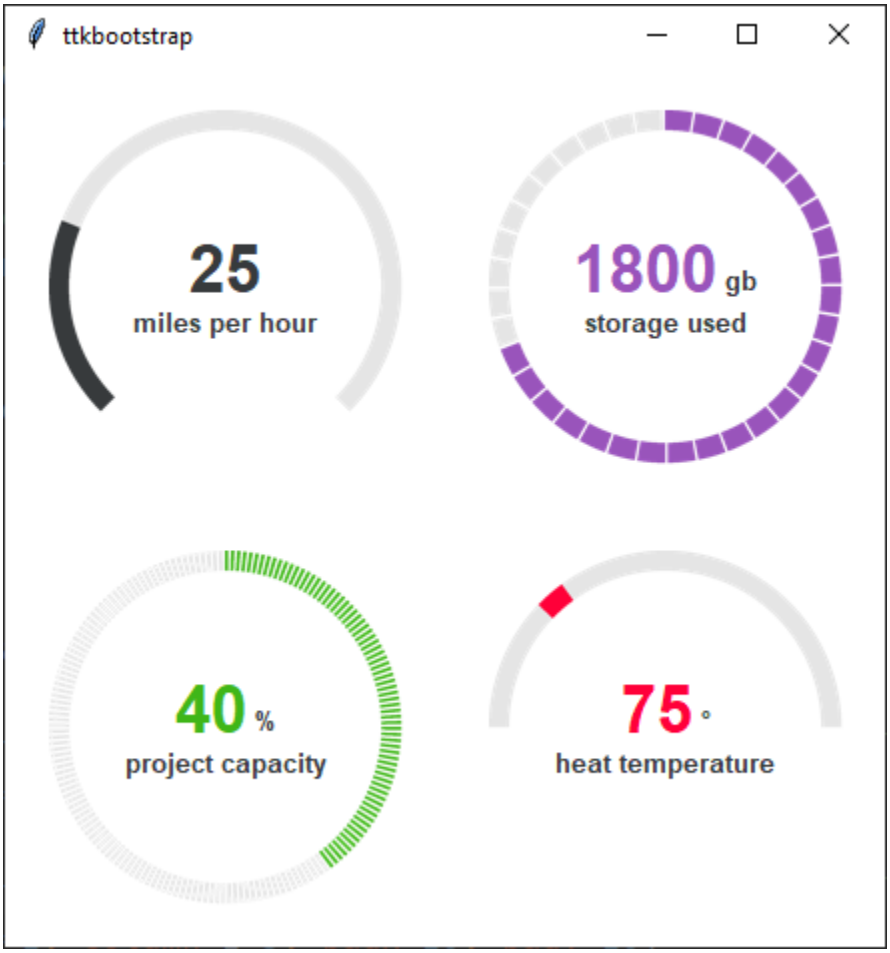
2.4.11.1 Overview

This widget is very flexible. The `metertype` parameter has two stock settings: *full* and *semi*, which shows a full circle and a semi-circle respectively. Customize the arc of the circle with the `arcrange` and `arcoffset` parameters. This moves the starting position of the arc and can also be used to make the arc longer or shorter.

The meter color is set with `meterstyle` and uses the *TMeter* style class. This also colors the center text. There is an optional supplementary label *below* the center text that can be styled with the `labelstyle` parameter, which excepts a *TLabel* style class. This setting also formats the text added with `textappend` and `textprepend`.

The **primary.TMeter** style is applied by default. The base style is further subclassed by each of the theme colors to produce the following color and style combinations:

The examples below demonstrate how flexible this widget can be. You can see the code for these in the *Cookbook*.



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**

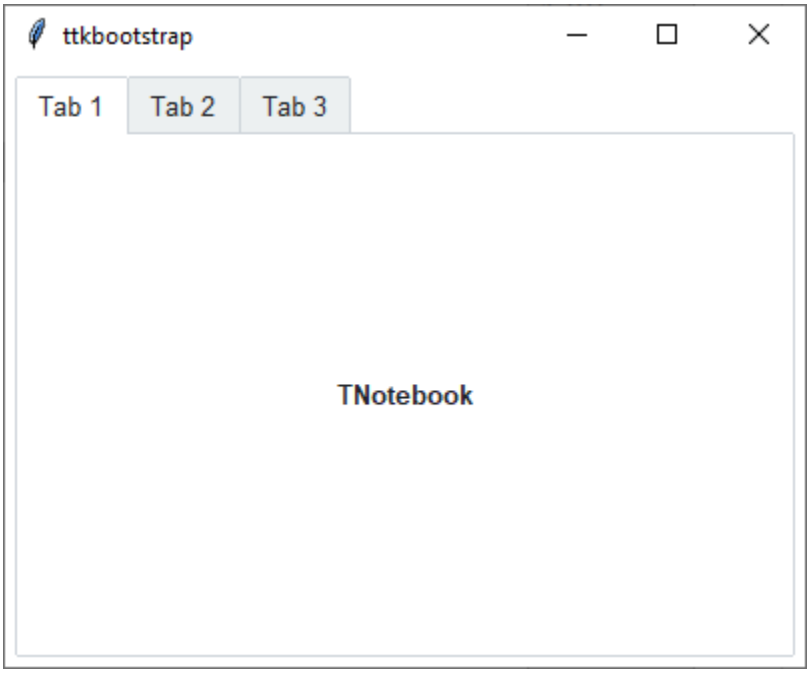
```
Meter(parent, amountused=25, labeltext='miles per hour', meterstyle='info.TLabel',␣
↪labelstyle='success.TLabel')
```

2.4.12 Notebook

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

2.4.12.1 Overview

The `ttk.Notebook` includes the **TNotebook** style class. Presently, this style contains default settings for light and dark themes, but no other styles are included. This may change in the future. See the *Create a custom style* section to learn how to customize and create a notebook style.



2.4.12.2 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 and use a **notebook**

```
# 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

Style options

background *color*

bordercolor *color*

darkcolor *color*

foreground *color*

lightcolor *color*:

padding *padding*

tabmargins *padding*

tabposition *n, s, e, w, ne, en, nw, wn, se, es, sw, ws*

TNotebook.Tab styling options include:

background *color*

bordercolor *color*

compound *left, right, top, button*

expand *padding*

font *font*

foreground *color*

padding *padding*

2.4.12.4 Create a custom style

Subclass an existing style to create a new one, using the pattern 'newstyle.OldStyle'. In this example, the tab position is set to the *southwest* corner of the notebook… by default it is on the *northwest* corner.

```
# set the tabs on the sw corner of the notebook
Style.configure('custom.TNotebook', tabposition='sw')
```

Use a custom style

```
nb = ttk.Notebook(parent, style='custom.TNotebook')
```

2.4.12.5 References

- https://www.pythontutorial.net/tkinter/tkinter-notebook/
- https://docs.python.org/3/library/tkinter.ttk.html#ttk-notebook
- https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/ttk-Notebook.html
- https://tcl.tk/man/tcl8.6/TkCmd/ttk_notebook.htm

2.4.13 PanedWindow

A `ttk.PanedWindow` widget displays a number of subwindows, stacked either vertically or horizontally. The user may adjust the relative sizes of the subwindows by dragging the sash between panes.

2.4.13.1 Overview

The `ttk.PanedWindow` includes the **TPanedwindow** style class. Presently, this style contains default settings for light and dark themes, but no other styles are included. This may change in the future. See the *Create a custom style* section to learn how to customize and create a paned window style.

2.4.13.2 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 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

Style options

TPanedwindow styling options:

background *color*

Sash styling options:

background *color*

bordercolor *color*

gripcount *count*

handlepad *amount*

handlesize *amount*

lightcolor *color*

sashpad *amount*

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

sashthickness *amount*

2.4.13.4 Create a custom style

Change the **relief** on all paned window sashes, and change the **gripcount**

```
Style.configure('Sash', relief='flat', gripcount=15)
```

Subclass an existing style to create a new one, using the pattern 'newstyle.OldStyle'

```
Style.configure('custom.TPanedwindow', background='red')
```

Use a custom style

```
ttk.PanedWindow(parent, style='custom.TPanedwindow')
```

2.4.13.5 References

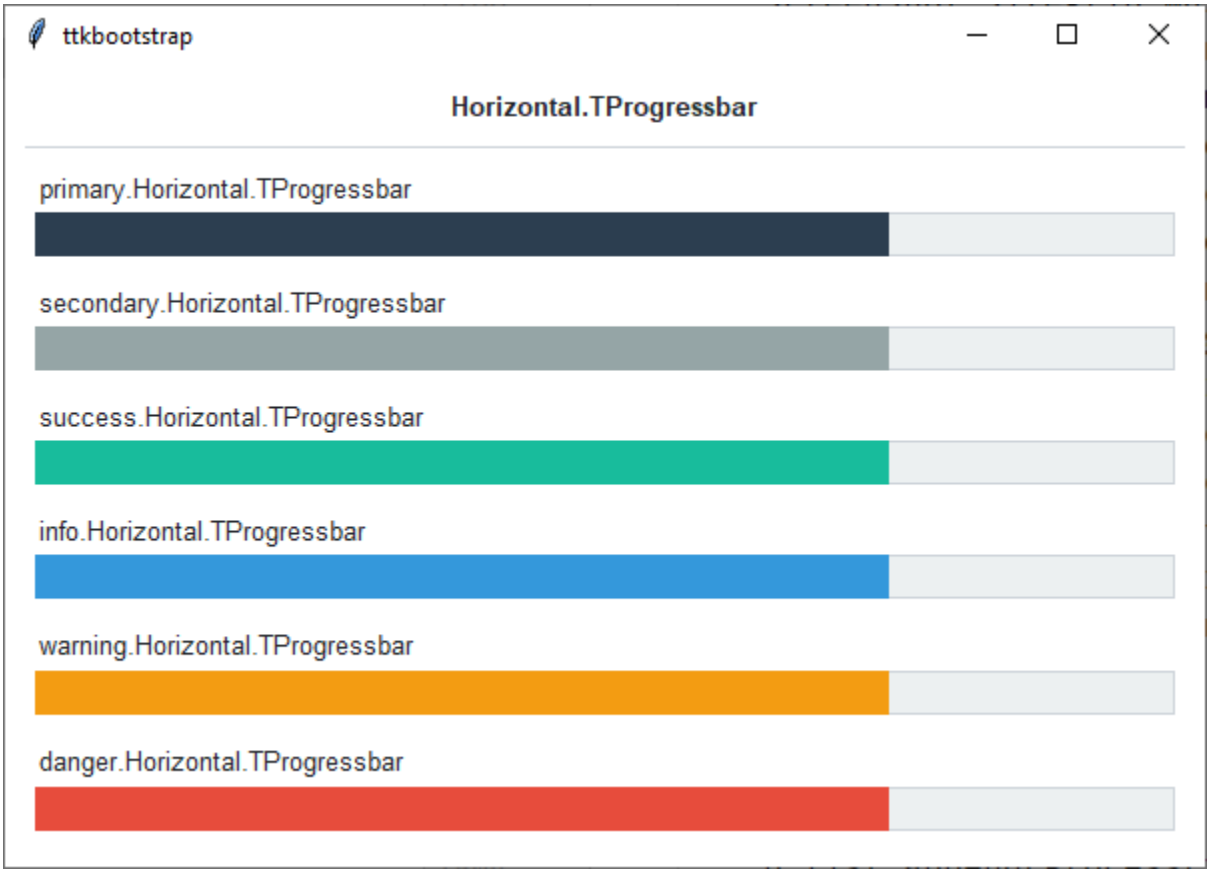
- https://www.pythontutorial.net/tkinter/tkinter-panedwindow/
- https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/ttk-PanedWindow.html
- https://tcl.tk/man/tcl8.6/TkCmd/ttk_panedwindow.htm

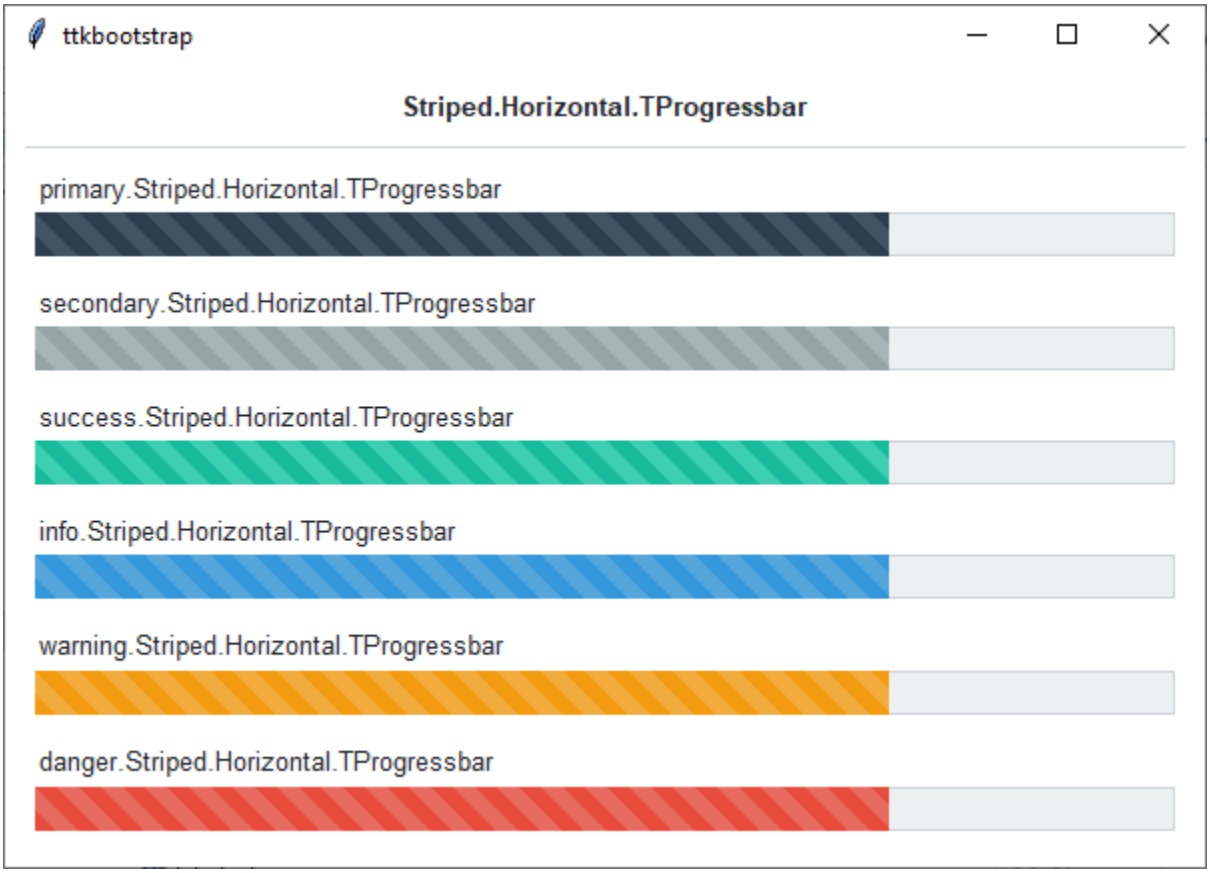
2.4.14 Progressbar

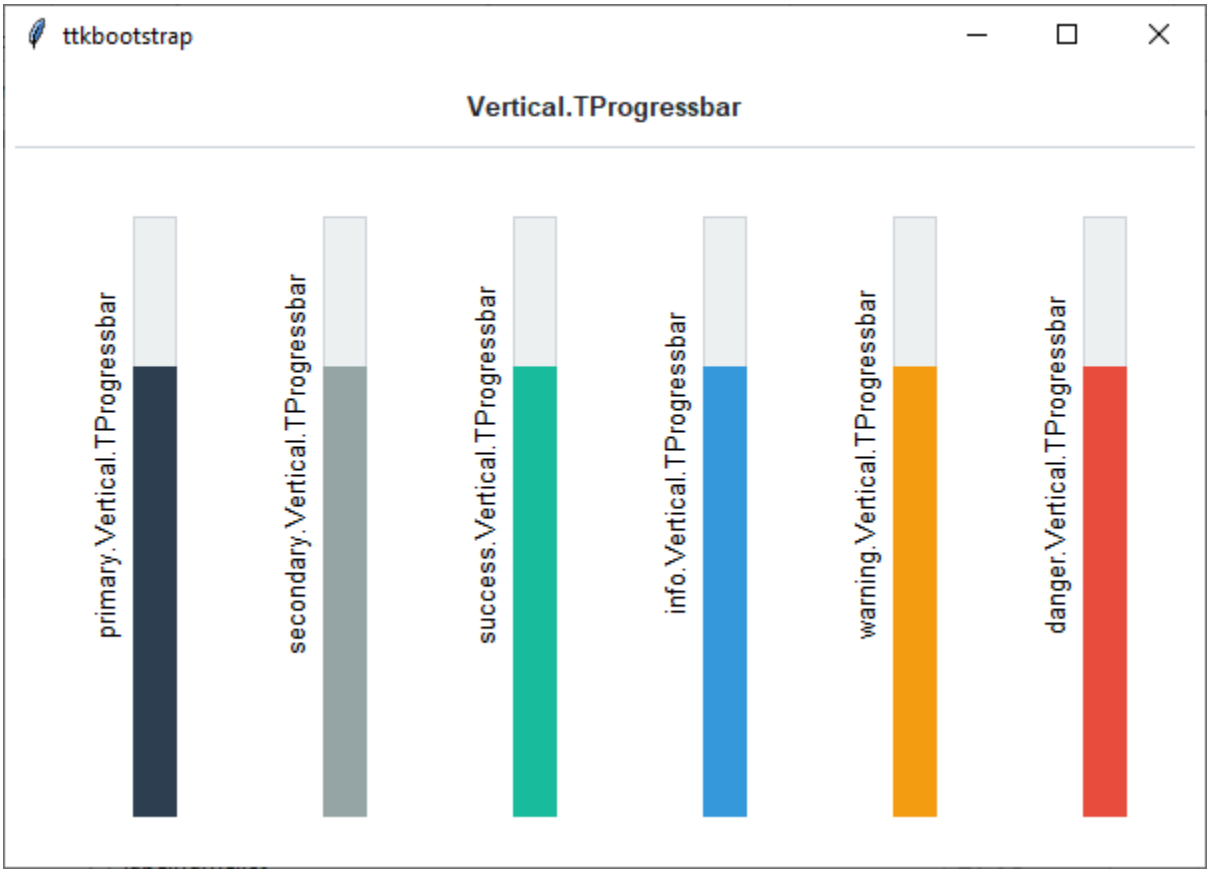
A `ttk.Progressbar` widget shows the status of a long-running operation. They can operate in two modes: determinate mode shows the amount completed relative to the total amount of work to be done, and indeterminate mode provides an animated display to let the user know that something is happening.

2.4.14.1 Overview

The `ttk.Progressbar` includes the **Horizontal.TProgressbar**, **Vertical.TProgressbar**, and **Striped.Horizontal.TProgressbar** styles. These styles are further subclassed by each of the theme colors to produce the following color and style combinations (the *primary* color is the default for all progress bars:




ttkbootstrap
Striped.Horizontal.TProgressbar
primary.Striped.Horizontal.TProgressbar
secondary.Striped.Horizontal.TProgressbar
success.Striped.Horizontal.TProgressbar
info.Striped.Horizontal.TProgressbar
warning.Striped.Horizontal.TProgressbar
danger.Striped.Horizontal.TProgressbar



2.4.14.2 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 progressbar**

```
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

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*

2.4.14.4 Create a custom style

Change the **thickness** and **relief** of all progressbars

```
Style.configure('TProgressbar', thickness=20, pbarrelief='flat')
```

Subclass an existing style to create a new one, using the pattern 'newstyle.OldStyle'

```
Style.configure('custom.Horizontal.TProgressbar', background='green', troughcolor='gray')
```

Use a custom style

```
ttk.Progressbar(parent, value=25, orient='horizontal', style='custom.Horizontal.
↪TProgressbar')
```

