

Chapter 3: GUI

Look And Feel Customization

Chapter 3: **Look and Feel Customization**

In this chapter, We will also **introduce a few new widgets that tkinter** offers us.

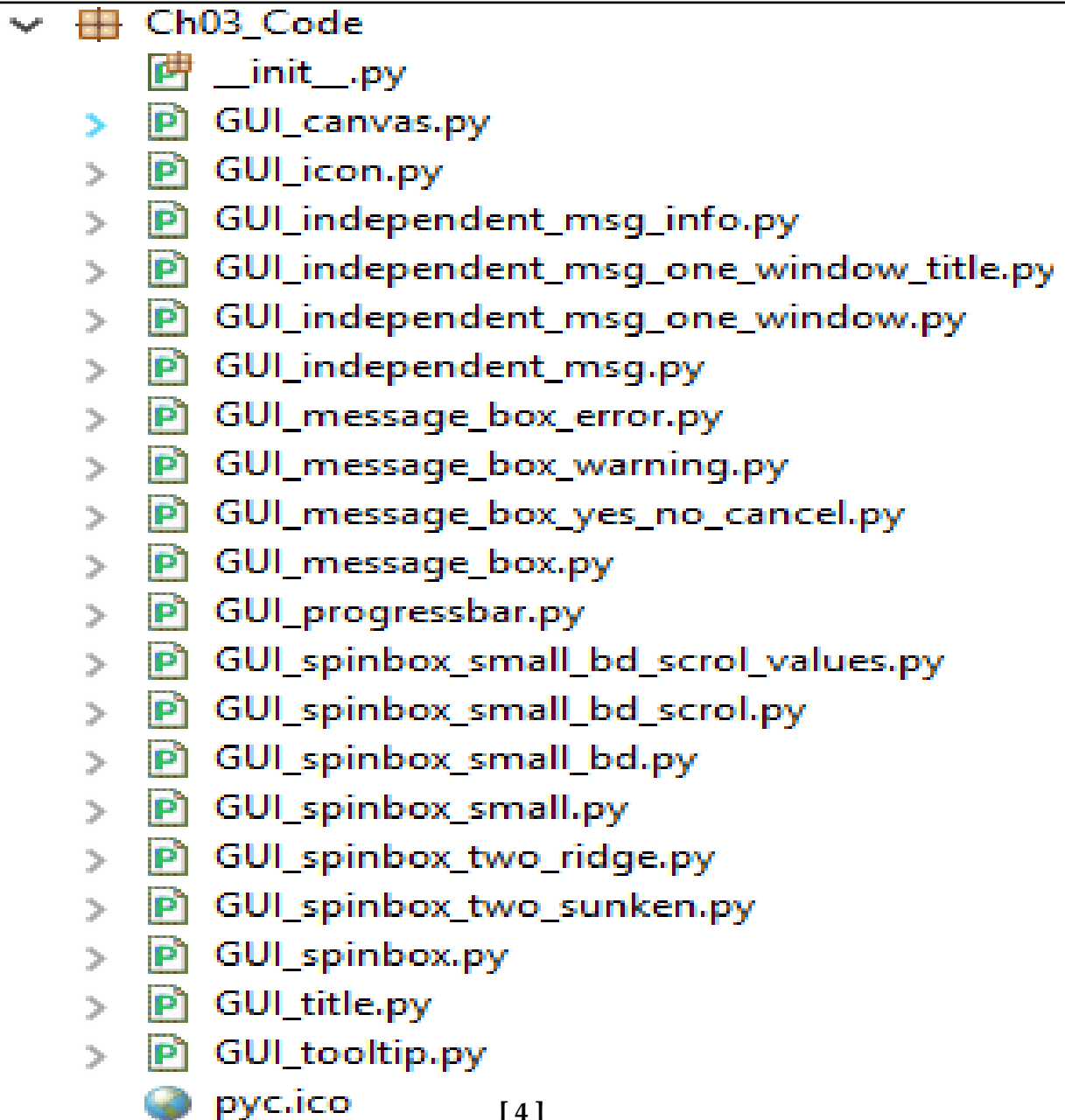
In the *Creating tooltips using Python* recipe, we will create a ToolTip OOP-style class.

You will learn how to create different message boxes, change the GUI window title, and much more. We will be using a spin box control to learn how to apply different styles.