2.4.14.5 References

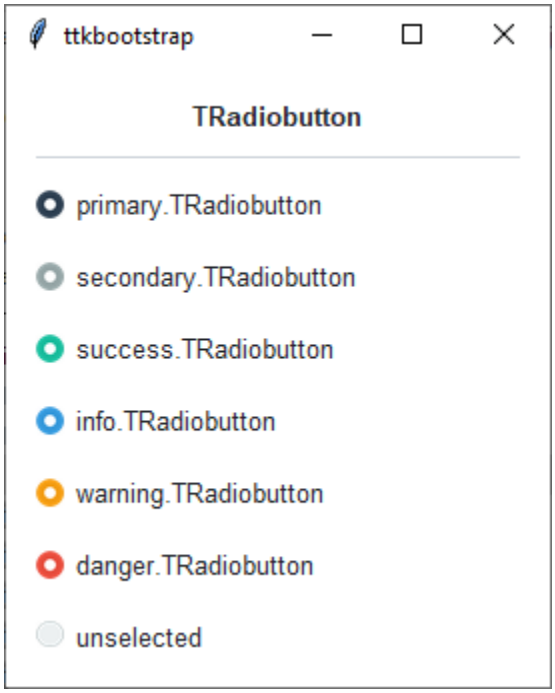
- https://docs.python.org/3/library/tkinter.ttk.html#ttk-progressbar
- https://www.pythontutorial.net/tkinter/tkinter-progressbar/
- https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/ttk-Progressbar.html
- https://tcl.tk/man/tcl8.6/TkCmd/ttk_progressbar.htm

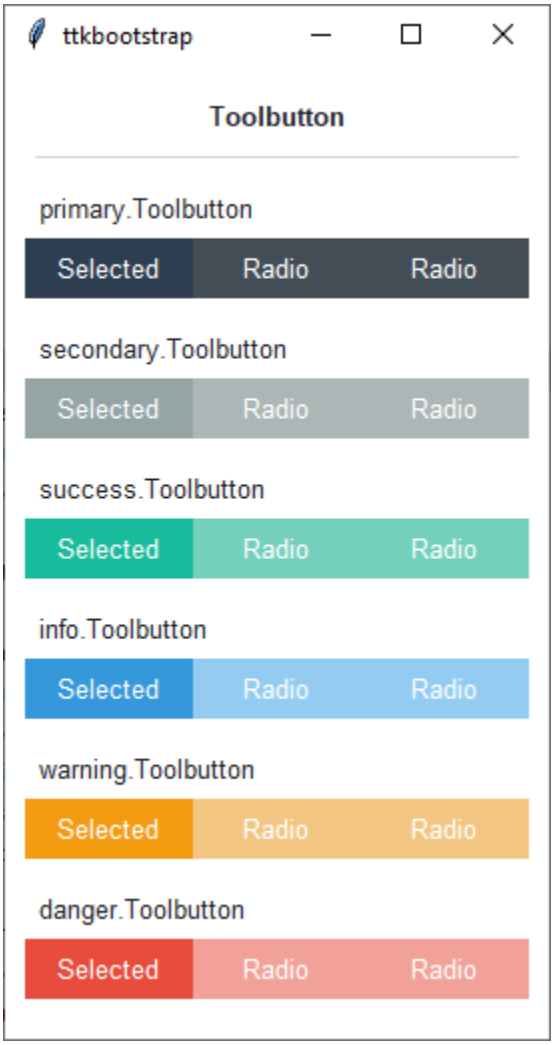
2.4.15 Radiobutton

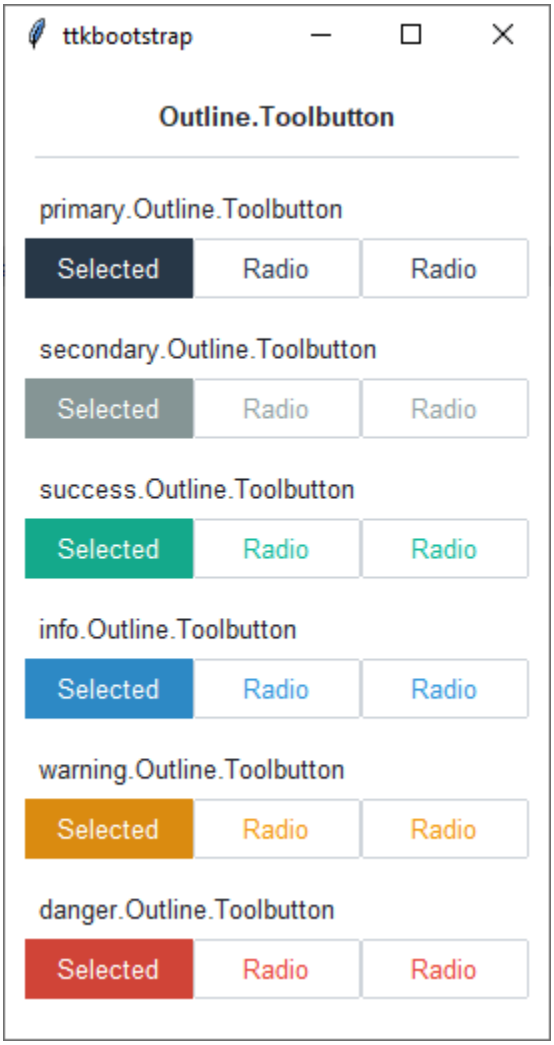
A `ttk.Radiobutton` widget is used in groups to show or change a set of mutually-exclusive options. Radiobuttons are linked to a tkinter variable, and have an associated value; when a radiobutton is clicked, it sets the variable to its associated value.

2.4.15.1 Overview

The `ttk.Radiobutton` includes the **TRadiobutton**, **ToolButton**, and **Outline.Toolbutton** styles. These styles are further subclassed by each of the theme colors to produce the following color and style combinations:




ttkbootstrap
Toolbutton
primary.Toolbutton
Selected
Radio
Radio
secondary.Toolbutton
Selected
Radio
Radio
success.Toolbutton
Selected
Radio
Radio
info.Toolbutton
Selected
Radio
Radio
warning.Toolbutton
Selected
Radio
Radio
danger.Toolbutton
Selected
Radio
Radio



2.4.15.2 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 **radiobutton**

```
ttk.Radiobutton(parent, text='option 1')
```

Create a default **toolbutton**

```
ttk.Radiobutton(parent, text='option 2', style='Toolbutton')
```

Create a default **outline toolbutton**

```
ttk.Radiobutton(parent, text='option 3', style='Outline.Toolbutton')
```

Create an **'info' radiobutton**

```
ttk.Radiobutton(parent, text='option 4', style='info.TRadiobutton')
```

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

Style options

background *color*

compound *compound*

foreground *foreground*

focuscolor *color*

focusthickness *amount*

font *font*

padding *padding*

2.4.15.4 Create a custom style

Change the **font** and **font-size** on all radiobuttons

```
Style.configure('TRadiobutton', font=('Helvetica', 12))
```

Change the **foreground color** when the radiobutton is **selected**

```
Style.map('TRadiobutton', foreground=[
    ('disabled', 'white'),
    ('selected', 'yellow'),
    ('!selected', 'gray')])
```

Subclass an existing style to create a new one, using the pattern 'newstyle.OldStyle'

```
Style.configure('custom.TRadiobutton', foreground='white', font=('Helvetica', 24))
```

Use a custom style

```
ttk.Radiobutton(parent, text='option 1', style='custom.TRadiobutton')
```

2.4.15.5 References

- https://www.pythontutorial.net/tkinter/tkinter-radio-button/
- https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/ttk-Radiobutton.html
- https://tcl.tk/man/tcl8.6/TkCmd/ttk_radiobutton.htm

2.4.16 Slider

A `ttk.Scale` widget is typically used to control the numeric value of a linked variable that varies uniformly over some range. A scale displays a slider that can be moved along over a trough, with the relative position of the slider over the trough indicating the value of the variable.

2.4.16.1 Overview

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

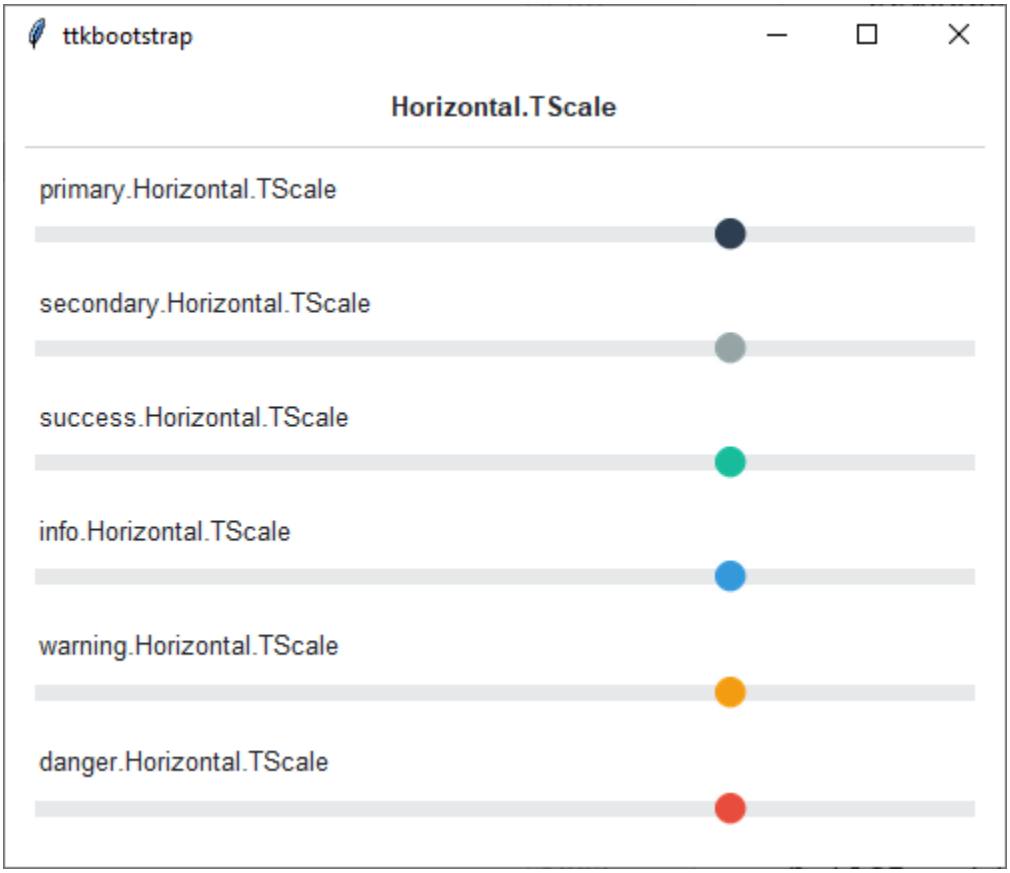
The examples below demonstrate how to *use a style* to create a widget. To learn more about how to *use the widget in ttk*, check out the *References* section for links to documentation and tutorials on this widget.

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')
```



2.4.16.3 Configuration

Use the following classes, states, and options when configuring or modifying a new ttk progressbar style. TTK Bootstrap uses an image layout for this widget, so styling options will be limited, and not all options below will be available for ttk bootstrap themes. See the python style documentation for more information on creating a style.

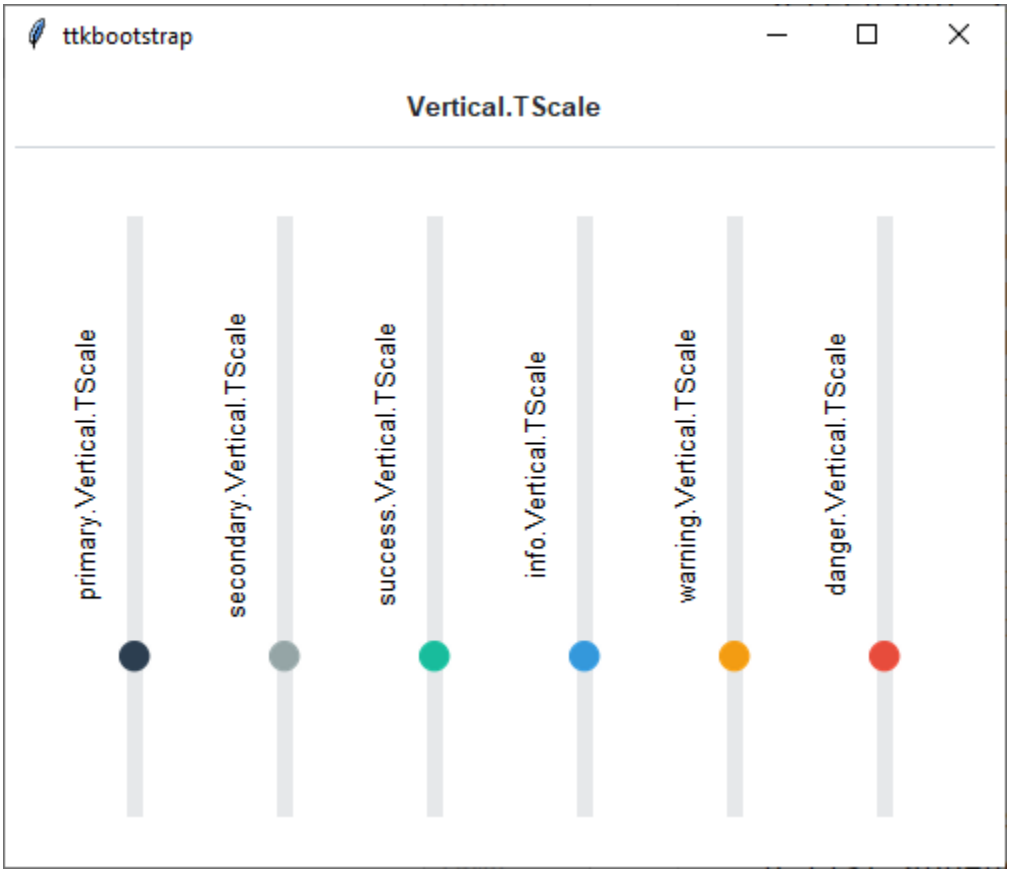
Create a new theme using TTK Creator if you want to change the default color scheme.

Class names

- Horizontal.TScale
- Vertical.TScale

Dynamic states

- Active



Style options

background *color*

borderwidth *amount*

darkcolor *color*

groovewidth *amount*

lightcolor *color*

sliderwidth *amount*

troughcolor *color*

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

2.4.16.4 References

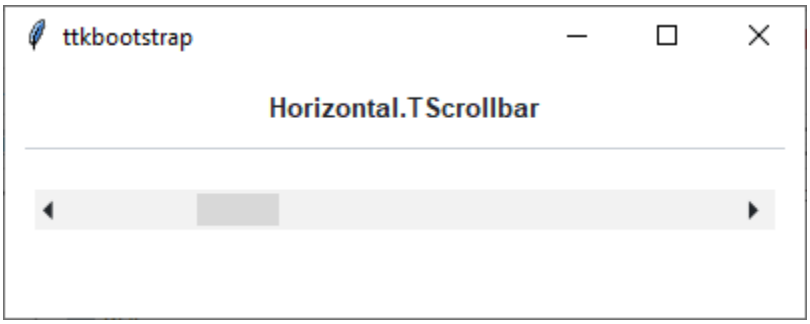
- https://www.pythontutorial.net/tkinter/tkinter-slider/
- https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/ttk-Scale.html
- https://tcl.tk/man/tcl8.6/TkCmd/ttk_scale.htm

2.4.17 Scrollbar

`ttk.Scrollbar` widgets are typically linked to an associated window that displays a document of some sort, such as a file being edited or a drawing. A scrollbar displays a thumb in the middle portion of the scrollbar, whose position and size provides information about the portion of the document visible in the associated window. The thumb may be dragged by the user to control the visible region. Depending on the theme, two or more arrow buttons may also be present; these are used to scroll the visible region in discrete units.

2.4.17.1 Overview

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

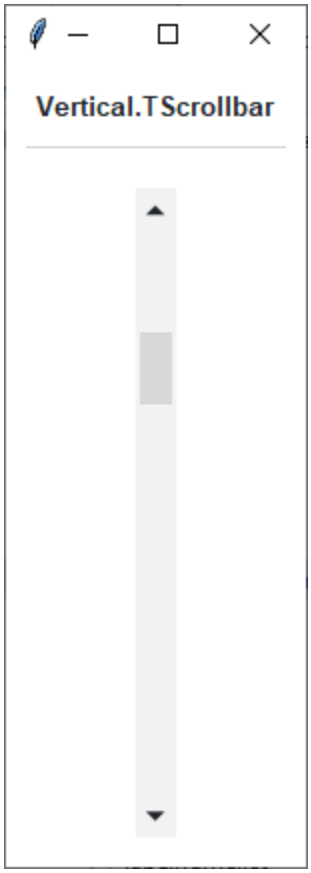
The examples below demonstrate how to *use a style* to create a widget. To learn more about how to *use the widget in ttk*, check out the *References* section for links to documentation and tutorials on this widget.

Create a default **horizontal scrollbar**

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

Create a default **vertical scrollbar**

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



2.4.17.3 Configuration

Use the following classes, states, and options when configuring or modifying a new ttk scrollbar style. TTK Bootstrap uses an image layout for parts of this widget (the arrows), so styling options will not affect these elements. However, if you choose to create your own scrollbar layout and style, you may use whatever style options are available for your custom 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.TScrollbar
- Vertical.TScrollbar

Dynamic states

- active
- disabled

Style options

arrowcolor *color*

arrowsize *amount*

background *color*

bordercolor *color*

gripcount *amount*

groovewidth *amount*

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

troughborderwidth *amount*

troughcolor *color*

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

width *amount*

2.4.17.4 Create a custom style

Change the **thickness** and **background** of all scrollbars

```
Style.configure('TScrollbar', width=30, background='black')
```

Subclass an existing style to create a new one, using the pattern 'newstyle.OldStyle'

```
Style.configure('custom.Horizontal.TScrollbar', background='black', troughcolor='white',␣
↪arrowcolor='white')
```

Use a custom style

```
ttk.Scrollbar(parent, orient='horizontal', style='custom.Horizontal.TScrollbar')
```

2.4.17.5 References

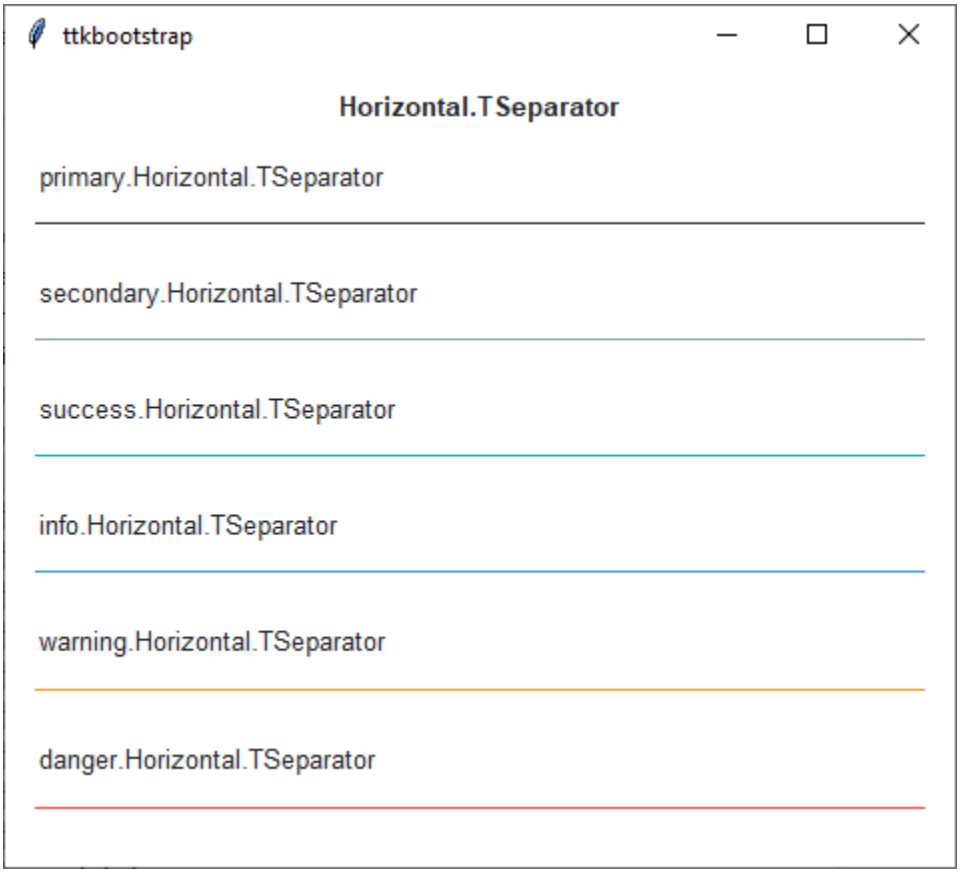
- https://www.pythontutorial.net/tkinter/tkinter-scrollbar/
- https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/ttk-Scrollbar.html
- https://www.tcl.tk/man/tcl8.6/TkCmd/ttk_scrollbar.htm

2.4.18 Separator

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

2.4.18.1 Overview

The `ttk.Separator` includes the **Horizontal.TSeparator** and **Vertical.TSeparator** style classes. These styles are applied by default to *horizontal* and *vertical* orientations. These styles are further subclassed by each of the theme colors to produce the following color and style combinations:



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.

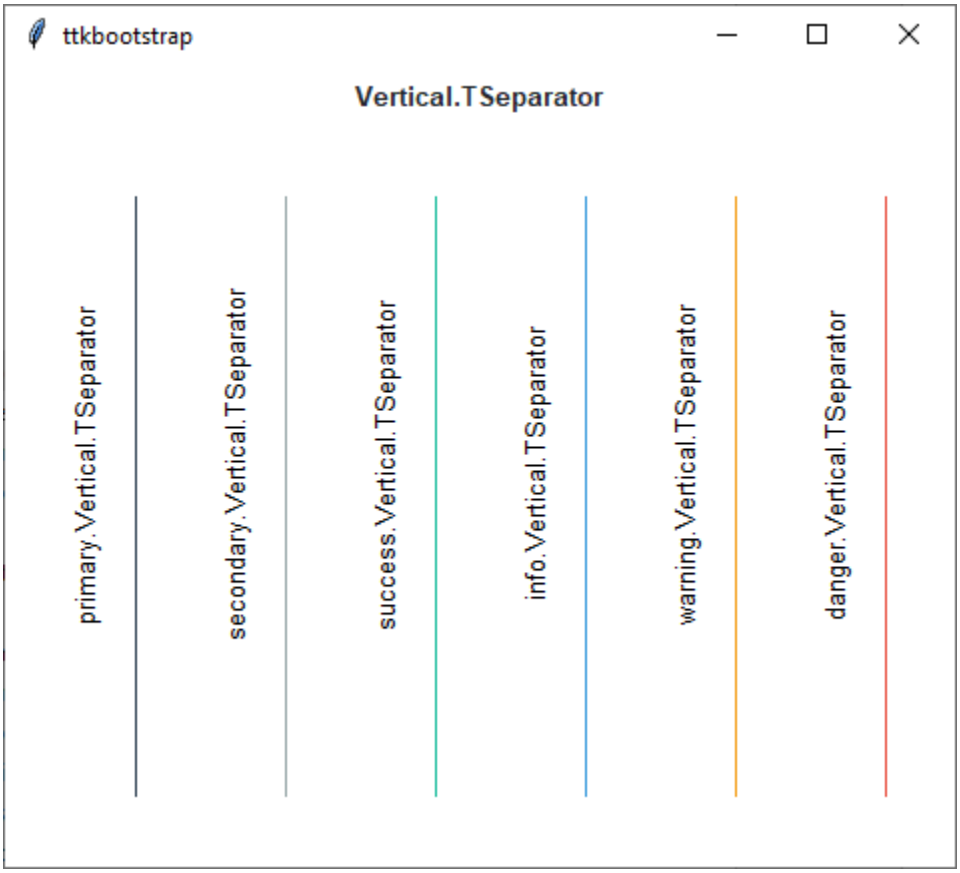
Create a default **horizontal separator**

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

Create a default **vertical separator**

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

Create an **info vertical separator**




```
ttk.Separator(parent, orient='vertical', style='info.Vertical.TSeparator')
```

2.4.18.3 Configuration

Use the following classes, states, and options when configuring or modifying a new ttk separator style. TTK Bootstrap uses an image layout for this widget, so it is not possible to create a custom style without building a new layout. However, if you decide to build your own layout, you are free to use the styling options below. 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.TSeparator
- Vertical.TSeparator

Style options

background *color*

2.4.18.4 References

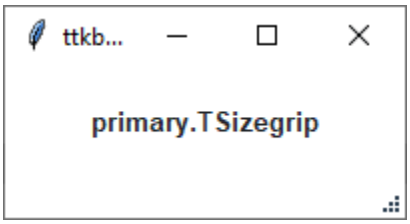
- https://www.pythontutorial.net/tkinter/tkinter-separator/
- https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/ttk-Separator.html
- https://www.tcl.tk/man/tcl8.6/TkCmd/ttk_separator.htm

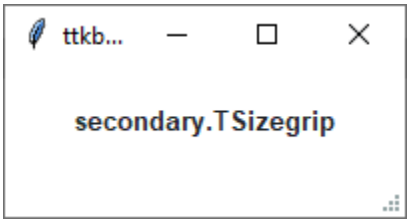
2.4.19 Sizegrip

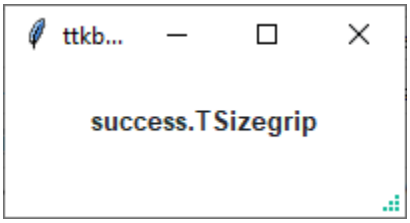
A `ttk.Sizegrip widget` (also known as a grow box) allows the user to resize the containing toplevel window by pressing and dragging the grip.

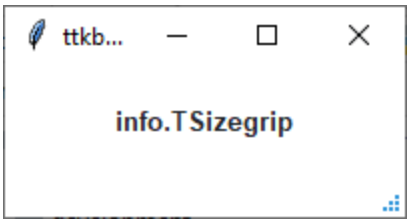
2.4.19.1 Overview

The `ttk.Sizegrip` includes the **TSizegrip** style class. By default, the color of the sizegrip is the *border* color for light themes and the *inputfg* color for dark themes. This is further subclassed by each of the theme colors to produce the following color and style combinations:









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')
```

2.4.19.3 Configuration

Use the following classes, states, and options when configuring or modifying a new ttk sizegrip style. TTK Bootstrap uses an image layout for this widget, so styling options will not be available for TTK Bootstrap themes. 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

- TSizegrip

Style options

- background *color*

2.4.19.4 References

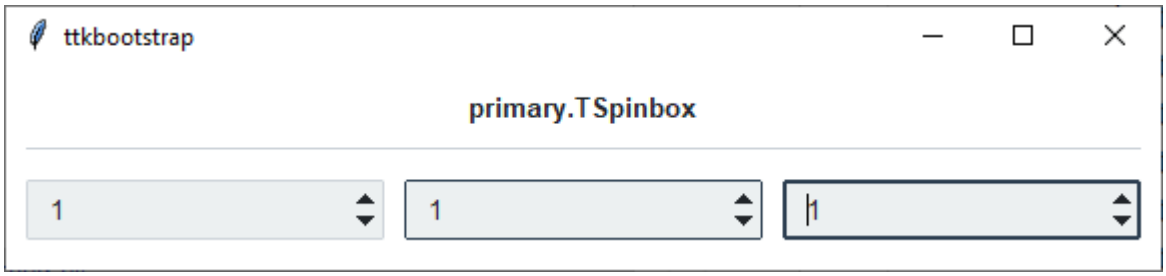
- https://docs.python.org/3/library/tkinter.ttk.html#sizegrip
- https://www.pythontutorial.net/tkinter/tkinter-sizegrip/
- https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/ttk-Sizegrip.html
- https://www.tcl.tk/man/tcl8.6/TkCmd/ttk_sizegrip.htm

2.4.20 Spinbox

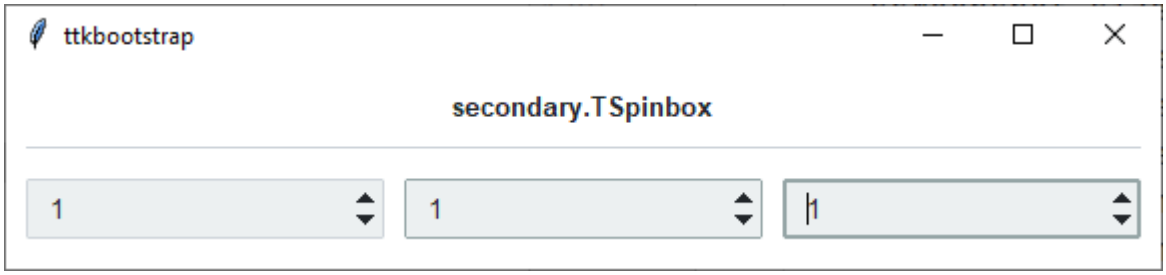
A `ttk.Spinbox` widget is a `ttk.Entry` widget with built-in up and down buttons that are used to either modify a numeric value or to select among a set of values. The widget implements all the features of the `ttk.Entry` widget including support of the `textvariable` option to link the value displayed by the widget to a tkinter variable.

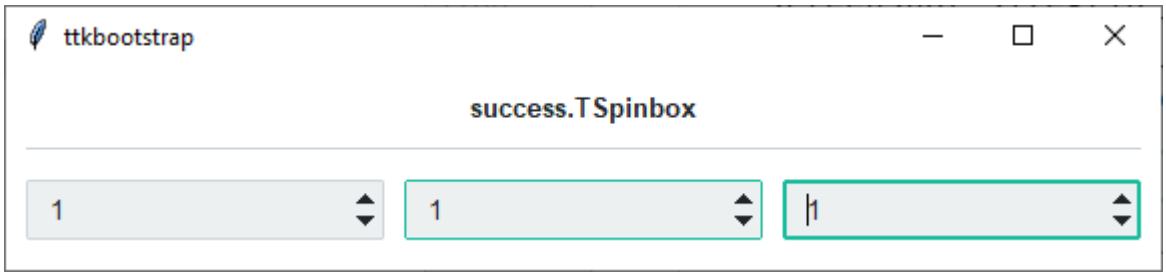
2.4.20.1 Overview

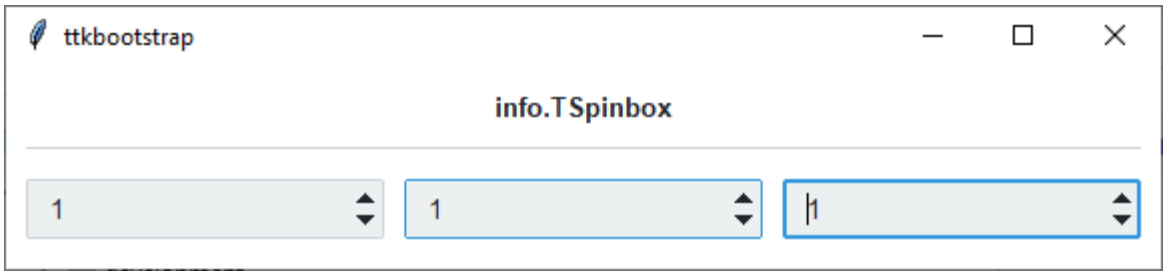
The `ttk.Spinbox` includes the **TSpinbox** class. The *primary* color is applied to this widget by default. This style is further subclassed by each of the theme colors to produce the following color and style combinations.

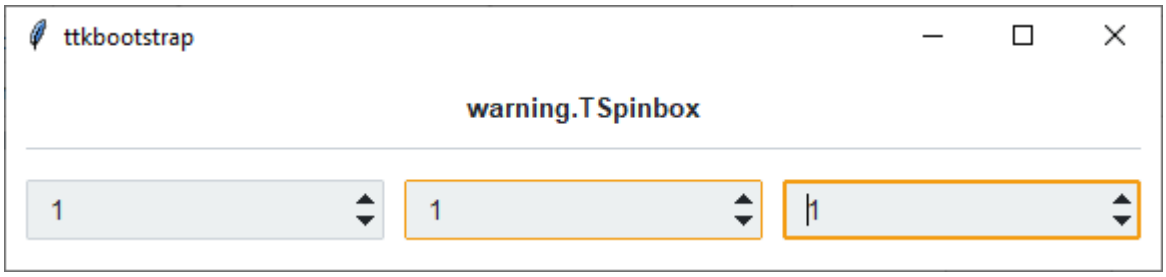


As you can see, in a *normal* state, all styles look the same. What distinguishes them are the colors that are used for the **active** and **focused** states.


ttkbootstrap
secondary.TSpinbox
1
1
1


ttkbootstrap
success.TSpinbox
1
1
1


ttkbootstrap
info.TSpinbox
1
1
1


ttkbootstrap
warning.TSpinbox
1
1
1


ttkbootstrap
danger.TSpinbox
1
1
1

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

Style options

arrowcolor *color*

arrowsize *amount*

background *color* (same as fieldbackground)

bordercolor *color*

darkcolor *color*

fieldbackground *color*

foreground *color*

insertcolor *color*

insertwidth *amount*

lightcolor *color*

padding *padding*

selectbackground *color*

selectforeground *color*

2.4.20.4 Create a custom style

Change the **arrow color** when in different states

```
Style.map('TSpinbox', arrowcolor=[
    ('disabled', 'gray'),
    ('pressed !disabled', 'blue'),
    ('focus !disabled', 'green'),
    ('hover !disabled', 'yellow')])
```

Subclass an existing style to create a new one, using the pattern 'newstyle.OldStyle'

```
Style.configure('custom.TSpinbox', background='green', foreground='white', font=(
↪'Helvetica', 24))
```

Use a custom style

```
ttk.Spinbox(parent, style='custom.TSpinbox')
```

2.4.20.5 References

- https://www.pythontutorial.net/tkinter/tkinter-spinbox/
- https://www.tcl.tk/man/tcl8.6/TkCmd/ttk_spinbox.htm

2.4.21 Treeview

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

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

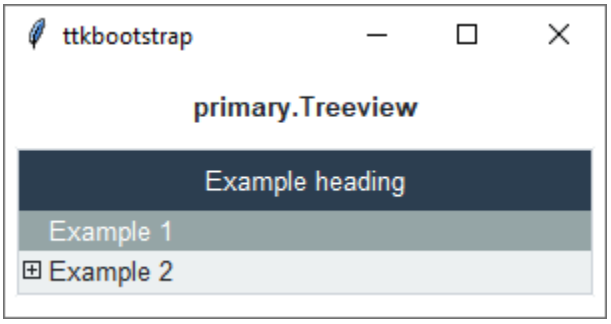
Each item is identified by a unique name. The widget will generate item IDs if they are not supplied by the caller. There is a distinguished root item, named {}. The root item itself is not displayed; its children appear at the top level of the hierarchy.

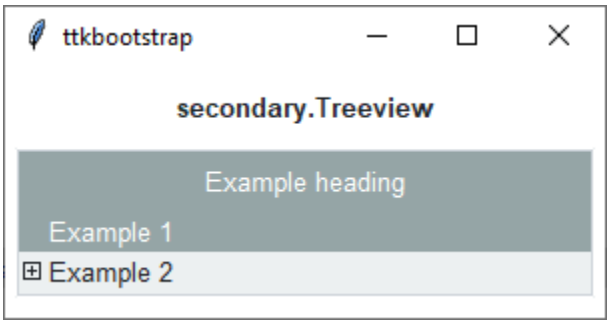
Each item also has a list of tags, which can be used to associate event bindings with individual items and control the appearance of the item.

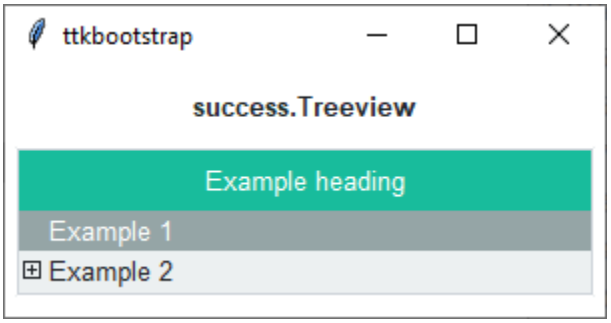
Treeview widgets support horizontal and vertical scrolling with the standard *[xy]scrollcommand* options and *[xy]view* widget commands.

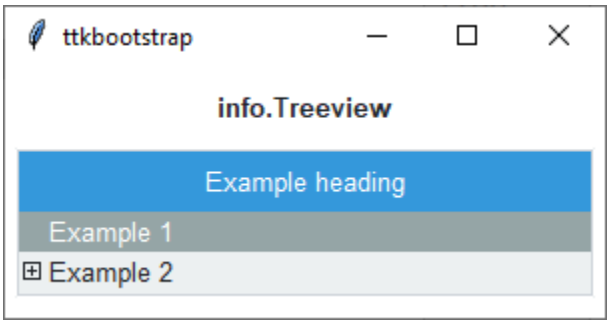
2.4.21.1 Overview

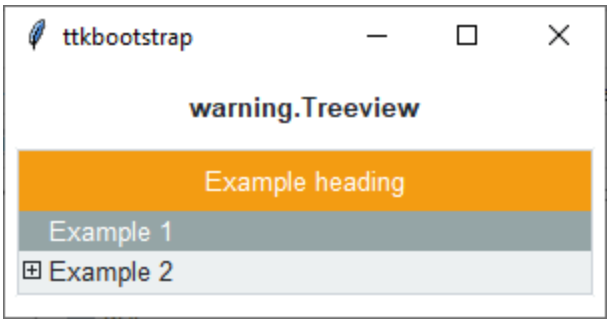
The `ttk.Treeview` includes the **Treeview** class. The *primary* color is applied to this widget by default. This style is further subclassed by each of the theme colors to produce the following color and style combinations.

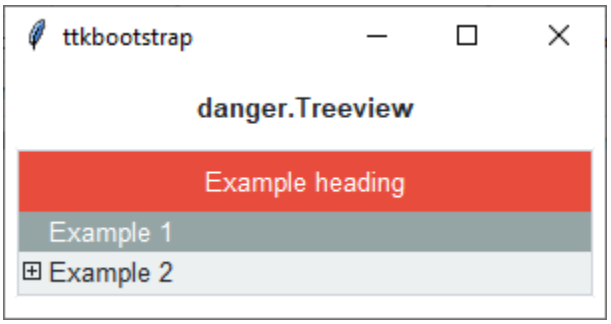












2.4.21.2 How to use

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

Dynamic states

- disabled
- selected

Style options

Treeview styling options include:

background *color*

fieldbackground *color*

font *font*

foreground *color*

rowheight *amount*

Heading styling options include:

background *color*

font *font*

relief *relief*

Item styling options include:

foreground *color*

indicatormargins *padding*

indicatorsize *amount*

padding *padding*

Cell styling options include:

padding *padding*

2.4.21.4 References

- https://docs.python.org/3/library/tkinter.ttk.html#treeview
- https://www.pythontutorial.net/tkinter/tkinter-treeview/
- https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/ttk-Treeview.html
- https://tcl.tk/man/tcl8.6/TkCmd/ttk_treeview.htm

CHAPTER

THREE

GALLERY

Below you will find a *growing* list of ttkbootstrap projects meant to provide inspiration or direction when creating your own applications. These are meant to demonstrate design and are not necessarily fully functional applications.

3.1 File Search Engine

This example demonstrates the use of several styles on the buttons, treeview, and progressbar. The overall theme is **journal**. For individual widgets, the applied styles are:

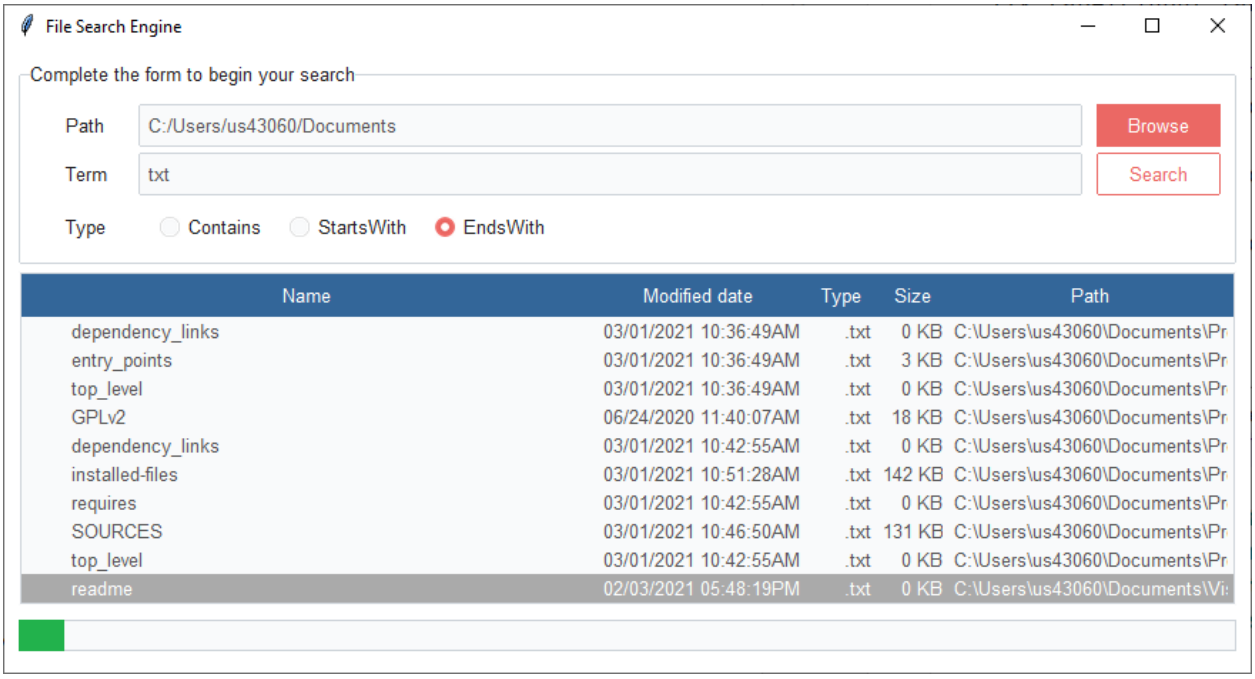
Browse `primary.TButton`

Search `primary.Outline.TButton`

Treeview `info.Treeview`

Progressbar `success.Horizontal.TProgressbar`

Additionally, this application uses threading and a queue to manage IO tasks in order to keep the gui interactive. The treeview updates the results in real-time and sets the focus and view on the most recently inserted result in the results treeview.