Content

- 1. Creating message boxes – the information, warning, and error**
- 2. Create independent message boxes**
- 3. Create the title of a tkinter window form**
- 4. Changing the icon of the main root window**
- 5. Using a spin box control**
- 6. Applying relief – sunken and raised appearance of widgets**
- 7. Creating tooltips using Python**
- 8. Adding Progressbar to the GUI**
- 9. Use the canvas widget**

Here is the overview of the Python modules for this chapter:



Ch03_Code

- __init__.py
- > GUI_canvas.py
- > GUI_icon.py
- > GUI_independent_msg_info.py
- > GUI_independent_msg_one_window_title.py
- > GUI_independent_msg_one_window.py
- > GUI_independent_msg.py
- > GUI_message_box_error.py
- > GUI_message_box_warning.py
- > GUI_message_box_yes_no_cancel.py
- > GUI_message_box.py
- > GUI_progressbar.py
- > GUI_spinbox_small_bd_scrol_values.py
- > GUI_spinbox_small_bd_scrol.py
- > GUI_spinbox_small_bd.py
- > GUI_spinbox_small.py
- > GUI_spinbox_two_ridge.py
- > GUI_spinbox_two_sunken.py
- > GUI_spinbox.py
- > GUI_title.py
- > GUI_tooltip.py
- pyc.ico

[4]

1. Creating message boxes – information, warning, and error

A message box is a pop-up window that gives feedback to the user. It can be informational, hinting at potential problems, as well as catastrophic errors.

Using Python to create message boxes is very easy.

Getting ready

We will add functionality to the **Help | About** menu item we created in **Chapter 2, Layout Management**, in the *Creating tabbed widgets* recipe.

The code is from `GUI_tabbed_all_widgets_both_tabs.py`. The typical feedback to the user when clicking the **Help | About** menu in most applications is informational. We'll start with this information and then vary the design pattern to show warnings and errors.

How to do it...

Here are the steps to follow to create a message box in Python:

1. Open

`GUI_tabbed_all_widgets_both_tabs.py` from Chapter 2, *Layout Management*, and save the module as `GUI_message_box.py`.

2. Add the following line of code to the top of the module where the import statements live:

```
from tkinter import messagebox  
as msg
```

3. Next, create a callback function that will display a message box. We have to place the code of the callback above the code where we attach the callback to the menu item, because this is still procedural and not OOP code.

Add the following code just above the lines where we create the help menu:

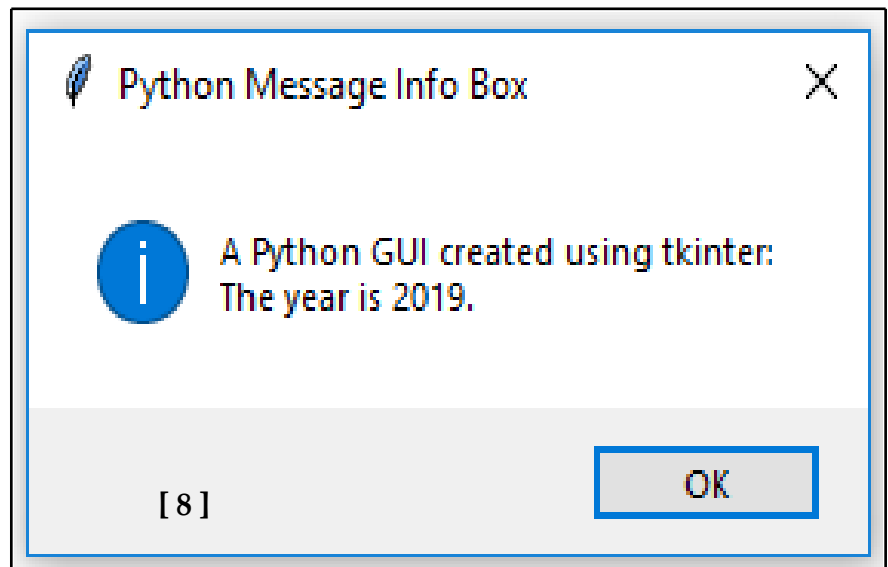
```
def _msgBox():  
    msg.showinfo('Python Message  
    Info Box', 'A Python GUI  
    created using tkinter:\nThe  
    year is 2019.')
```

The preceding instructions produce the following code, `GUI_message_box.py`:

```
# Display a Message Box
def _msgBox():
    msg.showinfo('Python Message Info Box', 'A Python GUI created using tkinter:\n\nThe year is 2019.')

# Add another Menu to the Menu Bar and an item
help_menu = Menu(menu_bar, tearoff=0)
help_menu.add_command(label="About", command=_msgBox) # display messagebox when clicked
menu_bar.add_cascade(label="Help", menu=help_menu)
```