Run this code live on repl.it

```
"""
    Author: Israel Dryer
    Modified: 2021-04-09
    Adapted for ttkbootstrap from: https://github.com/israel-dryer/File-Search-Engine-Tk
"""
import csv
import datetime
import pathlib
import tkinter
from queue import Queue
from threading import Thread
from tkinter import ttk
from tkinter.filedialog import askdirectory, asksaveasfilename

from ttkbootstrap import Style


class Application(tkinter.Tk):

    def __init__(self):
        super().__init__()
        self.title('File Search Engine')
        self.style = Style('journal')
        self.search = SearchEngine(self, padding=10)
        self.search.pack(fill='both', expand='yes')


class SearchEngine(ttk.Frame):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # application variables
        self.search_path_var = tkinter.StringVar(value=str(pathlib.Path().absolute()))
        self.search_term_var = tkinter.StringVar(value='txt')
        self.search_type_var = tkinter.StringVar(value='endswidth')
        self.search_count = 0

        # container for user input
        input_labelframe = ttk.Labelframe(self, text='Complete the form to begin your␣
↪search', padding=(20, 10, 10, 5))
        input_labelframe.pack(side='top', fill='x')
        input_labelframe.columnconfigure(1, weight=1)

        # file path input
        ttk.Label(input_labelframe, text='Path').grid(row=0, column=0, padx=10, pady=2,␣
↪sticky='ew')
        e1 = ttk.Entry(input_labelframe, textvariable=self.search_path_var)
        e1.grid(row=0, column=1, sticky='ew', padx=10, pady=2)
        b1 = ttk.Button(input_labelframe, text='Browse', command=self.on_browse, style=
↪'primary.TButton')
        b1.grid(row=0, column=2, sticky='ew', pady=2, ipadx=10)
```

(continues on next page)

(continued from previous page)

```
        # search term input
        ttk.Label(input_labelframe, text='Term').grid(row=1, column=0, padx=10, pady=2,␣
↪sticky='ew')
        e2 = ttk.Entry(input_labelframe, textvariable=self.search_term_var)
        e2.grid(row=1, column=1, sticky='ew', padx=10, pady=2)
        b2 = ttk.Button(input_labelframe, text='Search', command=self.on_search, style=
↪'primary.Outline.TButton')
        b2.grid(row=1, column=2, sticky='ew', pady=2)

        # search type selection
        ttk.Label(input_labelframe, text='Type').grid(row=2, column=0, padx=10, pady=2,␣
↪sticky='ew')
        option_frame = ttk.Frame(input_labelframe, padding=(15, 10, 0, 10))
        option_frame.grid(row=2, column=1, columnspan=2, sticky='ew')
        r1 = ttk.Radiobutton(option_frame, text='Contains', value='contains',␣
↪variable=self.search_type_var)
        r1.pack(side='left', fill='x', pady=2, padx=10)
        r2 = ttk.Radiobutton(option_frame, text='StartsWith', value='startswith',␣
↪variable=self.search_type_var)
        r2.pack(side='left', fill='x', pady=2, padx=10)
        r3 = ttk.Radiobutton(option_frame, text='EndsWith', value='endswith',␣
↪variable=self.search_type_var)
        r3.pack(side='left', fill='x', pady=2, padx=10)
        r3.invoke()

        # search results tree
        self.tree = ttk.Treeview(self, style='info.Treeview')
        self.tree.pack(fill='both', pady=5)
        self.tree['columns'] = ('modified', 'type', 'size', 'path')
        self.tree.column('#0', width=400)
        self.tree.column('modified', width=150, stretch=False, anchor='e')
        self.tree.column('type', width=50, stretch=False, anchor='e')
        self.tree.column('size', width=50, stretch=False, anchor='e')
        self.tree.heading('#0', text='Name')
        self.tree.heading('modified', text='Modified date')
        self.tree.heading('type', text='Type')
        self.tree.heading('size', text='Size')
        self.tree.heading('path', text='Path')

        # progress bar
        self.progressbar = ttk.Progressbar(self, orient='horizontal', mode='indeterminate
↪',
                                           style='success.Horizontal.TProgressbar')
        self.progressbar.pack(fill='x', pady=5)

        # right-click menu for treeview
        self.menu = tkinter.Menu(self, tearoff=False)
        self.menu.add_command(label='Reveal in file manager', command=self.on_
↪doubleclick_tree)
        self.menu.add_command(label='Export results to csv', command=self.export_to_csv)

        # event binding
```

(continues on next page)

(continued from previous page)

```
        self.tree.bind('<Double-1>', self.on_doubleclick_tree)
        self.tree.bind('<Button-3>', self.right_click_tree)

    def on_browse(self):
        """Callback for directory browse"""
        path = askdirectory(title='Directory')
        if path:
            self.search_path_var.set(path)

    def on_doubleclick_tree(self, event=None):
        """Callback for double-click tree menu"""
        try:
            id = self.tree.selection()[0]
        except IndexError:
            return
        if id.startswith('I'):
            self.reveal_in_explorer(id)

    def right_click_tree(self, event=None):
        """Callback for right-click tree menu"""
        try:
            id = self.tree.selection()[0]
        except IndexError:
            return
        if id.startswith('I'):
            self.menu.entryconfigure('Export results to csv', state='disabled')
            self.menu.entryconfigure('Reveal in file manager', state='normal')
        else:
            self.menu.entryconfigure('Export results to csv', state='normal')
            self.menu.entryconfigure('Reveal in file manager', state='disabled')
        self.menu.post(event.x_root, event.y_root)

    def on_search(self):
        """Search for a term based on the search type"""
        search_term = self.search_term_var.get()
        search_path = self.search_path_var.get()
        search_type = self.search_type_var.get()
        if search_term == '':
            return
        Thread(target=SearchEngine.file_search, args=(search_term, search_path, search_
↪type), daemon=True).start()
        self.progressbar.start(10)
        self.search_count += 1
        id = self.tree.insert('', 'end', self.search_count, text=f'Search {self.search_
↪count}')
        self.tree.item(id, open=True)
        self.check_queue(id)

    def reveal_in_explorer(self, id):
        """Callback for double-click event on tree"""
        values = self.tree.item(id, 'values')
        path = pathlib.Path(values[-1]).absolute().parent
```

(continues on next page)

(continued from previous page)

```
        pathlib.os.startfile(path)

    def export_to_csv(self, event=None):
        """Export values to csv file"""
        try:
            id = self.tree.selection()[0]
        except IndexError:
            return

        filename = asksaveasfilename(initialfile='results.csv',
                                     filetypes=[('Comma-separated', '*.csv'), ('Text',
↪'*.txt')])
        if filename:
            with open(filename, mode='w', newline='') as f:
                writer = csv.writer(f)
                writer.writerow(['Name', 'Modified date', 'Type', 'Size', 'Path'])
                children = self.tree.get_children(id)
                for child in children:
                    name = [self.tree.item(child, 'text')]
                    values = list(self.tree.item(child, 'values'))
                    writer.writerow(name + values)
        # open file in explorer
        pathlib.os.startfile(filename)

    def check_queue(self, id):
        """Check file queue and print results if not empty"""
        if searching and not file_queue.empty():
            filename = file_queue.get()
            self.insert_row(filename, id)
            self.update_idletasks()
            self.after(1, lambda: self.check_queue(id))
        elif not searching and not file_queue.empty():
            while not file_queue.empty():
                filename = file_queue.get()
                self.insert_row(filename, id)
            self.update_idletasks()
            self.progressbar.stop()
        elif searching and file_queue.empty():
            self.after(100, lambda: self.check_queue(id))
        else:
            self.progressbar.stop()

    def insert_row(self, file, id):
        """Insert new row in tree search results"""
        try:
            file_stats = file.stat()
            file_name = file.stem
            file_modified = datetime.datetime.fromtimestamp(file_stats.st_mtime).
↪strftime('%m/%d/%Y %I:%M:%S%p')
            file_type = file.suffix.lower()
            file_size = SearchEngine.convert_size(file_stats.st_size)
            file_path = file.absolute()
```

(continues on next page)

(continued from previous page)

```
            iid = self.tree.insert(id, 'end', text=file_name, values=(file_modified,␣
↪file_type, file_size, file_path))
            self.tree.selection_set(iid)
            self.tree.see(iid)
        except OSError:
            return

    @staticmethod
    def file_search(term, search_path, search_type):
        """Recursively search directory for matching files"""
        SearchEngine.set_searching(1)
        if search_type == 'contains':
            SearchEngine.find_contains(term, search_path)
        elif search_type == 'startswith':
            SearchEngine.find_startswith(term, search_path)
        elif search_type == 'endswith':
            SearchEngine.find_endswith(term, search_path)

    @staticmethod
    def find_contains(term, search_path):
        """Find all files that contain the search term"""
        for path, _, files in pathlib.os.walk(search_path):
            if files:
                for file in files:
                    if term in file:
                        file_queue.put(pathlib.Path(path) / file)
        SearchEngine.set_searching(False)

    @staticmethod
    def find_startswith(term, search_path):
        """Find all files that start with the search term"""
        for path, _, files in pathlib.os.walk(search_path):
            if files:
                for file in files:
                    if file.startswith(term):
                        file_queue.put(pathlib.Path(path) / file)
        SearchEngine.set_searching(False)

    @staticmethod
    def find_endswith(term, search_path):
        """Find all files that end with the search term"""
        for path, _, files in pathlib.os.walk(search_path):
            if files:
                for file in files:
                    if file.endswith(term):
                        file_queue.put(pathlib.Path(path) / file)
        SearchEngine.set_searching(False)

    @staticmethod
    def set_searching(state=False):
        """Set searching status"""
        global searching
```

(continues on next page)

(continued from previous page)

```
        searching = state

    @staticmethod
    def convert_size(size):
        """Convert bytes to mb or kb depending on scale"""
        kb = size // 1000
        mb = round(kb / 1000, 1)
        if kb > 1000:
            return f'{mb:,.1f} MB'
        else:
            return f'{kb:,d} KB'


if __name__ == '__main__':
    file_queue = Queue()
    searching = False
    Application().mainloop()
```

3.2 File Backup Utility

In this example, I demonstrate how to use various styles to build a UI for a File Backup UI. The reference material is from an image you can find here. The overall theme of this application is **flatly**. I use a `CollapsingFrame` class to contain the left-side info panels as well as the output on the bottom right. These contain indicator buttons on the right-side of the header which collapse and expand the frame with a mouse-click action.

Some of the styles used in this application include:

top button bar `primary.TButton`

collapsible frames `secondary.TButton`

separators `secondary.Horizontal.TSeparator`

progress bar `success.Horizontal.TProgressbar`

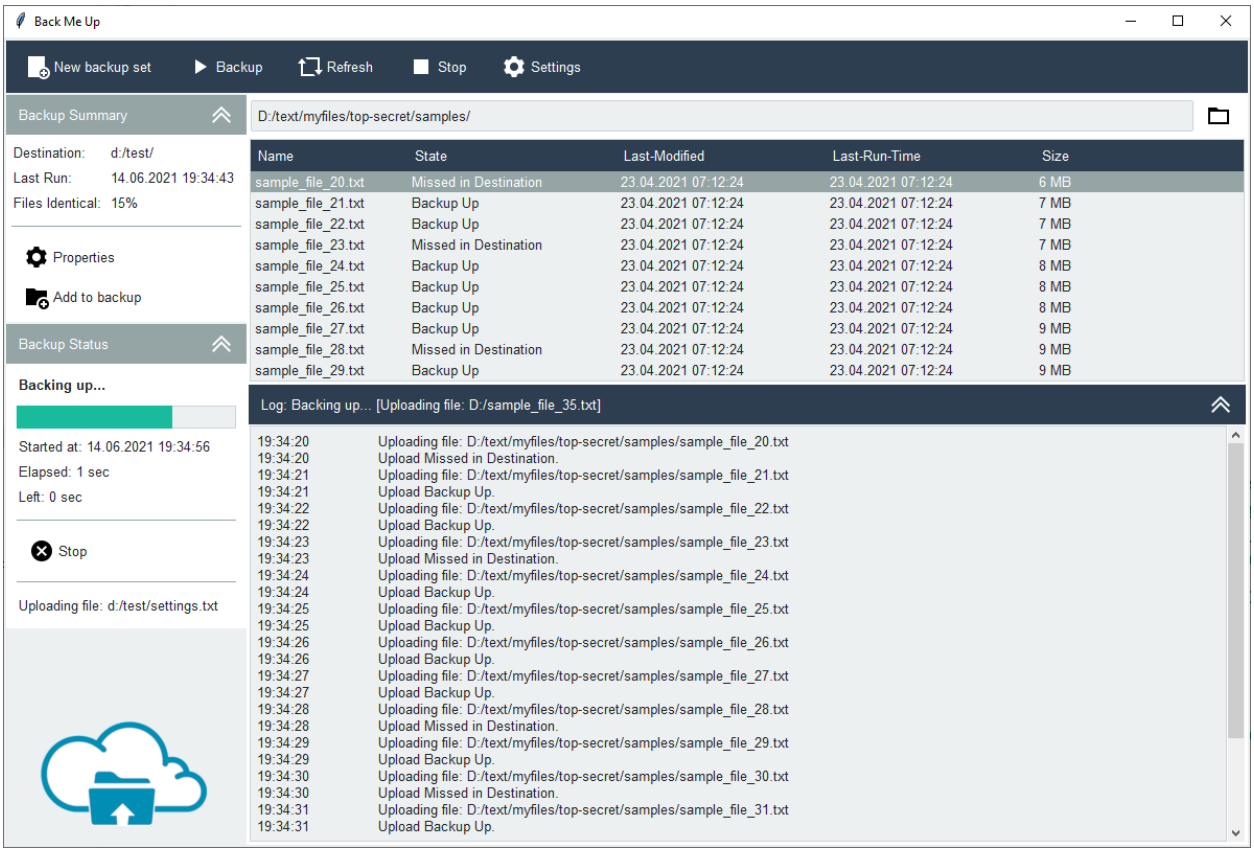
properties, stop, add-to-backup buttons `Link.TButton`

file open button `secondary.Link.TButton`

There are two custom styles which are subclassed from `TFrame` and `TLabel`. I used the **inputbg** color from the `Style.colors` property and applied this style to the left panel, and the logo image background.

```
"""
    Author: Israel Dryer
    Modified: 2021-04-23
    Adapted for ttkbootstrap from: http://www.leo-backup.com/screenshots.shtml
"""
import tkinter
from datetime import datetime
from random import choices
from tkinter import ttk
from tkinter.filedialog import askdirectory
from tkinter.messagebox import showinfo
from tkinter.scrolledtext import ScrolledText
```

(continues on next page)



(continued from previous page)

```
from pathlib import Path

from ttkbootstrap import Style


class Application(tkinter.Tk):

    def __init__(self):
        super().__init__()
        self.title('Back Me Up')
        self.style = Style()
        self.style.configure('bg.TFrame', background=self.style.colors.inputbg)
        self.style.configure('bg.TLabel', background=self.style.colors.inputbg)
        self.bmu = BackMeUp(self, padding=2, style='bg.TFrame')
        self.bmu.pack(fill='both', expand='yes')


class BackMeUp(ttk.Frame):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        # images
        p = Path(__file__).parent
```

(continues on next page)

(continued from previous page)

```
        self.img_properties_d = tkinter.PhotoImage(name='properties-dark', file=p /
↪'assets/icons8_settings_24px.png')
        self.img_properties_l = tkinter.PhotoImage(name='properties-light', file=p /
↪'assets/icons8_settings_24px_2.png')
        self.img_addtobackup_d = tkinter.PhotoImage(name='add-to-backup-dark', file=p /
↪'assets/icons8_add_folder_24px.png')
        self.img_addtobackup_l = tkinter.PhotoImage(name='add-to-backup-light', file=p /
↪'assets/icons8_add_book_24px.png')
        self.img_stopbackup_d = tkinter.PhotoImage(name='stop-backup-dark', file=p /
↪'assets/icons8_cancel_24px.png')
        self.img_stopbackup_l = tkinter.PhotoImage(name='stop-backup-light', file=p /
↪'assets/icons8_cancel_24px_1.png')
        self.img_play = tkinter.PhotoImage(name='play', file=p / 'assets/icons8_play_
↪24px_1.png')
        self.img_refresh = tkinter.PhotoImage(name='refresh', file=p / 'assets/icons8_
↪refresh_24px_1.png')
        self.img_stop_d = tkinter.PhotoImage(name='stop-dark', file=p / 'assets/icons8_
↪stop_24px.png')
        self.img_stop_l = tkinter.PhotoImage(name='stop-light', file=p / 'assets/icons8_
↪stop_24px_1.png')
        self.img_opened_folder = tkinter.PhotoImage(name='opened-folder', file=p /
↪'assets/icons8_opened_folder_24px.png')
        self.img_logo = tkinter.PhotoImage(name='logo', file=p / 'assets/backup.png')

        # ----- buttonbar
        buttonbar = ttk.Frame(self, style='primary.TFrame')
        buttonbar.pack(fill='x', pady=1, side='top')

        ## new backup
        bb_new_backup_btn = ttk.Button(buttonbar, text='New backup set', image='add-to-
↪backup-light', compound='left')
        bb_new_backup_btn.configure(command=lambda: showinfo(message='Adding new backup
↪'))
        bb_new_backup_btn.pack(side='left', ipadx=5, ipady=5, padx=(1, 0), pady=1)

        ## backup
        bb_backup_btn = ttk.Button(buttonbar, text='Backup', image='play', compound='left
↪')
        bb_backup_btn.configure(command=lambda: showinfo(message='Backing up...'))
        bb_backup_btn.pack(side='left', ipadx=5, ipady=5, padx=0, pady=1)

        ## refresh
        bb_refresh_btn = ttk.Button(buttonbar, text='Refresh', image='refresh', compound=
↪'left')
        bb_refresh_btn.configure(command=lambda: showinfo(message='Refreshing...'))
        bb_refresh_btn.pack(side='left', ipadx=5, ipady=5, padx=0, pady=1)

        ## stop
        bb_stop_btn = ttk.Button(buttonbar, text='Stop', image='stop-light', compound=
↪'left')
        bb_stop_btn.configure(command=lambda: showinfo(message='Stopping backup.'))
        bb_stop_btn.pack(side='left', ipadx=5, ipady=5, padx=0, pady=1)
```

(continues on next page)

(continued from previous page)

```
        ## settings
        bb_settings_btn = ttk.Button(buttonbar, text='Settings', image='properties-light
↪', compound='left')
        bb_settings_btn.configure(command=lambda: showinfo(message='Changing settings'))
        bb_settings_btn.pack(side='left', ipadx=5, ipady=5, padx=0, pady=1)

        # ----- left panel
        left_panel = ttk.Frame(self, style='bg.TFrame')
        left_panel.pack(side='left', fill='y')

        ## ----- backup summary (collapsible)
        bus_cf = CollapsingFrame(left_panel)
        bus_cf.pack(fill='x', pady=1)

        ## container
        bus_frm = ttk.Frame(bus_cf, padding=5)
        bus_frm.columnconfigure(1, weight=1)
        bus_cf.add(bus_frm, title='Backup Summary', style='secondary.TButton')

        ## destination
        ttk.Label(bus_frm, text='Destination:').grid(row=0, column=0, sticky='w', pady=2)
        ttk.Label(bus_frm, textvariable='destination').grid(row=0, column=1, sticky='ew',
↪ padx=5, pady=2)
        self.setvar('destination', 'd:/test/')

        ## last run
        ttk.Label(bus_frm, text='Last Run:').grid(row=1, column=0, sticky='w', pady=2)
        ttk.Label(bus_frm, textvariable='lastrun').grid(row=1, column=1, sticky='ew',␣
↪padx=5, pady=2)
        self.setvar('lastrun', '14.06.2021 19:34:43')

        ## files Identical
        ttk.Label(bus_frm, text='Files Identical:').grid(row=2, column=0, sticky='w',␣
↪pady=2)
        ttk.Label(bus_frm, textvariable='filesidentical').grid(row=2, column=1, sticky=
↪'ew', padx=5, pady=2)
        self.setvar('filesidentical', '15%')

        ## section separator
        bus_sep = ttk.Separator(bus_frm, style='secondary.Horizontal.TSeparator')
        bus_sep.grid(row=3, column=0, columnspan=2, pady=10, sticky='ew')

        ## properties button
        bus_prop_btn = ttk.Button(bus_frm, text='Properties', image='properties-dark',␣
↪compound='left')
        bus_prop_btn.configure(command=lambda: showinfo(message='Changing properties'),␣
↪style='Link.TButton')
        bus_prop_btn.grid(row=4, column=0, columnspan=2, sticky='w')

        ## add to backup button
        bus_add_btn = ttk.Button(bus_frm, text='Add to backup', image='add-to-backup-dark
↪', compound='left')
```

(continues on next page)

(continued from previous page)

```
        bus_add_btn.configure(command=lambda: showinfo(message='Adding to backup'),␣
↪style='Link.TButton')
        bus_add_btn.grid(row=5, column=0, columnspan=2, sticky='w')

        # ----- backup status (collapsible)
        status_cf = CollapsingFrame(left_panel)
        status_cf.pack(fill='x', pady=1)

        ## container
        status_frm = ttk.Frame(status_cf, padding=10)
        status_frm.columnconfigure(1, weight=1)
        status_cf.add(status_frm, title='Backup Status', style='secondary.TButton')

        ## progress message
        status_prog_lbl = ttk.Label(status_frm, textvariable='prog-message', font=
↪'Helvetica 10 bold')
        status_prog_lbl.grid(row=0, column=0, columnspan=2, sticky='w')
        self.setvar('prog-message', 'Backing up...')

        ## progress bar
        status_prog = ttk.Progressbar(status_frm, variable='prog-value', style='success.
↪Horizontal.TProgressbar')
        status_prog.grid(row=1, column=0, columnspan=2, sticky='ew', pady=(10, 5))
        self.setvar('prog-value', 71)

        ## time started
        ttk.Label(status_frm, textvariable='prog-time-started').grid(row=2, column=0,␣
↪columnspan=2, sticky='ew', pady=2)
        self.setvar('prog-time-started', 'Started at: 14.06.2021 19:34:56')

        ## time elapsed
        ttk.Label(status_frm, textvariable='prog-time-elapsed').grid(row=3, column=0,␣
↪columnspan=2, sticky='ew', pady=2)
        self.setvar('prog-time-elapsed', 'Elapsed: 1 sec')

        ## time remaining
        ttk.Label(status_frm, textvariable='prog-time-left').grid(row=4, column=0,␣
↪columnspan=2, sticky='ew', pady=2)
        self.setvar('prog-time-left', 'Left: 0 sec')

        ## section separator
        status_sep = ttk.Separator(status_frm, style='secondary.Horizontal.TSeparator')
        status_sep.grid(row=5, column=0, columnspan=2, pady=10, sticky='ew')

        ## stop button
        status_stop_btn = ttk.Button(status_frm, text='Stop', image='stop-backup-dark',␣
↪compound='left')
        status_stop_btn.configure(command=lambda: showinfo(message='Stopping backup'),␣
↪style='Link.TButton')
        status_stop_btn.grid(row=6, column=0, columnspan=2, sticky='w')

        ## section separator
```

(continues on next page)

(continued from previous page)

```
        status_sep = ttk.Separator(status_frm, style='secondary.Horizontal.TSeparator')
        status_sep.grid(row=7, column=0, columnspan=2, pady=10, sticky='ew')

        # current file message
        ttk.Label(status_frm, textvariable='current-file-msg').grid(row=8, column=0,␣
↪columnspan=2, pady=2, sticky='ew')
        self.setvar('current-file-msg', 'Uploading file: d:/test/settings.txt')

        # logo
        ttk.Label(left_panel, image='logo', style='bg.TLabel').pack(side='bottom')

        # ---- right panel
        right_panel = ttk.Frame(self, padding=(2, 1))
        right_panel.pack(side='right', fill='both', expand='yes')

        ## file input
        browse_frm = ttk.Frame(right_panel)
        browse_frm.pack(side='top', fill='x', padx=2, pady=1)
        file_entry = ttk.Entry(browse_frm, textvariable='folder-path')
        file_entry.pack(side='left', fill='x', expand='yes')
        open_btn = ttk.Button(browse_frm, image='opened-folder', style='secondary.Link.
↪TButton',
                              command=self.get_directory)
        open_btn.pack(side='right')

        ## Treeview
        tv = ttk.Treeview(right_panel, show='headings')
        tv['columns'] = ('name', 'state', 'last-modified', 'last-run-time', 'size')
        tv.column('name', width=150, stretch=True)
        for col in ['last-modified', 'last-run-time', 'size']:
            tv.column(col, stretch=False)
        for col in tv['columns']:
            tv.heading(col, text=col.title(), anchor='w')
        tv.pack(fill='x', pady=1)

        ## scrolling text output
        scroll_cf = CollapsingFrame(right_panel)
        scroll_cf.pack(fill='both', pady=1)
        output_container = ttk.Frame(scroll_cf, padding=1)
        self.setvar('scroll-message', 'Log: Backing up... [Uploading file: D:/sample_
↪file_35.txt]')
        st = ScrolledText(output_container)
        st.pack(fill='both', expand='yes')
        scroll_cf.add(output_container, textvariable='scroll-message')

        # ----- seed with some sample data ----------------------------------------------
↪-------------------------------

        ## starting sample directory
        file_entry.insert('end', 'D:/text/myfiles/top-secret/samples/')

        ## treeview and backup logs
```

(continues on next page)

(continued from previous page)

```
        for x in range(20, 35):
            result = choices(['Backup Up', 'Missed in Destination'])[0]
            st.insert('end', f'19:34:{x}\t\t Uploading file: D:/text/myfiles/top-secret/
↪samples/sample_file_{x}.txt\n')
            st.insert('end', f'19:34:{x}\t\t Upload {result}.\n')
            timestamp = datetime.now().strftime('%d.%m.%Y %H:%M:%S')
            tv.insert('', 'end', x, values=(f'sample_file_{x}.txt', result, timestamp,␣
↪timestamp, f'{int(x // 3)} MB'))
        tv.selection_set(20)

    def get_directory(self):
        """Open dialogue to get directory and update directory variable"""
        self.update_idletasks()
        d = askdirectory()
        if d:
            self.setvar('folder-path', d)


class CollapsingFrame(ttk.Frame):
    """
    A collapsible frame widget that opens and closes with a button click.
    """

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.columnconfigure(0, weight=1)
        self.cumulative_rows = 0
        p = Path(__file__).parent
        self.images = [tkinter.PhotoImage(name='open', file=p / 'assets/icons8_double_up_
↪24px.png'),
                       tkinter.PhotoImage(name='closed', file=p / 'assets/icons8_double_
↪right_24px.png')]

    def add(self, child, title="", style='primary.TButton', **kwargs):
        """Add a child to the collapsible frame

        :param ttk.Frame child: the child frame to add to the widget
        :param str title: the title appearing on the collapsible section header
        :param str style: the ttk style to apply to the collapsible section header
        """
        if child.winfo_class() != 'TFrame':  # must be a frame
            return
        style_color = style.split('.')[0]
        frm = ttk.Frame(self, style=f'{style_color}.TFrame')
        frm.grid(row=self.cumulative_rows, column=0, sticky='ew')

        # header title
        lbl = ttk.Label(frm, text=title, style=f'{style_color}.Inverse.TLabel')
        if kwargs.get('textvariable'):
            lbl.configure(textvariable=kwargs.get('textvariable'))
        lbl.pack(side='left', fill='both', padx=10)
```

(continues on next page)

(continued from previous page)

```
        # header toggle button
        btn = ttk.Button(frm, image='open', style=style, command=lambda c=child: self._
↪toggle_open_close(child))
        btn.pack(side='right')

        # assign toggle button to child so that it's accesible when toggling (need to␣
↪change image)
        child.btn = btn
        child.grid(row=self.cumulative_rows + 1, column=0, sticky='news')

        # increment the row assignment
        self.cumulative_rows += 2

    def _toggle_open_close(self, child):
        """
        Open or close the section and change the toggle button image accordingly

        :param ttk.Frame child: the child element to add or remove from grid manager
        """
        if child.winfo_viewable():
            child.grid_remove()
            child.btn.configure(image='closed')
        else:
            child.grid()
            child.btn.configure(image='open')


if __name__ == '__main__':
    Application().mainloop()
```

3.3 Media Player

This example demonstrates how to build a media player GUI. The buttons are simple unicode characters. The overall theme is **minty** and the following styles are applied to the widgets:

Control Buttons `primary.TButton`

File Button `secondary.TButton`

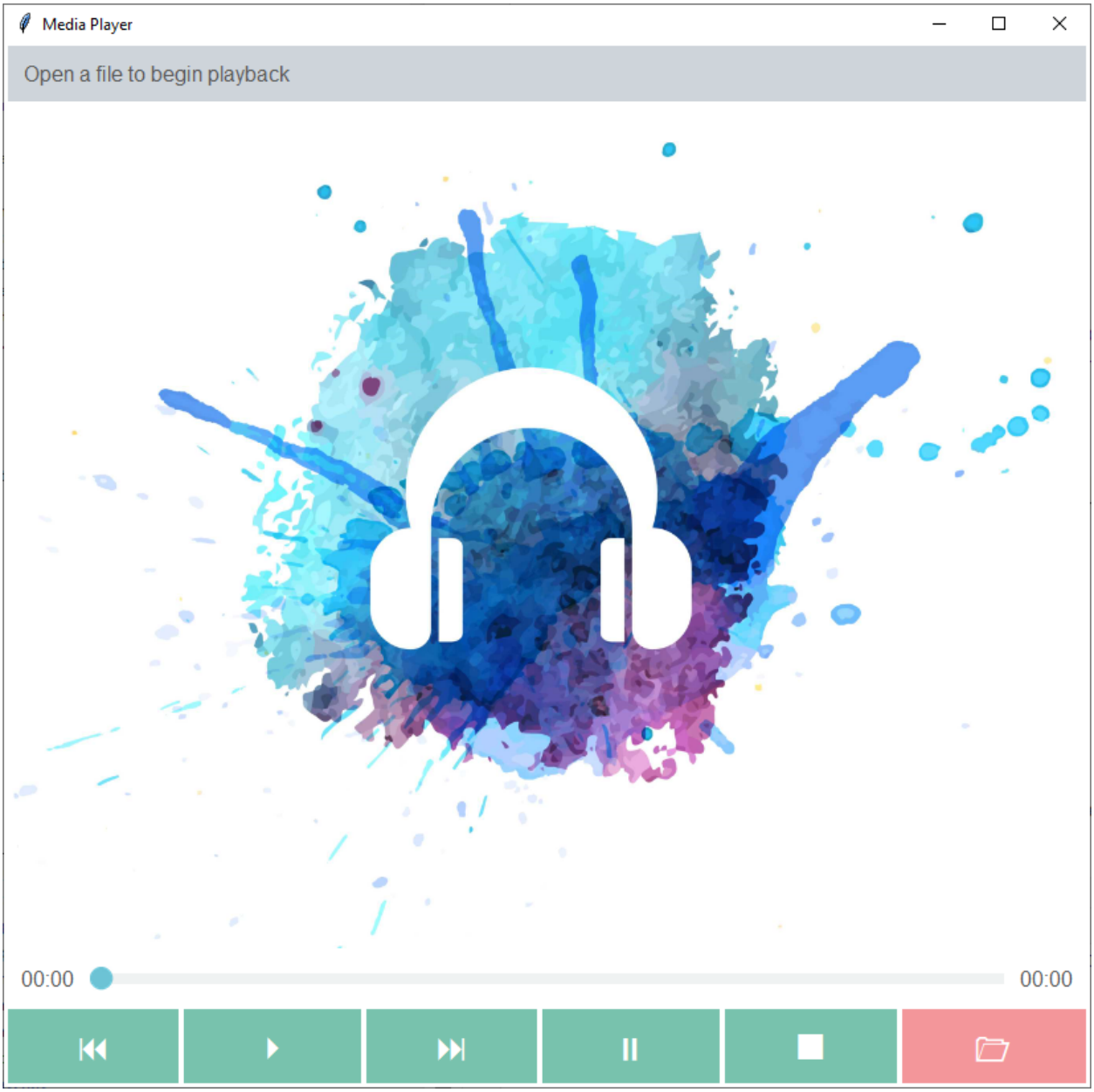
Scale `info.Horizontal.TScale`

Additionally, I subclassed the `TLabel` to create a new `header.TLabel` style that changes the background using the theme color `border` with some additional padding.

This is a ttkbootstrap adaptation of the media player GUI you can find here, which includes the implementation of the VLC package for controlling audio and video.

```
"""
    Author: Israel Dryer
    Modified: 2021-04-07
    Adapted for ttkbootstrap from: https://github.com/israel-dryer/Mini-VLC-Player
"""
```

(continues on next page)


Media Player
Open a file to begin playback
00:00
00:00

(continued from previous page)

```
import tkinter
from tkinter import ttk
from pathlib import Path
from ttkbootstrap import Style


class Application(tkinter.Tk):

    def __init__(self):
        super().__init__()
        self.title('Media Player')
        self.style = Style()
        self.style.theme_use('minty')
        self.player = Player(self)
        self.player.pack(fill='both', expand='yes')
        self.style.configure('TButton', font='Helvetica 20')
        self.style.configure('header.TLabel', background=self.style.colors.border,␣
↪padding=10)


class Player(ttk.Frame):
    """
    An interface for a media player
    """

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.configure(padding=1)
        self.background = tkinter.PhotoImage(file=Path(__file__).parent/'assets/mp_
↪background.png')
        self.controls = {
            'skip-previous': '\u23EE',
            'play': '\u23F5',
            'pause': '\u23F8',
            'stop': '\u23F9',
            'skip-next': '\u23ED',
            'open-file': '\U0001f4c2'}

        # track information header
        self.track_info = tkinter.StringVar(value='Open a file to begin playback')
        header = ttk.Label(self, textvariable=self.track_info, font='Helvetica 12',␣
↪style='header.TLabel')
        header.pack(fill='x', padx=2)

        # media container
        self.container = ttk.Label(self, image=self.background)
        self.container.pack(fill='both', expand='yes')

        # progress bar
        progress_frame = ttk.Frame(self, padding=10)
        progress_frame.pack(fill='x', expand='yes')
        self.time_elapsed = ttk.Label(progress_frame, text='00:00', font='Helvetica 12')
```

(continues on next page)

(continued from previous page)

```
        self.time_elapsed.pack(side='left')
        self.time_scale = ttk.Scale(progress_frame, orient='horizontal', style='info.
↪Horizontal.TScale')
        self.time_scale.pack(side='left', fill='x', expand='yes', padx=10)
        self.time_remaining = ttk.Label(progress_frame, text='00:00', font='Helvetica 12
↪')
        self.time_remaining.pack(side='right')

        # button controls
        control_frame = ttk.Frame(self)
        control_frame.pack(fill='x', expand='yes')
        self.buttons = {
            'play': ttk.Button(control_frame, text=self.controls['play']),
            'skip-previous': ttk.Button(control_frame, text=self.controls['skip-previous
↪']),
            'skip-next': ttk.Button(control_frame, text=self.controls['skip-next']),
            'pause': ttk.Button(control_frame, text=self.controls['pause']),
            'stop': ttk.Button(control_frame, text=self.controls['stop']),
            'open-file': ttk.Button(control_frame, text=self.controls['open-file'],␣
↪style='secondary.TButton')}
        for button in ['skip-previous', 'play', 'skip-next', 'pause', 'stop', 'open-file
↪']:
            self.buttons[button].pack(side='left', fill='x', expand='yes', ipadx=5,␣
↪ipady=5, padx=2, pady=2)


if __name__ == '__main__':
    Application().mainloop()
```

3.4 Magic Mouse

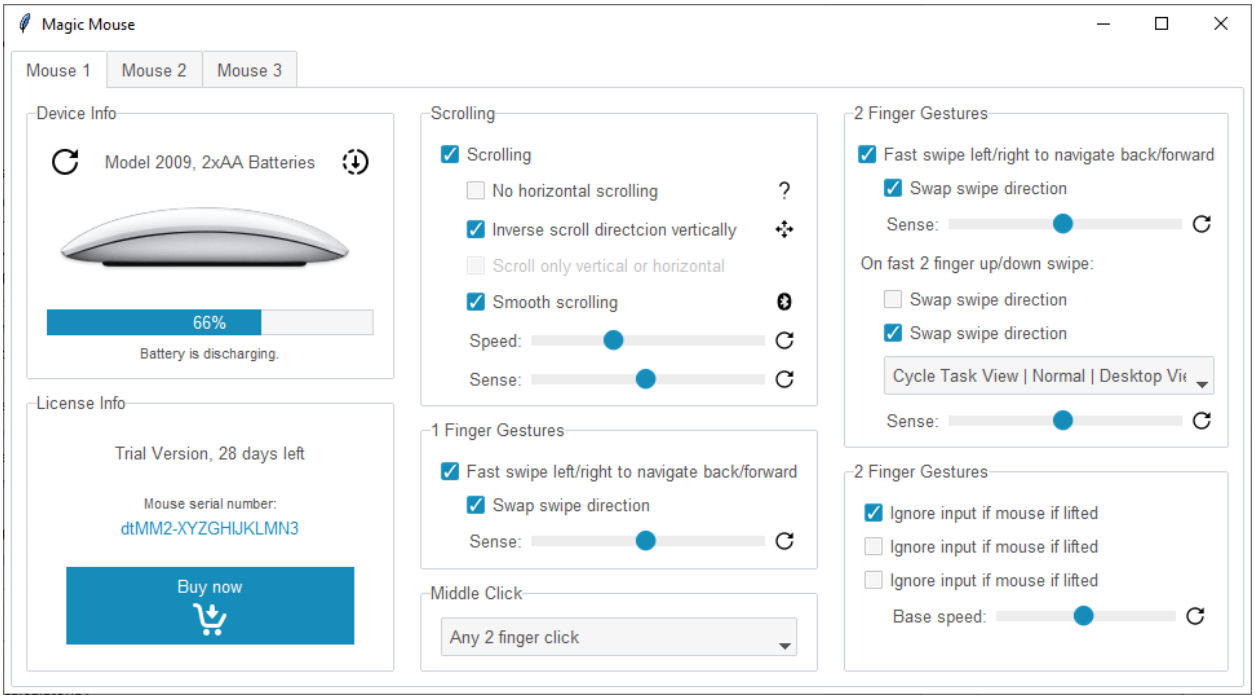
This application demonstrates a complicated design with many options and several label frames. The overall theme is **lumen**. Other than the default styles, the following styles are applied directly to various widgets widgets:

Image Buttons `Link.TButton`

License Number `primary.TLabel`

```
"""
    Author: Israel Dryer
    Modified: 2021-04-13
    Adapted for ttkbootstrap from: https://magicutilities.net/magic-mouse/features
"""
import tkinter
from tkinter import PhotoImage
from tkinter import ttk
from tkinter.messagebox import showinfo
from pathlib import Path
from ttkbootstrap import Style
```

(continues on next page)



(continued from previous page)

```
class Application(tkinter.Tk):

    def __init__(self):
        super().__init__()
        self.title('Magic Mouse')
        self.style = Style('lumen')
        self.window = ttk.Frame(self)
        self.window.pack(fill='both', expand='yes')
        self.nb = ttk.Notebook(self.window)
        self.nb.pack(fill='both', expand='yes', padx=5, pady=5)
        mu = MouseUtilities(self.nb)
        self.nb.add(mu, text='Mouse 1')

        # add demo tabs
        self.nb.add(ttk.Frame(self.nb), text='Mouse 2')
        self.nb.add(ttk.Frame(self.nb), text='Mouse 3')


class MouseUtilities(ttk.Frame):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        p = Path(__file__).parent
        self.images = {
            'reset': PhotoImage(name='reset', file=p/'assets/magic_mouse/icons8_reset_
↪24px.png'),
            'reset-small': PhotoImage(name='reset-small', file=p/'assets/magic_mouse/
↪icons8_reset_16px.png'),
            'submit': PhotoImage(name='submit', file=p/'assets/magic_mouse/icons8_submit_
↪progress_24px.png'),
```

(continues on next page)