4. Run the code. Clicking **Help** | **About** now causes the following pop-up window to appear:



Let's transform this code into a warning message box pop-up window instead:

1. Open `GUI_message_box.py` and save the module as `GUI_message_box_warning.py`.

2. Comment out the `msg.showinfo` line.

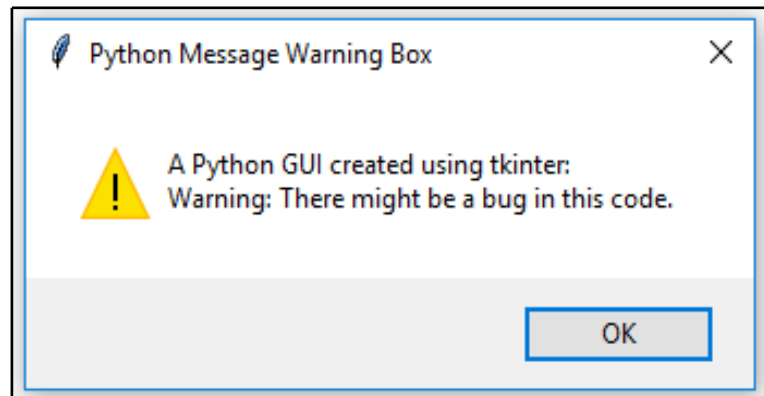
3. Replace the information box code with warning box code:

```
msg.showwarning('Python Message Warning Box',  
'A Python GUI created using tkinter:' '\nWarning:  
There might be a bug in this code.')
```

The preceding instructions produce the following code,
`GUI_message_box_warning.py`:

```
# Display a Message Box
def _msgBox():
#     msg.showinfo('Python Message Info Box', 'A Python GUI created using tkinter:'
#                 '\nThe year is 2019.')
#     msg.showwarning('Python Message Warning Box', 'A Python GUI created using tkinter:'
#                   '\nWarning: There might be a bug in this code.')
```

4. Running the preceding code will now result in the following slightly modified message box:



Displaying an error message box is simple and usually warns the user of a serious problem. As we did in the previous code snippet, comment out the previous line and add the following code, as we have done here:

1. Open `GUI_message_box_warning.py` and save the module as `GUI_message_box_error.py`.

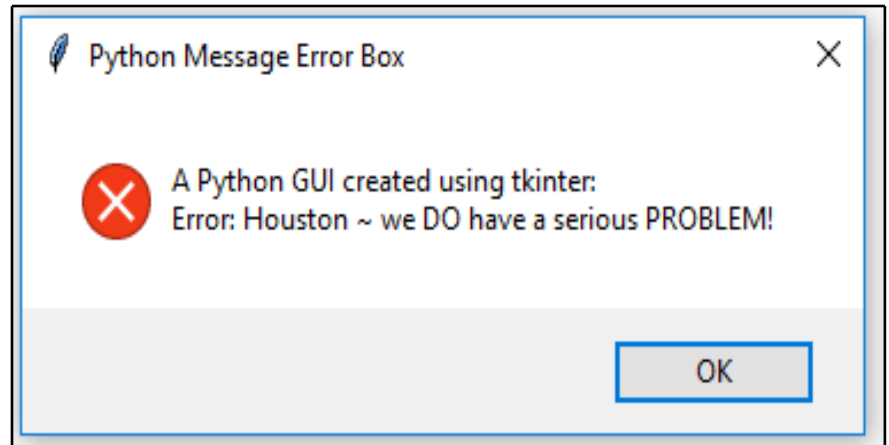
2. Replace the warning box code with error box code:

```
msg.showerror('Python Message  
Error Box', 'A Python GUI  
created using tkinter:'  
             '\nError: Houston ~ we DO have a  
             serious PROBLEM!')
```

The preceding instructions produce the following code:

```
# Display a Message Box
def _msgBox():
#     msg.showinfo('Python Message Info Box', 'A Python GUI created using tkinter:\nThe year is 2019.')
#     msg.showwarning('Python Message Warning Box', 'A Python GUI created using tkinter:'
#                     '\nWarning: There might be a bug in this code.')
    msg.showerror('Python Message Error Box', 'A Python GUI created using tkinter:'
                  '\nError: Houston ~ we DO have a serious PROBLEM!')
```

3. Run the `GUI_message_box_error.py` file.
The error message looks like this:



There are different message boxes that display more than one **OK** button, and we can program our responses according to the user's selection.