(continued from previous page)

```
            'question': PhotoImage(name='question', file=p/'assets/magic_mouse/icons8_
↪question_mark_16px.png'),
            'direction': PhotoImage(name='direction', file=p/'assets/magic_mouse/icons8_
↪move_16px.png'),
            'bluetooth': PhotoImage(name='bluetooth', file=p/'assets/magic_mouse/icons8_
↪bluetooth_2_16px.png'),
            'buy': PhotoImage(name='buy', file=p/'assets/magic_mouse/icons8_buy_26px_2.
↪png'),
            'mouse': PhotoImage(name='mouse', file=p/'assets/magic_mouse/magic_mouse.png
↪')
        }

        for i in range(3):
            self.columnconfigure(i, weight=1)
        self.rowconfigure(0, weight=1)

        # Column 1␣
↪==========================================================================================
        col1 = ttk.Frame(self, padding=10)
        col1.grid(row=0, column=0, sticky='news')

        ## device info ------------------------------------------------------------------
↪--------------------------------
        dev_info = ttk.Labelframe(col1, text='Device Info', padding=10)
        dev_info.pack(side='top', fill='both', expand='yes')

        ### header
        dev_info_header = ttk.Frame(dev_info, padding=5)
        dev_info_header.pack(fill='x')
        ttk.Button(dev_info_header, image='reset', style='Link.TButton', command=self.
↪callback).pack(side='left')
        ttk.Label(dev_info_header, text='Model 2009, 2xAA Batteries').pack(side='left',␣
↪fill='x', padx=15)
        ttk.Button(dev_info_header, image='submit', style='Link.TButton', command=self.
↪callback).pack(side='left')

        ### image
        ttk.Label(dev_info, image='mouse').pack(fill='x')

        ### progressbar
        pb = ttk.Progressbar(dev_info, value=66)  # also used as a container for the %␣
↪complete label
        pb.pack(fill='x', pady=5, padx=5)
        ttk.Label(pb, text='66%', style='primary.Invert.TLabel').pack()

        ### progress message
        self.setvar('progress', 'Battery is discharging.')
        ttk.Label(dev_info, textvariable='progress', font='Helvetica 8', anchor='center
↪').pack(fill='x')

        ## licence info ------------------------------------------------------------------
↪--------------------------------
```

(continues on next page)

(continued from previous page)

```
        lic_info = ttk.Labelframe(col1, text='License Info', padding=20)
        lic_info.pack(side='top', fill='both', expand='yes', pady=(10, 0))
        lic_info.rowconfigure(0, weight=1)
        lic_info.columnconfigure(0, weight=2)
        lic_title = ttk.Label(lic_info, text='Trial Version, 28 days left', anchor=
↪'center')
        lic_title.pack(fill='x', pady=(0, 20))
        ttk.Label(lic_info, text='Mouse serial number:', anchor='center', font=
↪'Helvetica 8').pack(fill='x')
        self.setvar('license', 'dtMM2-XYZGHIJKLMN3')
        lic_num = ttk.Label(lic_info, textvariable='license', style='primary.TLabel',␣
↪anchor='center')
        lic_num.pack(fill='x', pady=(0, 20))
        buy_now = ttk.Button(lic_info, image='buy', text='Buy now', compound='bottom',␣
↪command=self.callback)
        buy_now.pack(padx=10, fill='x')

        # Column 2␣
↪==========================================================================================
        col2 = ttk.Frame(self, padding=10)
        col2.grid(row=0, column=1, sticky='news')

        ## scrolling ------------------------------------------------------------------
↪------------------------------
        scrolling = ttk.Labelframe(col2, text='Scrolling', padding=(15, 10))
        scrolling.pack(side='top', fill='both', expand='yes')

        op1 = ttk.Checkbutton(scrolling, text='Scrolling', variable='op1')
        op1.pack(fill='x', pady=5)

        ### no horizontal scrolling
        op2 = ttk.Checkbutton(scrolling, text='No horizontal scrolling', variable='op2')
        op2.pack(fill='x', padx=(20, 0), pady=5)
        ttk.Button(op2, image='question', style='Link.TButton', command=self.callback).
↪pack(side='right')

        ### inverse
        op3 = ttk.Checkbutton(scrolling, text='Inverse scroll directcion vertically',␣
↪variable='op3')
        op3.pack(fill='x', padx=(20, 0), pady=5)
        ttk.Button(op3, image='direction', style='Link.TButton', command=self.callback).
↪pack(side='right')

        ### Scroll only vertical or horizontal
        op4 = ttk.Checkbutton(scrolling, text='Scroll only vertical or horizontal',␣
↪state='disabled')
        op4.configure(variable='op4')
        op4.pack(fill='x', padx=(20, 0), pady=5)

        ### smooth scrolling
        op5 = ttk.Checkbutton(scrolling, text='Smooth scrolling', variable='op5')
        op5.pack(fill='x', padx=(20, 0), pady=5)
```

(continues on next page)

(continued from previous page)

```
        ttk.Button(op5, image='bluetooth', style='Link.TButton', command=self.callback).
↪pack(side='right')

        ### scroll speed
        scroll_speed_frame = ttk.Frame(scrolling)
        scroll_speed_frame.pack(fill='x', padx=(20, 0), pady=5)
        ttk.Label(scroll_speed_frame, text='Speed:').pack(side='left')
        ttk.Scale(scroll_speed_frame, value=35, from_=1, to=100).pack(side='left', fill=
↪'x', expand='yes', padx=5)
        scroll_speed_btn = ttk.Button(scroll_speed_frame, image='reset-small', style=
↪'Link.TButton')
        scroll_speed_btn.configure(command=self.callback)
        scroll_speed_btn.pack(side='left')

        ### scroll sense
        scroll_sense_frame = ttk.Frame(scrolling)
        scroll_sense_frame.pack(fill='x', padx=(20, 0), pady=(5, 0))
        ttk.Label(scroll_sense_frame, text='Sense:').pack(side='left')
        ttk.Scale(scroll_sense_frame, value=50, from_=1, to=100).pack(side='left', fill=
↪'x', expand='yes', padx=5)
        scroll_sense_btn = ttk.Button(scroll_sense_frame, image='reset-small', style=
↪'Link.TButton')
        scroll_sense_btn.configure(command=self.callback)
        scroll_sense_btn.pack(side='left')

        ## 1 finger gestures ------------------------------------------------------------
↪-------------------------------
        finger_gest = ttk.Labelframe(col2, text='1 Finger Gestures', padding=(15, 10))
        finger_gest.pack(side='top', fill='both', expand='yes', pady=(10, 0))

        op6 = ttk.Checkbutton(finger_gest, text='Fast swipe left/right to navigate back/
↪forward', variable='op6')
        op6.pack(fill='x', pady=5)
        ttk.Checkbutton(finger_gest, text='Swap swipe direction', variable='op7').
↪pack(fill='x', padx=(20, 0), pady=5)

        ### gest sense
        gest_sense_frame = ttk.Frame(finger_gest)
        gest_sense_frame.pack(fill='x', padx=(20, 0), pady=(5, 0))

        ttk.Label(gest_sense_frame, text='Sense:').pack(side='left')

        ttk.Scale(gest_sense_frame, value=50, from_=1, to=100).pack(side='left', fill='x
↪', expand='yes', padx=5)

        gest_sense_btn = ttk.Button(gest_sense_frame, image='reset-small', style='Link.
↪TButton')
        gest_sense_btn.configure(command=self.callback)
        gest_sense_btn.pack(side='left')

        ## middle click ----------------------------------------------------------------
↪-------------------------------
```

(continues on next page)

(continued from previous page)

```
        middle_click = ttk.Labelframe(col2, text='Middle Click', padding=(15, 10))
        middle_click.pack(side='top', fill='both', expand='yes', pady=(10, 0))

        cbo = ttk.Combobox(middle_click, values=['Any 2 finger click', 'Other 1', 'Other␣
↪2'])
        cbo.current(0)
        cbo.pack(fill='x')

        # Column 3␣
↪==========================================================================================
        col3 = ttk.Frame(self, padding=10)
        col3.grid(row=0, column=2, sticky='news')

        ## two finger gestures ----------------------------------------------------------
↪-------------------------------
        two_finger_gest = ttk.Labelframe(col3, text='2 Finger Gestures', padding=10)
        two_finger_gest.pack(side='top', fill='both')

        op7 = ttk.Checkbutton(two_finger_gest, text='Fast swipe left/right to navigate␣
↪back/forward', variable='op7')
        op7.pack(fill='x', pady=5)

        op8 = ttk.Checkbutton(two_finger_gest, text='Swap swipe direction', variable='op8
↪')
        op8.pack(fill='x', padx=(20, 0), pady=5)

        ### gest sense
        gest_sense_frame = ttk.Frame(two_finger_gest)
        gest_sense_frame.pack(fill='x', padx=(20, 0), pady=(5, 0))

        ttk.Label(gest_sense_frame, text='Sense:').pack(side='left')

        ttk.Scale(gest_sense_frame, value=50, from_=1, to=100).pack(side='left', fill='x
↪', expand='yes', padx=5)

        gest_sense_btn = ttk.Button(gest_sense_frame, image='reset-small', style='Link.
↪TButton')
        gest_sense_btn.configure(command=self.callback)
        gest_sense_btn.pack(side='left')

        ### fast two finger swipe down
        ttk.Label(two_finger_gest, text='On fast 2 finger up/down swipe:').pack(fill='x',
↪ pady=(10, 5))

        op9 = ttk.Checkbutton(two_finger_gest, text='Swap swipe direction', variable='op9
↪')
        op9.pack(fill='x', padx=(20, 0), pady=5)

        op10 = ttk.Checkbutton(two_finger_gest, text='Swap swipe direction', variable=
↪'op10')
        op10.pack(fill='x', padx=(20, 0), pady=5)
```

(continues on next page)

(continued from previous page)

```
        two_finger_cbo = ttk.Combobox(two_finger_gest, values=['Cycle Task View | Normal␣
↪| Desktop View'])
        two_finger_cbo.current(0)
        two_finger_cbo.pack(fill='x', padx=(20, 0), pady=5)

        ### two finger sense
        two_finger_sense_frame = ttk.Frame(two_finger_gest)
        two_finger_sense_frame.pack(fill='x', padx=(20, 0), pady=(5, 0))

        ttk.Label(two_finger_sense_frame, text='Sense:').pack(side='left')

        ttk.Scale(two_finger_sense_frame, value=50, from_=1, to=100).pack(side='left',␣
↪fill='x', expand='yes', padx=5)

        two_finger_sense_btn = ttk.Button(two_finger_sense_frame, image='reset-small',␣
↪style='Link.TButton')
        two_finger_sense_btn.configure(command=self.callback)
        two_finger_sense_btn.pack(side='left')

        ## mouse options ----------------------------------------------------------------
↪-------------------------------
        mouse_options = ttk.Labelframe(col3, text='2 Finger Gestures', padding=(15, 10))
        mouse_options.pack(side='top', fill='both', expand='yes', pady=(10, 0))

        op11 = ttk.Checkbutton(mouse_options, text='Ignore input if mouse if lifted',␣
↪variable='op11')
        op11.pack(fill='x', pady=5)

        op12 = ttk.Checkbutton(mouse_options, text='Ignore input if mouse if lifted',␣
↪variable='op12')
        op12.pack(fill='x', pady=5)

        op13 = ttk.Checkbutton(mouse_options, text='Ignore input if mouse if lifted',␣
↪variable='op13')
        op13.pack(fill='x', pady=5)

        ### base speed
        base_speed_sense_frame = ttk.Frame(mouse_options)
        base_speed_sense_frame.pack(fill='x', padx=(20, 0), pady=(5, 0))

        ttk.Label(base_speed_sense_frame, text='Base speed:').pack(side='left')

        ttk.Scale(base_speed_sense_frame, value=50, from_=1, to=100).pack(side='left',␣
↪fill='x', expand='yes', padx=5)

        base_speed_sense_btn = ttk.Button(base_speed_sense_frame, image='reset-small',␣
↪style='Link.TButton')
        base_speed_sense_btn.configure(command=self.callback)
        base_speed_sense_btn.pack(side='left')

        # turn on all checkbuttons
        for i in range(1, 14):
```

(continues on next page)

(continued from previous page)

```
            self.setvar(f'op{i}', 1)

        # turn off select buttons
        for j in [2, 9, 12, 13]:
            self.setvar(f'op{j}', 0)

    def callback(self):
        """Demo callback"""
        showinfo(title='Button callback', message="You pressed a button.")


if __name__ == '__main__':
    Application().mainloop()
```

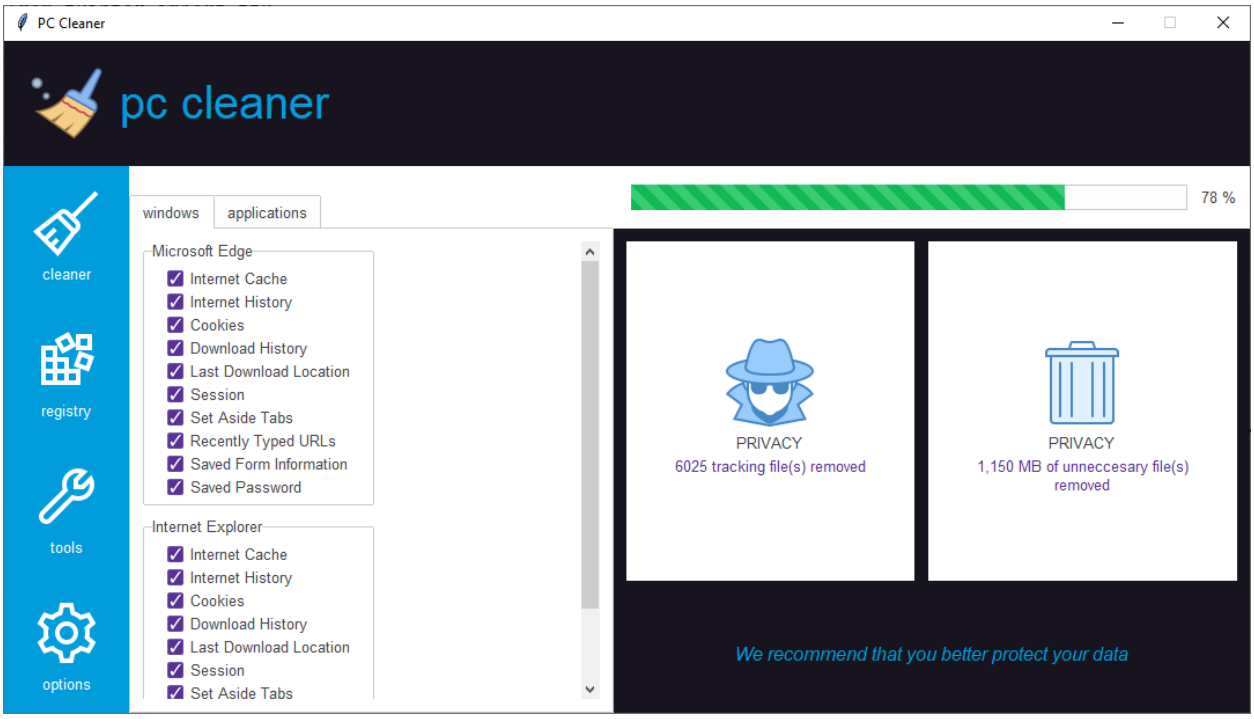
3.5 PC Cleaner

In this example, I demonstrate how to use various styles to build a UI for a PC Cleaner application. This is adapted from an image you can find here. The overall theme is *pulse*. This application includes several widget styles including a custom header style which is configured in the `init` method that changes the background and foreground colors from theme colors available in the `Style.colors` property.

Action buttons `info.TButton`

Progressbar `success.Striped.Horizontal.TProgressbar`

There is a `secondary.TButton` style applied to the result card frames. This gives the cards the same format as a button for any attributes they share. This effectively gives it a highlight color and hover effect. Additionally, by putting another label or card inside with padding around, you can create a border effect, with the card background serving as the border. By increasing the internal padding, you can effectively increase the border size.



Run this code live on repl.it

```
"""
    Author: Israel Dryer
    Modified: 2021-04-09
    Adapted from: https://images.idgesg.net/images/article/2018/08/cw_win10_utilities_ss_
↪02-100769136-orig.jpg
"""
import tkinter
from tkinter import ttk
from pathlib import Path
from ttkbootstrap import Style


class Application(tkinter.Tk):

    def __init__(self):
        super().__init__()
        self.title('PC Cleaner')
        self.style = Style('pulse')
        self.cleaner = Cleaner(self)
        self.cleaner.pack(fill='both', expand='yes')

        # custom styles
        self.style.configure('header.TLabel', background=self.style.colors.secondary,␣
↪foreground=self.style.colors.info)

        # do not allow window resizing
        self.resizable(False, False)


class Cleaner(ttk.Frame):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        # application images
        p = Path(__file__).parent
        self.logo_img = tkinter.PhotoImage(name='logo', file=p/'assets/icons8_broom_64px_
↪1.png')
        self.brush_img = tkinter.PhotoImage(name='cleaner', file=p/'assets/icons8_broom_
↪64px.png')
        self.registry_img = tkinter.PhotoImage(name='registry', file=p/'assets/icons8_
↪registry_editor_64px.png')
        self.tools_img = tkinter.PhotoImage(name='tools', file=p/'assets/icons8_wrench_
↪64px.png')
        self.options_img = tkinter.PhotoImage(name='options', file=p/'assets/icons8_
↪settings_64px.png')
        self.privacy_img = tkinter.PhotoImage(name='privacy', file=p/'assets/icons8_spy_
↪80px.png')
        self.junk_img = tkinter.PhotoImage(name='junk', file=p/'assets/icons8_trash_can_
↪80px.png')
        self.protect_img = tkinter.PhotoImage(name='protect', file=p/'assets/icons8_
↪protect_40px.png')
```

(continues on next page)

(continued from previous page)

```
        # header
        header_frame = ttk.Frame(self, padding=20, style='secondary.TFrame')
        header_frame.grid(row=0, column=0, columnspan=3, sticky='ew')
        ttk.Label(header_frame, image='logo', style='header.TLabel').pack(side='left')
        logo_text = ttk.Label(header_frame, text='pc cleaner', font=('TkDefaultFixed',␣
↪30), style='header.TLabel')
        logo_text.pack(side='left', padx=10)

        # action buttons
        action_frame = ttk.Frame(self)
        action_frame.grid(row=1, column=0, sticky='nsew')
        cleaner_btn = ttk.Button(action_frame, image='cleaner', text='cleaner', compound=
↪'top', style='info.TButton')
        cleaner_btn.pack(side='top', fill='both', ipadx=10, ipady=10)
        registry_btn = ttk.Button(action_frame, image='registry', text='registry',␣
↪compound='top', style='info.TButton')
        registry_btn.pack(side='top', fill='both', ipadx=10, ipady=10)
        tools_btn = ttk.Button(action_frame, image='tools', text='tools', compound='top',
↪ style='info.TButton')
        tools_btn.pack(side='top', fill='both', ipadx=10, ipady=10)
        options_btn = ttk.Button(action_frame, image='options', text='options', compound=
↪'top', style='info.TButton')
        options_btn.pack(side='top', fill='both', ipadx=10, ipady=10)

        # option notebook
        notebook = ttk.Notebook(self)
        notebook.grid(row=1, column=1, sticky='nsew', pady=(25, 0))

        ## windows tab
        windows_tab = ttk.Frame(notebook, padding=10)
        wt_scrollbar = tkinter.Scrollbar(windows_tab)
        wt_scrollbar.pack(side='right', fill='y')
        wt_canvas = tkinter.Canvas(windows_tab, border=0, highlightthickness=0,␣
↪yscrollcommand=wt_scrollbar.set)
        wt_canvas.pack(side='left', fill='both')

        ### adjust the scrollregion when the size of the canvas changes
        wt_canvas.bind('<Configure>', lambda e: wt_canvas.configure(scrollregion=wt_
↪canvas.bbox('all')))
        wt_scrollbar.configure(command=wt_canvas.yview)
        scroll_frame = ttk.Frame(wt_canvas)
        wt_canvas.create_window((0, 0), window=scroll_frame, anchor='nw')

        radio_options = [
            'Internet Cache', 'Internet History', 'Cookies', 'Download History', 'Last␣
↪Download Location',
            'Session', 'Set Aside Tabs', 'Recently Typed URLs', 'Saved Form Information',
↪ 'Saved Password']

        edge = ttk.Labelframe(scroll_frame, text='Microsoft Edge', padding=(20, 5))
        edge.pack(fill='both')
```

(continues on next page)

(continued from previous page)

```
        explorer = ttk.Labelframe(scroll_frame, text='Internet Explorer', padding=(20,␣
↪5))
        explorer.pack(fill='both', pady=10)

        ### add radio buttons to each label frame section
        for section in [edge, explorer]:
            for opt in radio_options:
                cb = ttk.Checkbutton(section, text=opt, state='normal')
                cb.invoke()
                cb.pack(side='top', pady=2, fill='x')
        notebook.add(windows_tab, text='windows')

        ## empty tab for looks
        notebook.add(ttk.Frame(notebook), text='applications')

        # results frame
        results_frame = ttk.Frame(self)
        results_frame.grid(row=1, column=2, sticky='nsew')

        ## progressbar with text indicator
        pb_frame = ttk.Frame(results_frame, padding=(0, 10, 10, 10))
        pb_frame.pack(side='top', fill='x', expand='yes')
        pb = ttk.Progressbar(pb_frame, style='success.Striped.Horizontal.TProgressbar',␣
↪variable='progress')
        pb.pack(side='left', fill='x', expand='yes', padx=(15, 10))
        ttk.Label(pb_frame, text='%').pack(side='right')
        ttk.Label(pb_frame, textvariable='progress').pack(side='right')
        self.setvar('progress', 78)

        ## result cards
        cards_frame = ttk.Frame(results_frame, name='cards-frame', style='secondary.
↪TFrame')
        cards_frame.pack(fill='both', expand='yes')

        ### privacy card
        priv_card = ttk.Frame(cards_frame, padding=1, style='secondary.TButton')
        priv_card.pack(side='left', fill='both', padx=(10, 5), pady=10)
        priv_container = ttk.Frame(priv_card, padding=40)
        priv_container.pack(fill='both', expand='yes')
        priv_lbl = ttk.Label(priv_container, image='privacy', text='PRIVACY', compound=
↪'top', anchor='center')
        priv_lbl.pack(fill='both', padx=20, pady=(40, 0))
        ttk.Label(priv_container, textvariable='priv_lbl', style='primary.TLabel').
↪pack(pady=(0, 20))
        self.setvar('priv_lbl', '6025 tracking file(s) removed')

        ### junk card
        junk_card = ttk.Frame(cards_frame, padding=1, style='secondary.TButton')
        junk_card.pack(side='left', fill='both', padx=(5, 10), pady=10)
        junk_container = ttk.Frame(junk_card, padding=40)
        junk_container.pack(fill='both', expand='yes')
```

(continues on next page)

(continued from previous page)

```
        junk_lbl = ttk.Label(junk_container, image='junk', text='PRIVACY', compound='top
↪', anchor='center')
        junk_lbl.pack(fill='both', padx=20, pady=(40, 0))
        ttk.Label(junk_container, textvariable='junk_lbl', style='primary.TLabel',␣
↪justify='center').pack(pady=(0, 20))
        self.setvar('junk_lbl', '1,150 MB of unneccesary file(s)\nremoved')

        ## user notification
        note_frame = ttk.Frame(results_frame, style='secondary.TFrame', padding=40)
        note_frame.pack(fill='both')
        note_msg = ttk.Label(note_frame, text='We recommend that you better protect your␣
↪data', anchor='center',
                             style='header.TLabel', font=('Helvetica', 12, 'italic'))
        note_msg.pack(fill='both')


if __name__ == '__main__':
    Application().mainloop()
```

3.6 Equalizer

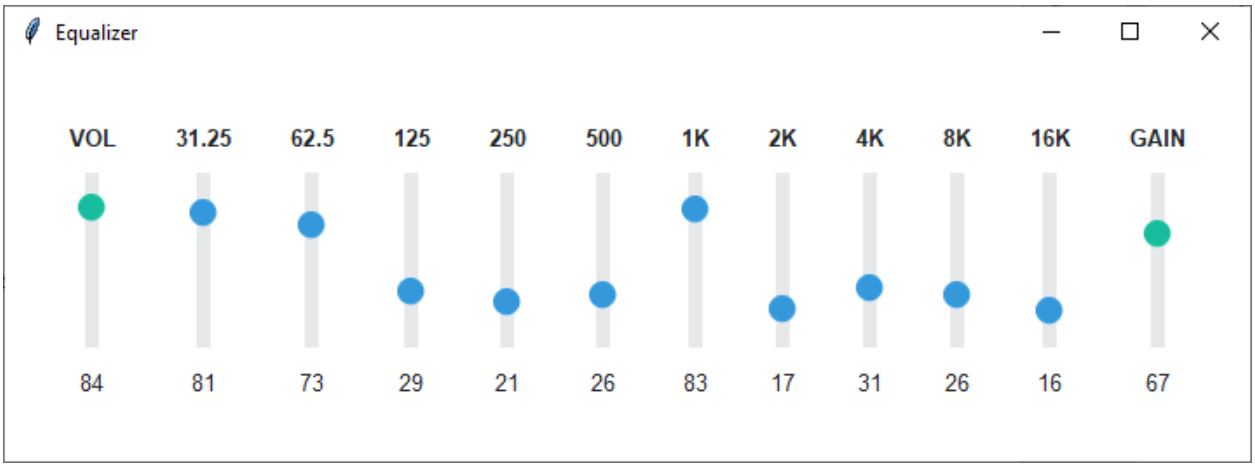
This example demonstrates the use of styles to differentiate scale or "slider" functions. The `ttk.Scale` widget is one of several that include orientation in the style class. The overall theme is **flatly** and the following styles are applied to the widgets to create contrast:

Volume `success.Vertical.TScale`

Gain `success.Vertical.TScale`

Other `info.Vertical.TScale`

Now for some comments on the code: Because I wanted the scale value to be reflected in a label below the scale, this application is a lot more complicated than it really needs to be due to some oddities of the ttk scale implementation. The `ttk.Scale` widget outputs a double type, which means that in order to display a nice rounded integer, that number has to be converted when updated. Fortunately, the scale widget has a `command` parameter for setting a callback. The callback will get the scale value, which can then be converted into a nice clean format.



Note: For a vertical orientation, the `from_` parameter corresponds to the top and `to` corresponds to the bottom of the widget, so you'll need to take this into account when you set the minimum and maximum numbers for your scale range.

Run this code live on repl.it

```
"""
    Author: Israel Dryer
    Modified: 2021-04-07
"""
import tkinter
from random import randint
from tkinter import ttk
from ttkbootstrap import Style


class Application(tkinter.Tk):

    def __init__(self):
        super().__init__()
        self.title('Equalizer')
        self.style = Style()
        self.eq = Equalizer(self)
        self.eq.pack(fill='both', expand='yes')


class Equalizer(ttk.Frame):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.configure(padding=20)
        controls = ['VOL', '31.25', '62.5', '125', '250', '500', '1K', '2K', '4K', '8K',
↪'16K', 'GAIN']

        # create band widgets
        for c in controls:
            # starting random value
            value = randint(1, 99)
            self.setvar(c, value)

            # container
            frame = ttk.Frame(self, padding=5)
            frame.pack(side='left', fill='y', padx=10)

            # header
            ttk.Label(frame, text=c, anchor='center', font=('Helvetica 10 bold')).
↪pack(side='top', fill='x', pady=10)

            # slider
            scale = ttk.Scale(frame, orient='vertical', from_=99, to=1, value=value)
            scale.pack(fill='y')
            scale.configure(command=lambda val, name=c: self.setvar(name, f'{float(val):.
↪0f}'))
```

(continues on next page)

(continued from previous page)

```
            # set slider style
            scale.configure(style='success.Vertical.TScale' if c in ['VOL', 'GAIN'] else
↪'info.Vertical.TScale')

            # slider value label
            ttk.Label(frame, textvariable=c).pack(pady=10)


if __name__ == '__main__':
    Application().mainloop()
```

3.7 Collapsing Frame

This example demonstrates how to build a collapsing frame widget. Each frame added to the widget can be assigned a title and style. The overall theme is **flatly** and various widget styles are applied to distinguish the option groups.

option group 1 `primary.TFrame`

option group 2 `danger.TFrame`

option group 3 `success.TFrame`

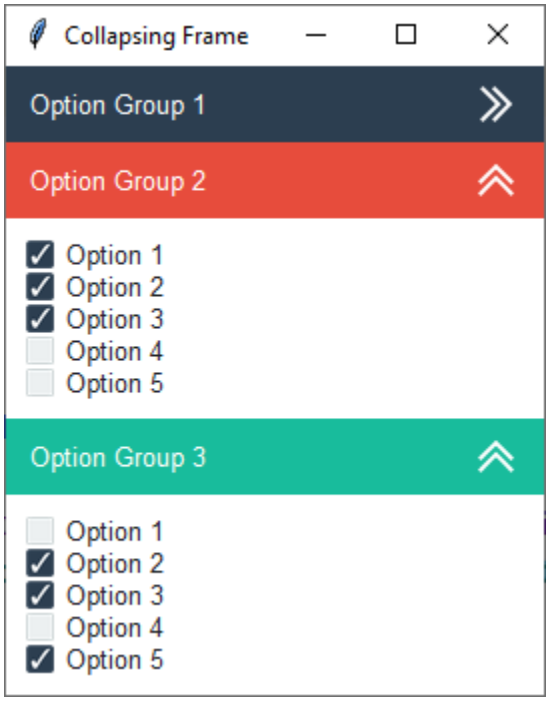
The collapse functionality is created by removing contents of the child frame and then adding it again with the grid manager. The toggle checks to see if the contents is visible on the screen, and if not, will add the contents back with the grid manager, otherwise, it will all be removed. This is all done in the `_toggle_open_close` method. Additionally the button image alternates from *open* to *closed* to give a visual hint about the child frame state.

A style argument can be passed into the widget constructor to change the widget header color. The constructor extracts the color from the style class and applies it internally to color the header, using a label class for the title, and a button class for the button. The `primary.Invert.TLabel` class inverts the foreground and background colors of the standard `primary.TLabel` style so that the background shows the primary color, similar to a button.

Run this code live on repl.it

```
"""
    Author: Israel Dryer
    Modified: 2021-04-08
"""
import tkinter
from tkinter import ttk
from pathlib import Path

from ttkbootstrap import Style


class Application(tkinter.Tk):

    def __init__(self):
        super().__init__()
        self.title('Collapsing Frame')
        self.style = Style()
```

(continues on next page)



(continued from previous page)

```
        cf = CollapsingFrame(self)
        cf.pack(fill='both')

        # option group 1
        group1 = ttk.Frame(cf, padding=10)
        for x in range(5):
            ttk.Checkbutton(group1, text=f'Option {x + 1}').pack(fill='x')
        cf.add(group1, title='Option Group 1', style='primary.TButton')

        # option group 2
        group2 = ttk.Frame(cf, padding=10)
        for x in range(5):
            ttk.Checkbutton(group2, text=f'Option {x + 1}').pack(fill='x')
        cf.add(group2, title='Option Group 2', style='danger.TButton')

        # option group 3
        group3 = ttk.Frame(cf, padding=10)
        for x in range(5):
            ttk.Checkbutton(group3, text=f'Option {x + 1}').pack(fill='x')
        cf.add(group3, title='Option Group 3', style='success.TButton')


class CollapsingFrame(ttk.Frame):
    """
    A collapsible frame widget that opens and closes with a button click.
    """

    def __init__(self, *args, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        super().__init__(*args, **kwargs)
        self.columnconfigure(0, weight=1)
        self.cumulative_rows = 0
        p = Path(__file__).parent
        self.images = [tkinter.PhotoImage(name='open', file=p/'assets/icons8_double_up_
↪24px.png'),
                       tkinter.PhotoImage(name='closed', file=p/'assets/icons8_double_
↪right_24px.png')]

    def add(self, child, title="", style='primary.TButton', **kwargs):
        """Add a child to the collapsible frame

        :param ttk.Frame child: the child frame to add to the widget
        :param str title: the title appearing on the collapsible section header
        :param str style: the ttk style to apply to the collapsible section header
        """
        if child.winfo_class() != 'TFrame':  # must be a frame
            return
        style_color = style.split('.')[0]
        frm = ttk.Frame(self, style=f'{style_color}.TFrame')
        frm.grid(row=self.cumulative_rows, column=0, sticky='ew')

        # header title
        lbl = ttk.Label(frm, text=title, style=f'{style_color}.Invert.TLabel')
        if kwargs.get('textvariable'):
            lbl.configure(textvariable=kwargs.get('textvariable'))
        lbl.pack(side='left', fill='both', padx=10)

        # header toggle button
        btn = ttk.Button(frm, image='open', style=style, command=lambda c=child: self._
↪toggle_open_close(child))
        btn.pack(side='right')

        # assign toggle button to child so that it's accesible when toggling (need to␣
↪change image)
        child.btn = btn
        child.grid(row=self.cumulative_rows + 1, column=0, sticky='news')

        # increment the row assignment
        self.cumulative_rows += 2

    def _toggle_open_close(self, child):
        """
        Open or close the section and change the toggle button image accordingly

        :param ttk.Frame child: the child element to add or remove from grid manager
        """
        if child.winfo_viewable():
            child.grid_remove()
            child.btn.configure(image='closed')
        else:
            child.grid()
```

(continues on next page)

(continued from previous page)

```
            child.btn.configure(image='open')


if __name__ == '__main__':
    Application().mainloop()
```

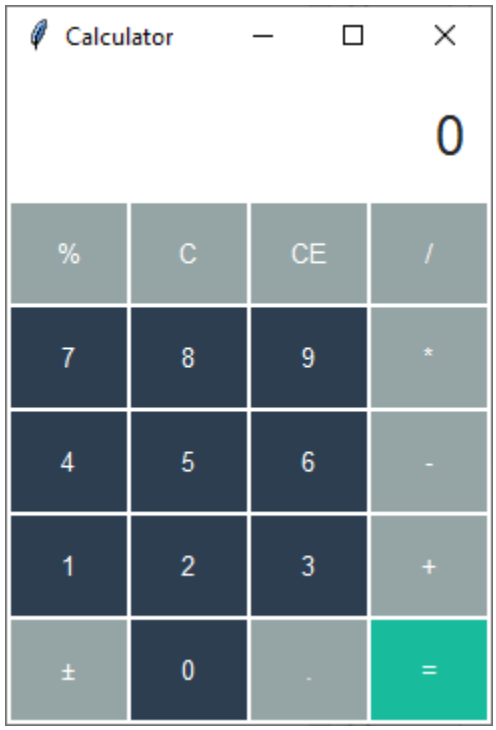
3.8 Calculator

This basic calculator demonstrates how to use color styles to differentiate button functions. The overall theme is **flatly** and the following styles are applied to the widgets:

Digits `primary.TButton`

Operators `secondary.TButton`

Equals `success.TButton`



Run this code live on repl.it

```
"""
    Author: Israel Dryer
    Modified: 2021-04-09
"""
import tkinter
from tkinter import ttk

from ttkbootstrap import Style
```

(continues on next page)

(continued from previous page)

```
class Application(tkinter.Tk):

    def __init__(self):
        super().__init__()
        self.title('Calculator')
        self.style = Style('flatly')
        self.style.configure('.', font='TkFixedFont 16')
        self.calc = Calculator(self)
        self.calc.pack(fill='both', expand='yes')


class Calculator(ttk.Frame):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.configure(padding=1)

        # number display
        self.display_var = tkinter.StringVar(value=0)
        self.display = ttk.Label(self, textvariable=self.display_var, font='TkFixedFont␣
↪20', anchor='e')
        self.display.grid(row=0, column=0, columnspan=4, sticky='ew', pady=15, padx=10)

        # button layout
        button_matrix = [
            ('%', 'C', 'CE', '/'), (7, 8, 9, '*'), (4, 5, 6, '-'), (1, 2, 3, '+'), ('±',
↪ 0, '.', '=')]

        # create buttons with various styling
        for i, row in enumerate(button_matrix):
            for j, lbl in enumerate(row):
                if isinstance(lbl, int):
                    btn = ttk.Button(self, text=lbl, width=2, style='primary.TButton')
                elif lbl == '=':
                    btn = ttk.Button(self, text=lbl, width=2, style='success.TButton')
                else:
                    btn = ttk.Button(self, text=lbl, width=2, style='secondary.TButton')
                btn.grid(row=i + 1, column=j, sticky='nsew', padx=1, pady=1, ipadx=10,␣
↪ipady=10)

                # bind button press
                btn.bind("<Button-1>", self.press_button)

        # variables used for collecting button input
        self.position_left = ''
        self.position_right = '0'
        self.position_is_left = True
        self.running_total = 0.0

    def press_button(self, event):
        value = event.widget['text']
```

(continues on next page)

(continued from previous page)

```
        if isinstance(value, int):
            if self.position_is_left:
                self.position_left = f'{self.position_left}{value}'
            else:
                self.position_right = str(value) if self.position_right == '0' else f'
↪{self.position_right}{value}'
        elif value == '.':
            self.position_is_left = False
        elif value in ['/', '-', '+', '*']:
            self.operator = value
            self.running_total = float(self.display_var.get())
            self.reset_variables()
        elif value == '=':
            operation = f'{self.running_total}{self.operator}{self.display_var.get()}'
            result = eval(operation)
            self.display_var.set(result)
            return
        elif value in ['CE', 'C']:
            self.reset_variables()
            self.operator = None
            self.running_total = 0
            return

        # update the number display
        self.display_var.set('.'.join([self.position_left, self.position_right]))

    def reset_variables(self):
        self.display_var.set(0)
        self.position_is_left = True
        self.position_left = ''
        self.position_right = '0'


if __name__ == '__main__':
    Application().mainloop()
```

3.9 Simple Data Entry Form

This simple data entry form accepts user input and then prints it to the screen when submitted. The overall theme is **flatly** on the first example and **darkly** on the second, with the following styles applied to specific widgets:

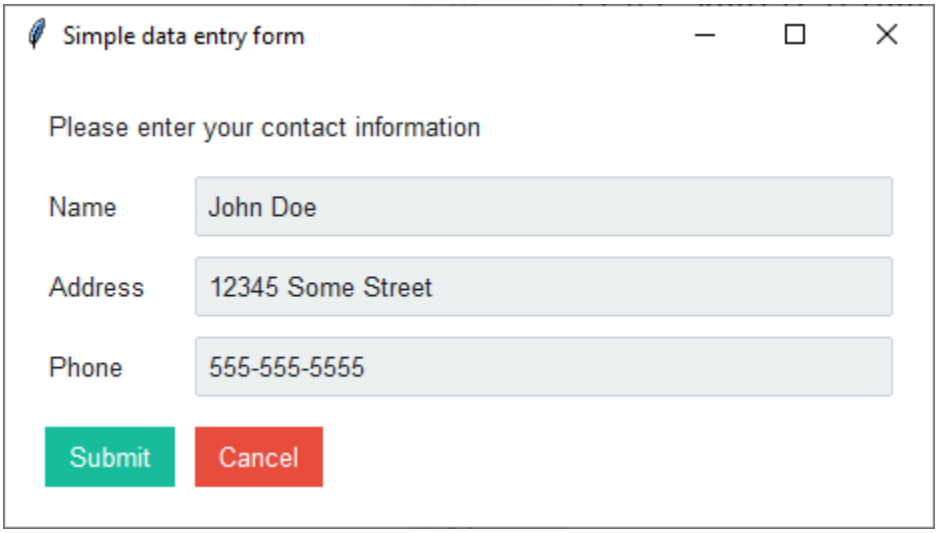
Submit `style="info.TButton"`

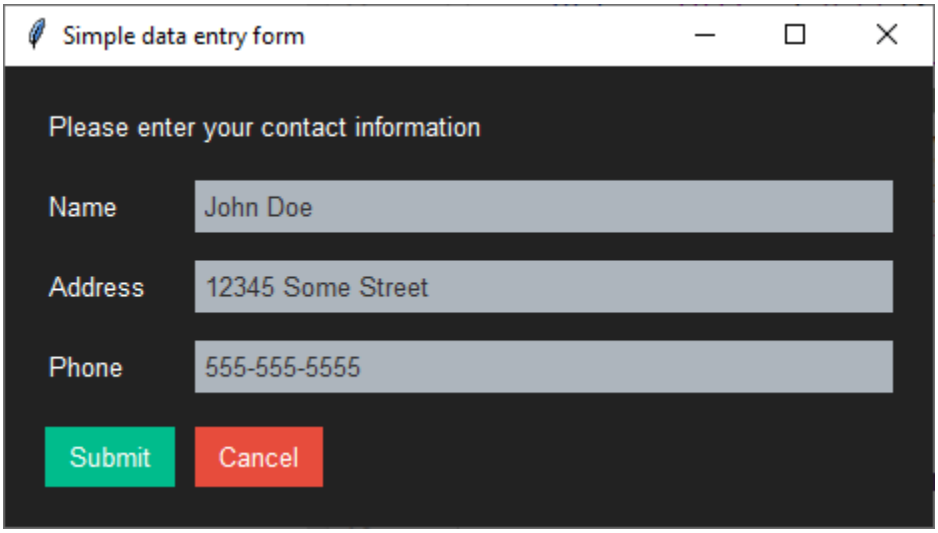
Cancel `style="danger.TButton"`

Run this code live on repl.it

```
"""
    Author: Israel Dryer
    Modified: 2021-04-07
```