The following is a simple example that illustrates this technique:

1. Open `GUI_message_box_error.py` and save the module as `GUI_message_box[13]_yes_no_cancel.py`.

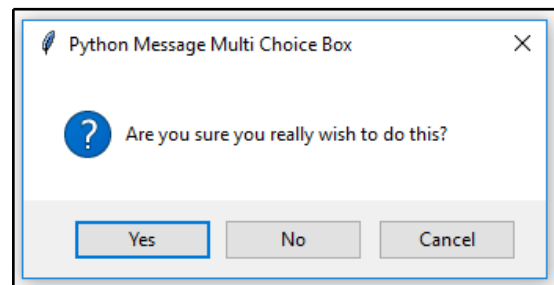
2. Replace the error box with a `yes_no_cancel` box:

```
answer =  
msg.askyesnocancel("Python  
Message Multi Choice Box",  
"Are you sure you really  
wish to do this?")
```

The preceding instructions produce the following code:

```
# Display a Message Box  
def _msgBox():  
#   msg.showinfo('Python Message Info Box', 'A Python GUI created using tkinter:\nThe year is 2019.')  
#   msg.showwarning('Python Message Warning Box', 'A Python GUI created using tkinter:\nWarning: There might be a bug in this code.')  
#   msg.showerror('Python Message Error Box', 'A Python GUI created using tkinter:\nError: Houston ~ we DO have a serious PROBLEM!')  
answer = msg.askyesnocancel("Python Message Multi Choice Box", "Are you sure you really wish to do this?")  
print(answer)
```

3. Run the `GUI_message_box_yes_no_cancel.py` file. Running this GUI code results in a popup whose user response can be used to branch on the answer of this event-driven GUI loop, by saving it in the `answer` variable:



The console output using Eclipse shows that clicking the **Yes** button results in the Boolean value of **True** being assigned to the `answer` variable:

```
Console X
<terminated> GUI_message_box_yes_no_cancel.py [C:\Python37\python.exe]
False
None
True
[ 15 ]
```

For example, we could use the following code:

```
If answer == True:  
    <do something>
```

Clicking No returns False and Cancel returns None

Now, let's go behind the scenes to understand the code better.

2. Create independent message boxes

In this recipe, we will create our tkinter message boxes as standalone top-level GUI windows.

You will first notice that, by doing so, we end up with an extra window, so we will explore ways to hide this window.

In the previous recipe, we invoked tkinter message boxes via our Help | About menu from our main GUI form.

Getting ready

We have already created the title of a message box in the previous recipe, Creating message boxes - information, warning, and error. We will not reuse the code from the previous recipe, but build a new GUI using very few lines of Python code.

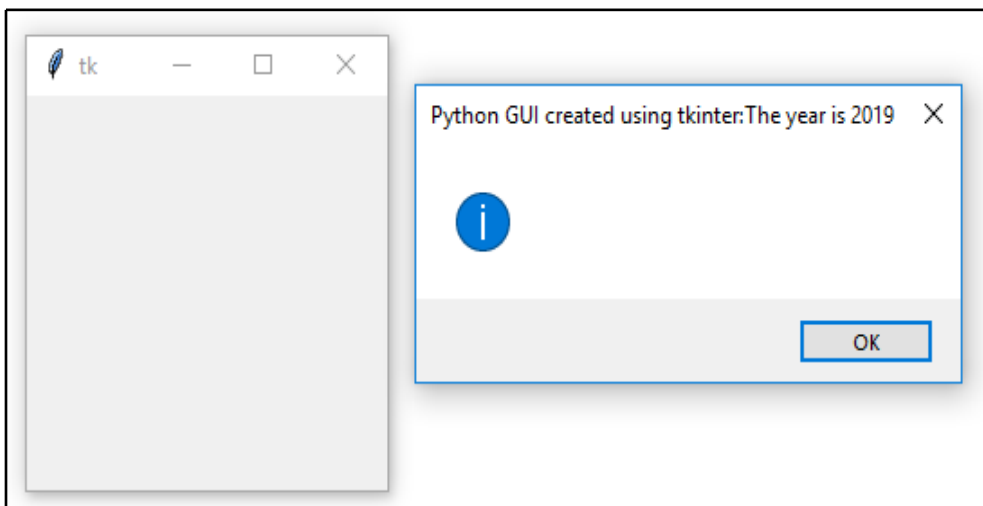
How to do it...

We can create a simple message box as follows:

1. Create a new module and save it as GUI_independent_msg.py.
2. Add the following two lines of code, which is all that is required:

```
from tkinter import messagebox as msg
msg.showinfo('Python GUI created using
tkinter:\nThis is 2019')
```

3.Run the `GUI_independent_msg.py` file. This will result in the following two windows:



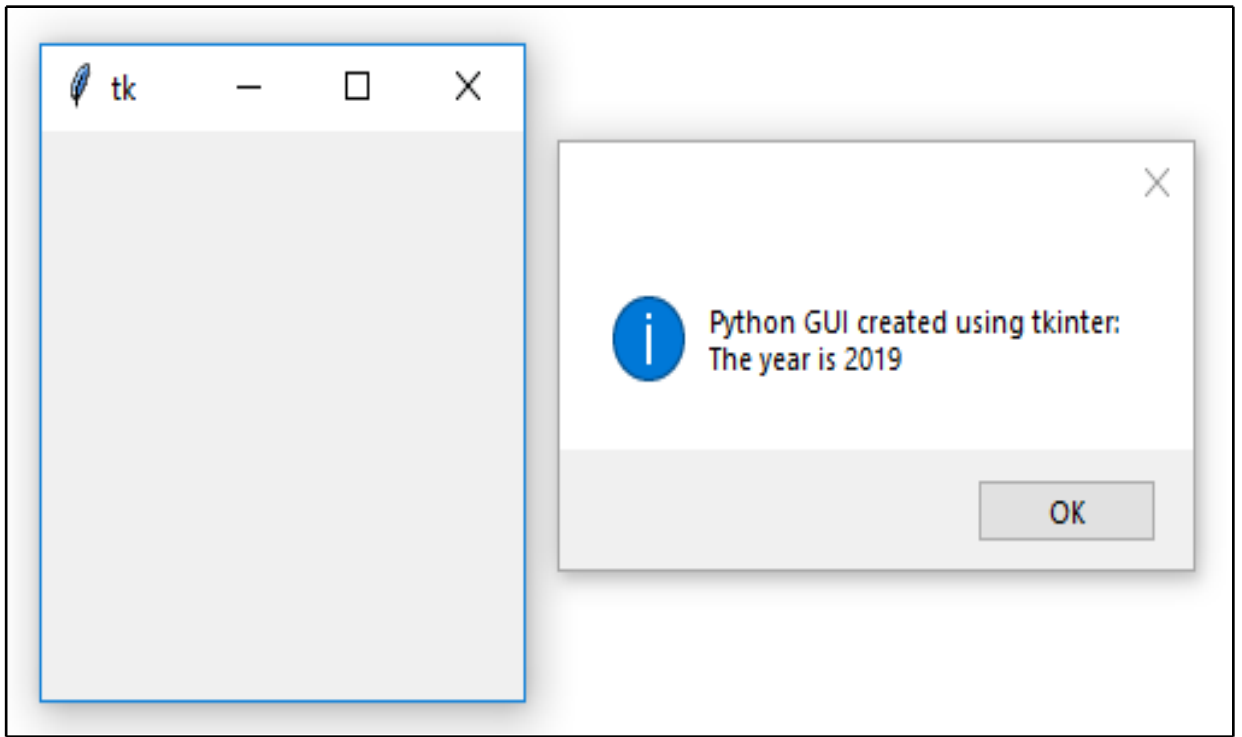
Let's solve this now. We can change the Python code by adding a single or double quote followed by a comma:

1. Open `GUI_independent_msg.py` and save the module as `GUI_independent_msg_info.py`.

2. Create an empty title:

```
from tkinter import messagebox as msg
msg.showinfo("", 'Python GUI created
using tkinter:\nThe year is 2019')
```

3. Run the `GUI_independent_msg_info.py` file. Now, we do not have a title but our text ended up inside the popup, as we had intended:



The first parameter is the title and the second is the text displayed in the pop-up message box. By adding an empty pair of single or double quotes followed by a comma, we can move our text from the title into the pop-up message box.

We still need a title, and we definitely want to get rid of this unnecessary second window. The second window is caused by a Windows event loop. We can get rid of it by suppressing it.