(continues on next page)


Simple data entry form
Please enter your contact information
Name
John Doe
Address
12345 Some Street
Phone
555-555-5555
Submit
Cancel


Simple data entry form
Please enter your contact information
Name
John Doe
Address
12345 Some Street
Phone
555-555-5555
Submit
Cancel

(continued from previous page)

```
"""
import tkinter
from tkinter import ttk

from ttkbootstrap import Style


class Application(tkinter.Tk):

    def __init__(self):
        super().__init__()
        self.title('Simple data entry form')
        self.style = Style('darkly')
        self.form = EntryForm(self)
        self.form.pack(fill='both', expand='yes')


class EntryForm(ttk.Frame):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.configure(padding=(20, 10))
        self.columnconfigure(2, weight=1)

        # form variables
        self.name = tkinter.StringVar(value='', name='name')
        self.address = tkinter.StringVar(value='', name='address')
        self.phone = tkinter.StringVar(value='', name='phone')

        # form headers
        ttk.Label(self, text='Please enter your contact information', width=60).
↪grid(columnspan=3, pady=10)

        # create label/entry rows
        for i, label in enumerate(['name', 'address', 'phone']):
            ttk.Label(self, text=label.title()).grid(row=i + 1, column=0, sticky='ew',␣
↪pady=10, padx=(0, 10))
            ttk.Entry(self, textvariable=label).grid(row=i + 1, column=1, columnspan=2,␣
↪sticky='ew')

        # submit button
        self.submit = ttk.Button(self, text='Submit', style='success.TButton',␣
↪command=self.print_form_data)
        self.submit.grid(row=4, column=0, sticky='ew', pady=10, padx=(0, 10))

        # cancel button
        self.cancel = ttk.Button(self, text='Cancel', style='danger.TButton',␣
↪command=self.quit)
        self.cancel.grid(row=4, column=1, sticky='ew')

    def print_form_data(self):
        print(self.name.get(), self.address.get(), self.phone.get())
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    Application().mainloop()
```

3.10 Timer Widget

This simple data entry form accepts user input and then prints it to the screen when submitted. The overall theme is **flatly** and various styles are applied to the buttons depending on state and function:

Start `style="info.TButton"`

Pause `style="info.Outline.TButton"`

Reset `style="success.TButton"`

Exit `style="danger.TButton"`

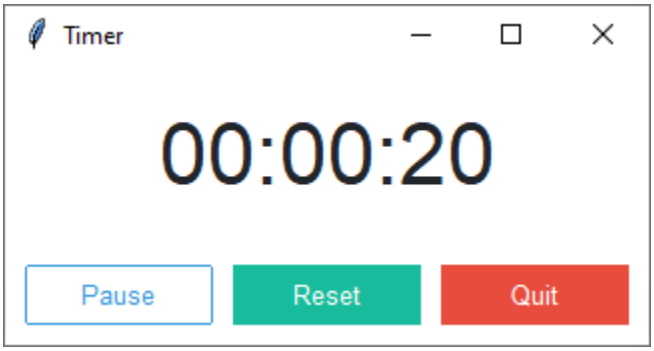


Fig. 1: timer is running

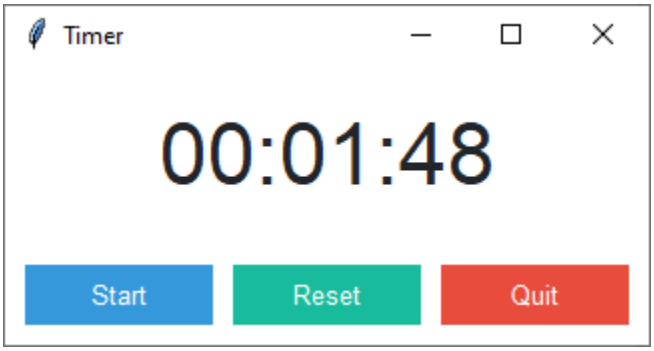


Fig. 2: timer is paused

Run this code live on repl.it

```
"""
    Author: Israe Dryer
    Modified: 2021-04-07
    Adapted for ttkbootstrap from: https://github.com/PySimpleGUI/PySimpleGUI/blob/
↪master/DemoPrograms/Demo_Desktop_Widget_Timer.py
```

(continues on next page)

(continued from previous page)

```
"""
import tkinter
from tkinter import ttk
from ttkbootstrap import Style


class Application(tkinter.Tk):

    def __init__(self):
        super().__init__()
        self.title('Timer')
        self.style = Style()
        self.timer = TimerWidget(self)
        self.timer.pack(fill='both', expand='yes')


class TimerWidget(ttk.Frame):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        # variables
        self.running = tkinter.BooleanVar(value=False)
        self.after_id = tkinter.StringVar()
        self.time_elapsed = tkinter.IntVar()
        self.time_text = tkinter.StringVar(value='00:00:00')

        # timer label
        self.timer_lbl = ttk.Label(self, font='-size 32', anchor='center',␣
↪textvariable=self.time_text)
        self.timer_lbl.pack(side='top', fill='x', padx=60, pady=20)

        # control buttons
        self.toggle_btn = ttk.Button(self, text='Start', width=10, style='info.TButton',␣
↪command=self.toggle)
        self.toggle_btn.pack(side='left', fill='x', expand='yes', padx=10, pady=10)

        self.reset_btn = ttk.Button(self, text='Reset', width=10, style='success.TButton
↪', command=self.reset)
        self.reset_btn.pack(side='left', fill='x', expand='yes', pady=10)

        self.quit_btn = ttk.Button(self, text='Quit', width=10, style='danger.TButton',␣
↪command=self.quit)
        self.quit_btn.pack(side='left', fill='x', expand='yes', padx=10, pady=10)

    def toggle(self):
        if self.running.get():
            self.pause()
            self.running.set(False)
            self.toggle_btn.configure(text='Start', style='info.TButton')
        else:
            self.start()
```

(continues on next page)

(continued from previous page)

```
            self.running.set(True)
            self.toggle_btn.configure(text='Pause', style='info.Outline.TButton')

    def pause(self):
        self.after_cancel(self.after_id.get())

    def start(self):
        self.after_id.set(self.after(1, self.increment))

    def increment(self):
        current = self.time_elapsed.get() + 1
        self.time_elapsed.set(current)
        time_str = '{:02d}:{:02d}:{:02d}'.format((current // 100) // 60, (current //␣
↪100) % 60, current % 100)
        self.time_text.set(time_str)
        self.after_id.set(self.after(100, self.increment))

    def reset(self):
        self.time_elapsed.set(0)
        self.time_text.set('00:00:00')


if __name__ == '__main__':
    Application().mainloop()
```

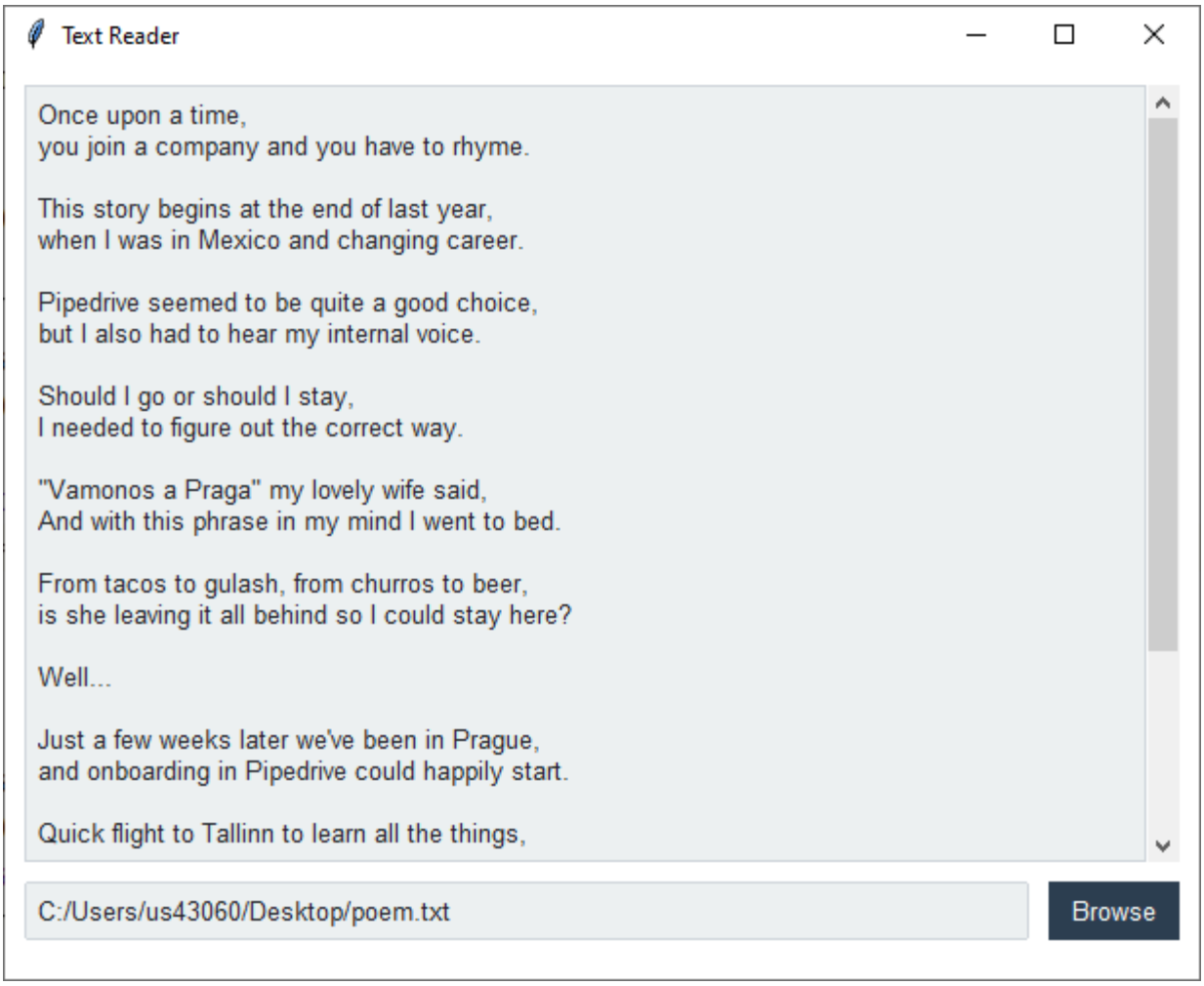
3.11 Text Reader

This application opens a text file and puts the data into a scrolled text widget. The overall theme is **flatly**. The `Style.colors` property was used to adjust the highlight colors on the text widget.

Run this code live on repl.it

```
"""
    Author: Israel Dryer
    Modified: 2021-04-07
"""
import tkinter
from tkinter import ttk
from tkinter.filedialog import askopenfilename
from tkinter.scrolledtext import ScrolledText

from ttkbootstrap import Style


class Application(tkinter.Tk):

    def __init__(self):
        super().__init__()
        self.title('Text Reader')
        self.style = Style()
```

(continues on next page)


Text Reader
Once upon a time,
you join a company and you have to rhyme.
This story begins at the end of last year,
when I was in Mexico and changing career.
Pipedrive seemed to be quite a good choice,
but I also had to hear my internal voice.
Should I go or should I stay,
I needed to figure out the correct way.
"Vamonos a Praga" my lovely wife said,
And with this phrase in my mind I went to bed.
From tacos to gulash, from churros to beer,
is she leaving it all behind so I could stay here?
Well...
Just a few weeks later we've been in Prague,
and onboarding in Pipedrive could happily start.
Quick flight to Tallinn to learn all the things,
C:/Users/us43060/Desktop/poem.txt
Browse

(continued from previous page)

```
        self.reader = Reader(self)
        self.reader.pack(fill='both', expand='yes')


class Reader(ttk.Frame):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.configure(padding=10)
        self.filename = tkinter.StringVar()

        # scrolled text with custom highlight colors
        self.text_area = ScrolledText(self, highlightcolor=self.master.style.colors.
↪primary,
                                      highlightbackground=self.master.style.colors.
↪border, highlightthickness=1)
        self.text_area.pack(fill='both')

        # insert default text in text area
        self.text_area.insert('end', 'Click the browse button to open a new text file.')

        # filepath
        ttk.Entry(self, textvariable=self.filename).pack(side='left', fill='x', expand=
↪'yes', padx=(0, 5), pady=10)

        # browse button
        ttk.Button(self, text='Browse', command=self.open_file).pack(side='right', fill=
↪'x', padx=(5, 0), pady=10)

    def open_file(self):
        path = askopenfilename()
        if not path:
            return

        with open(path, encoding='utf-8') as f:
            self.text_area.delete('1.0', 'end')
            self.text_area.insert('end', f.read())
            self.filename.set(path)


if __name__ == '__main__':
    Application().mainloop()
```

The poem used in this demonstration can be found here.

CHAPTER

FOUR

COOKBOOK

A collection of examples that demonstrate how to use ttk and ttkbootstrap widgets in interesting and useful ways.

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*.

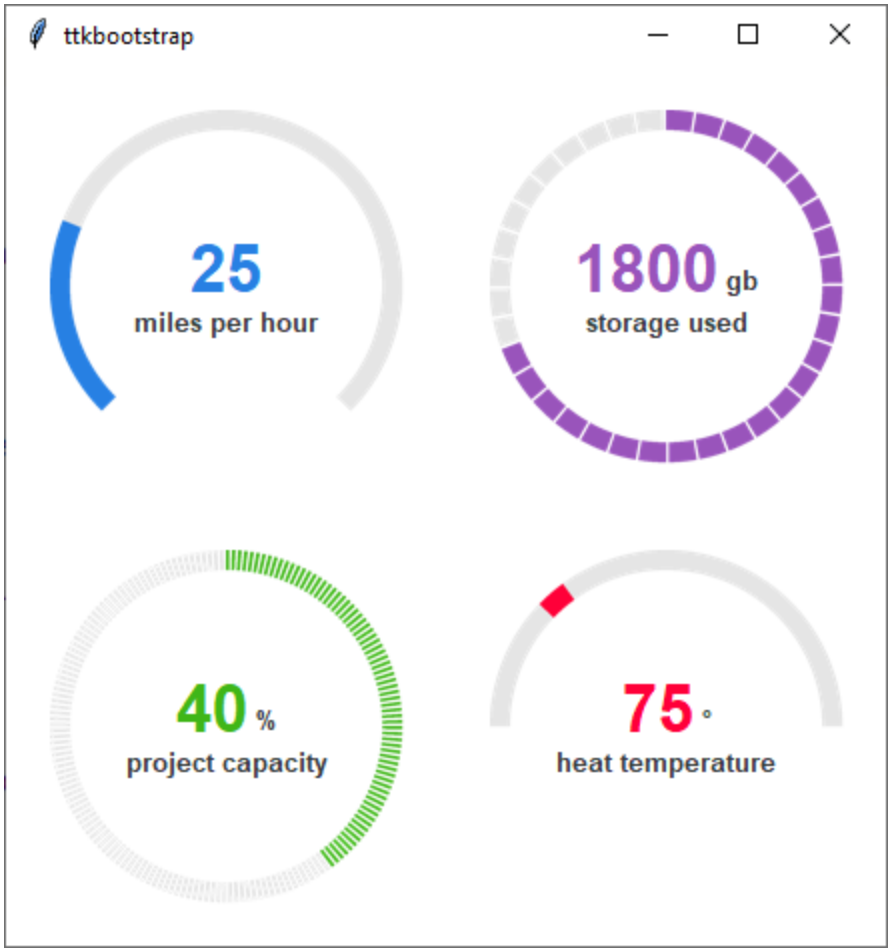
top-right the `stripethickness` is *10* pixels to give it a segmented appearance. The `meterstyle` is *info.TLabel*.

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()
```

CHAPTER

FIVE

REFERENCE

5.1 Module

5.1.1 Colors

class ttkbootstrap.**Colors**(*primary*, *secondary*, *success*, *info*, *warning*, *danger*, *bg*, *fg*, *selectbg*, *selectfg*, *border*, *inputfg*, *inputbg*, *light='#ddd'*, *dark='#333'*)

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.
- **light** (*str*) – a light color
- **dark** (*str*) – a dark color

get(*color_label*)

Lookup a color property

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

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.

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

5.1.3 StylerTTK

class ttkbootstrap.**StylerTTK**(*style*, *definition*)

Bases: object

A class to create a new ttk theme.

Create a new ttk theme by using a combination of built-in themes and some image-based elements using pillow. A theme is generated at runtime and is available to use with the Style class methods. The base theme of all **ttkbootstrap** themes is *clam*. In many cases, widget layouts are re-created using an assortment of elements from various styles such as *clam*, *alt*, *default*, etc…

theme_images

theme assets used for various widgets.

Type dict

settings

settings used to build the actual theme using the theme_create method.

Type dict

styler_tk

an object used to style tkinter widgets (not ttk).

Type StylerTk

theme

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

Type *ThemeDefinition*

Parameters

- **style** (Style) – an instance of ttk.Style.
- **definition** (ThemeDefinition) – an instance of ThemeDefinition; used to create the theme settings.

create_theme()

Create and style a new ttk theme. A wrapper around internal style methods.

update_ttk_theme_settings()
Update the settings dictionary that is used to create a theme. This is a wrapper on all the *_style_widget* methods which define the layout, configuration, and styling mapping for each ttk widget.

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

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.

5.2 Widgets

5.2.1 Button

class ttkbootstrap.widgets.**Button**(*parent=None*, ***kwargs*)

Bases: tkinter.ttk.Button

Ttk button widget, displays as a textual label and/or image, and evaluates a command when pressed.

Parameters **parent** (*Widget*) – The parent widget.

Keyword Arguments

- **class** (*str*) – Specifies the window class. The class is used when querying the option database for the window's other options, to determine the default bindtags for the window, and to select the widget's default layout and style. This is a read-only option; it may only be specified when the window is created, and may not be changed with the configure widget command.
- **compound** (*str*) – Specifies if the widget should display text and bitmaps/images at the same time, and if so, where the bitmap/image should be placed relative to the text. Must be one of the values **none**, **bottom**, **top**, **left**, **right**, or **center**. For example, the (default) value **none** specifies that the bitmap or image should (if defined) be displayed *instead* of the text, the value **left** specifies that the bitmap or image should be displayed to the *left* of the text, and the value **center** specifies that the bitmap or image should be displayed *underneath* the text.
- **cursor** (*str*) – Specifies the mouse cursor to be used for the widget. Names and values will vary according to your operating system. Examples can be found here: https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/cursors.html
- **image** (*PhotoImage or str*) – Specifies an image to display in the widget, which must have been created with `tk.PhotoImage` or *TkPhotoImage* ` if using **pillow**. Can also be a string representing the name of the photo if the photo has been given a name using the `name` parameter. Typically, if the `image` option is specified then it overrides other options that specify a bitmap or textual value to display in the widget, though this is controlled by the `compound` option; the `image` option may be reset to an empty string to re-enable a bitmap or text display.
- **state** (*str*) – May be set to **normal** or **disabled** to control the *disabled* state bit. This is a write-only option; setting it changes the widget state, but the state widget command does not affect the `state` option.
- **style** (*str*) – May be used to specify a custom widget style.
- **takefocus** (*bool*) – Determines whether the window accepts the focus during keyboard traversal (e.g., Tab and Shift-Tab). To remove the widget from focus traversal, use `takefocus=False`.
- **text** (*str*) – Specifies a string to be displayed inside the widget.
- **textvariable** (*StringVar or str*) – Specifies the name of a variable. Use the `StringVar` or the string representation if the variable has been named. The value of the variable is a text string to be displayed inside the widget; if the variable value changes then the widget will automatically update itself to reflect the new value.
- **underline** (*int*) – Specifies the integer index of a character to underline in the widget. This option is used by the default bindings to implement keyboard traversal for menu buttons and menu entries. 0 corresponds to the first character of the text displayed in the widget, 1 to the next character, and so on.

- **width** (*int*) – If the label is text, this option specifies the absolute width of the text area on the button, as a number of characters; the actual width is that number multiplied by the average width of a character in the current font. For image labels, this option is ignored. The option may also be configured in a style.
- **command** (*func*) – A callback function to evaluate when the widget is invoked.

invoke()
Invokes the command associated with the button.

5.2.2 Calendar

Classes and functions that enable the user to select a date.

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

Return type List

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*

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.

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.
- **`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.

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.

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.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

R

S

T

U

W