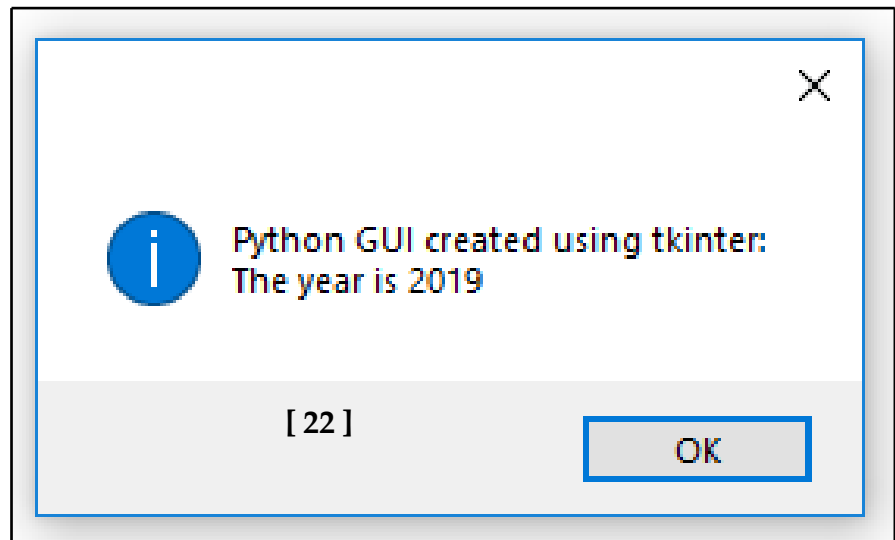
Add the following code:

1. Open `GUI_independent_msg_info.py` and save the module as `GUI_independent_msg_one_window.py`.
2. Import Tk create an instance of the Tk class, and call the `withdraw` method:

```
from tkinter import Tk  
root = Tk()  
root.withdraw()
```

Now, we have only one window. The `withdraw()` method removes the debug window that we are not interested in having floating around.

3. Run the code. This will result in the following window:



In order to add a title, all we have to do is place string into our empty first argument.

For example, consider the following code snippet:

1. Open

GUI_independent_msg_one_window.py and
save the module as
GUI_independent_msg_one_window_title.py.

2. Give it a title by adding some words into the first argument position:

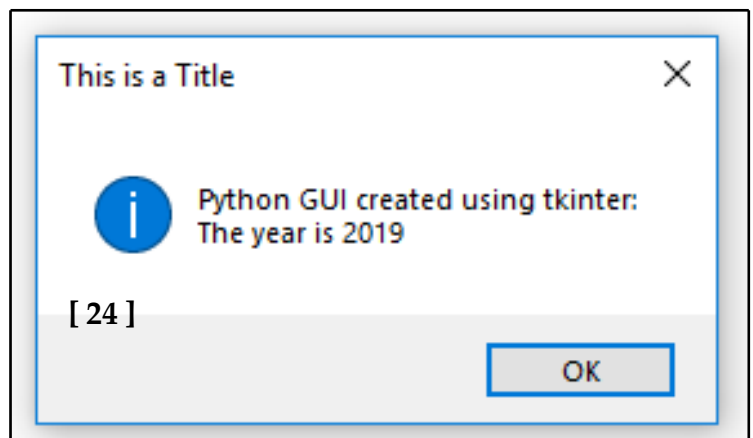
```
msg.showinfo('This is a Title', 'Python  
GUI created using tkinter:\n\nThe year is  
2019')
```

The preceding instructions produce the following code:

```
from tkinter import messagebox as msg
from tkinter import Tk
root = Tk()
root.withdraw()
msg.showinfo('This is a Title', 'Python GUI created using tkinter:\n\nThe year is 2019')
```

3. Run the

GUI_independent_msg_one_window_title.py file. Now, our dialog has a title, as shown in the following screenshot:



3. How to create the title of a tkinter window form

How to create independent message boxes. We just pass in a string as the first argument to the constructor of the widget.

Getting ready

Instead of a pop-up dialog window, we create the main root window and give it a title.

How to do it...

The following code creates the main window and adds a title to it. Here, we just focus on this aspect of our GUI:

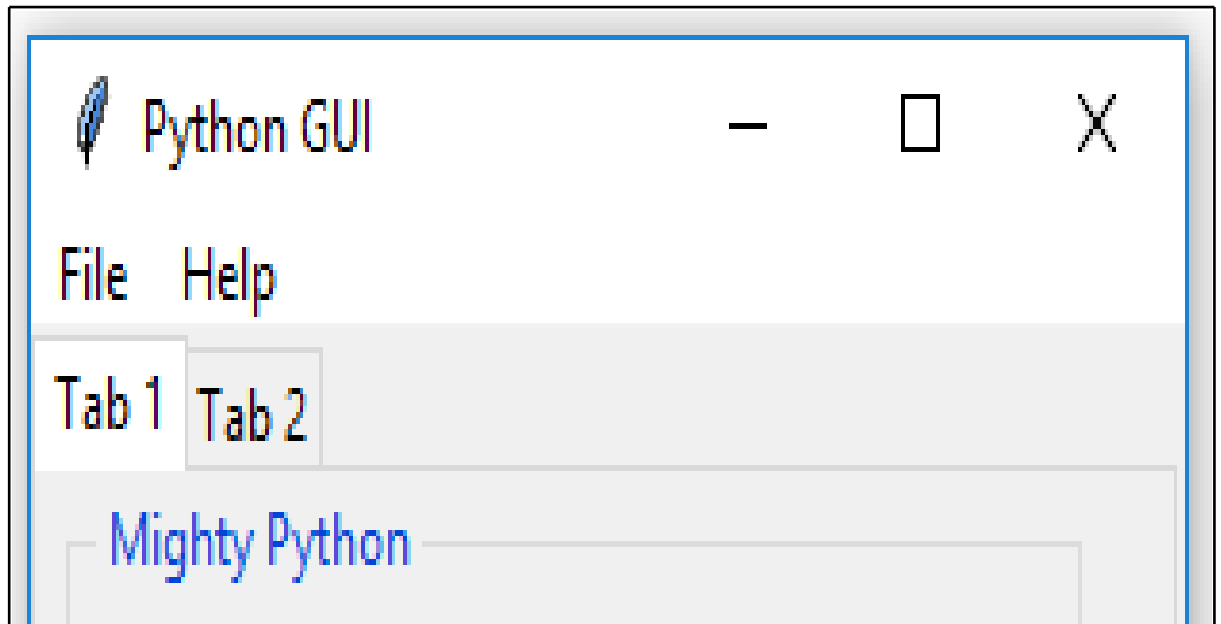
1. Open

GUI_tabbed_all_widgets_both_tabs.py
save the module as GUI_title.py.

2. Give the main window a title:

```
import tkinter as tk                                # Create instance
win = tk.Tk()                                       # Add a title
win.title("Python GUI")
```

3. Run the `GUI_title.py` file. This will result in the following two tabs:



Now, let's go behind the scenes to understand the code better.

4. Changing the icon of the main root window

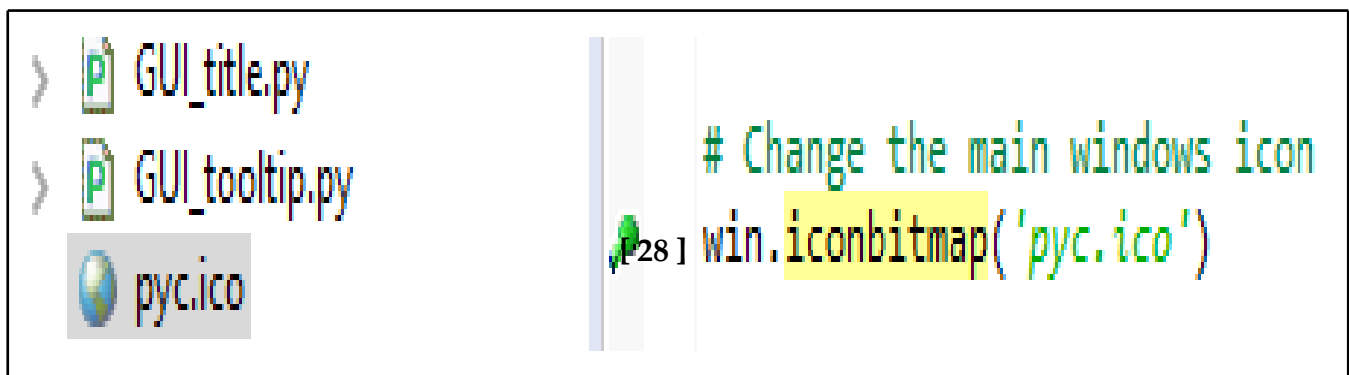
One way to customize our GUI is to give it an icon different from the default icon that ships out of the box with tkinter.

Getting ready

We are improving our GUI from the *Creating tabbed widgets* recipe in Chapter 2, *Layout Management*. We will use an icon that ships with Python, but you can use any icon you find useful. Make sure you have the full path to where the icon lives in your code, or you might get errors.

How to do it...

For this example, I have copied the icon from where I installed Python 3.7 to the same folder where the code lives. The following screenshot shows the icon that we will be using:

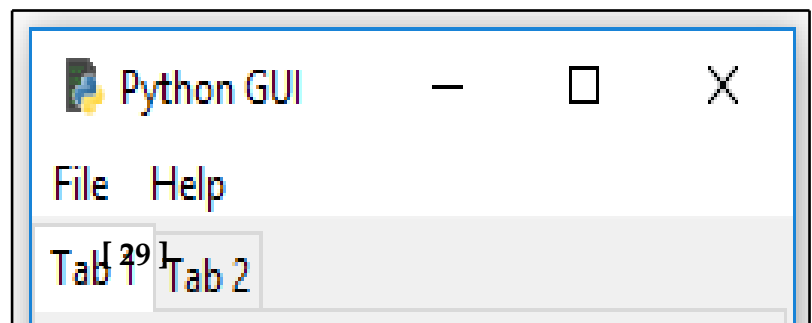


In order to use this or another icon file, perform the following steps:

1. Open GUI_title.py and save the module as GUI_icon.py.
2. Place the following code above the main event loop:

```
# Change the main  
windows icon  
win.iconbitmap('pyc.ico')
```

3. Run the GUI_icon.py file. Observe how the feather default icon in the top-left corner of the GUI changed:



5. Using a spin box control

In this recipe, we will use a Spinbox widget, and we will also bind the Enter key on the keyboard to one of our widgets. The Spinbox widget is a one-line widget, like the Entry widget, with the additional capability to restrict the values it will display. It also has some small up/down arrows to scroll up and down between the values.

Getting ready

We will use our tabbed GUI, from the How to create the title of a tkinter window form recipe, and add a Spinbox widget above the ScrolledText control. This simply requires us to increment the ScrolledText row value by one and insert our new Spinbox control in the row above the Entry widget.

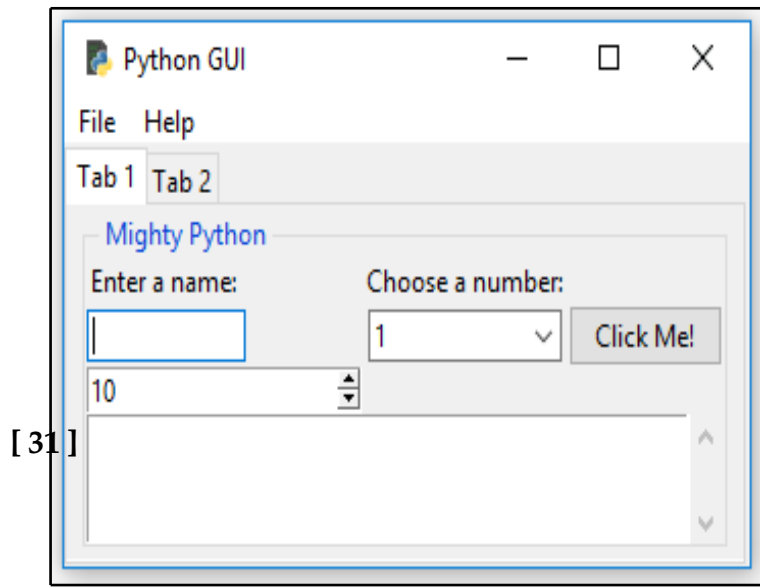
How to do it...

First, we add the Spinbox control by performing the following instructions:

1. Open GUI_title.py and save the module as GUI_spinbox.py.
2. Place the following code above the ScrolledText widget:

```
# Adding a Spinbox widget
spin = Spinbox(mighty,
from_=0, to=10)
spin.grid(column=0,
row=2)
```

3. Run the code. This will modify our GUI as follows:

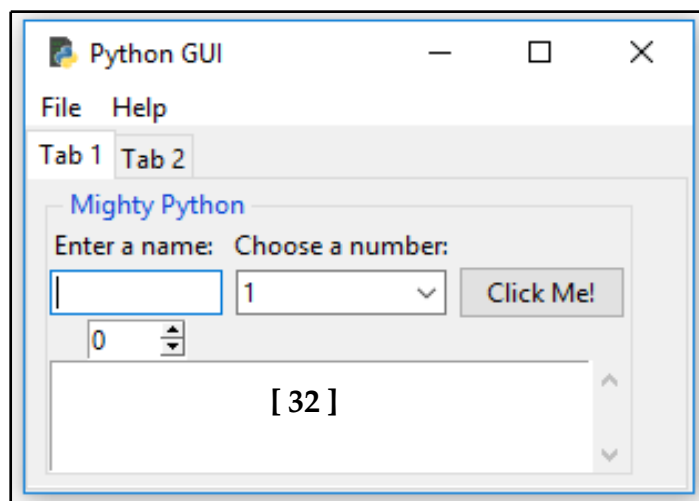


Next, we will reduce the size of the Spinbox widget:

1. Open `GUI_spinbox.py` and save the module as `GUI_spinbox_small.py`.
2. Add a width attribute when creating the Spinbox widget:

```
spin = Spinbox(mighty, from_=0, to=10,  
               width=5)
```

3. Running the preceding code results in the following GUI:

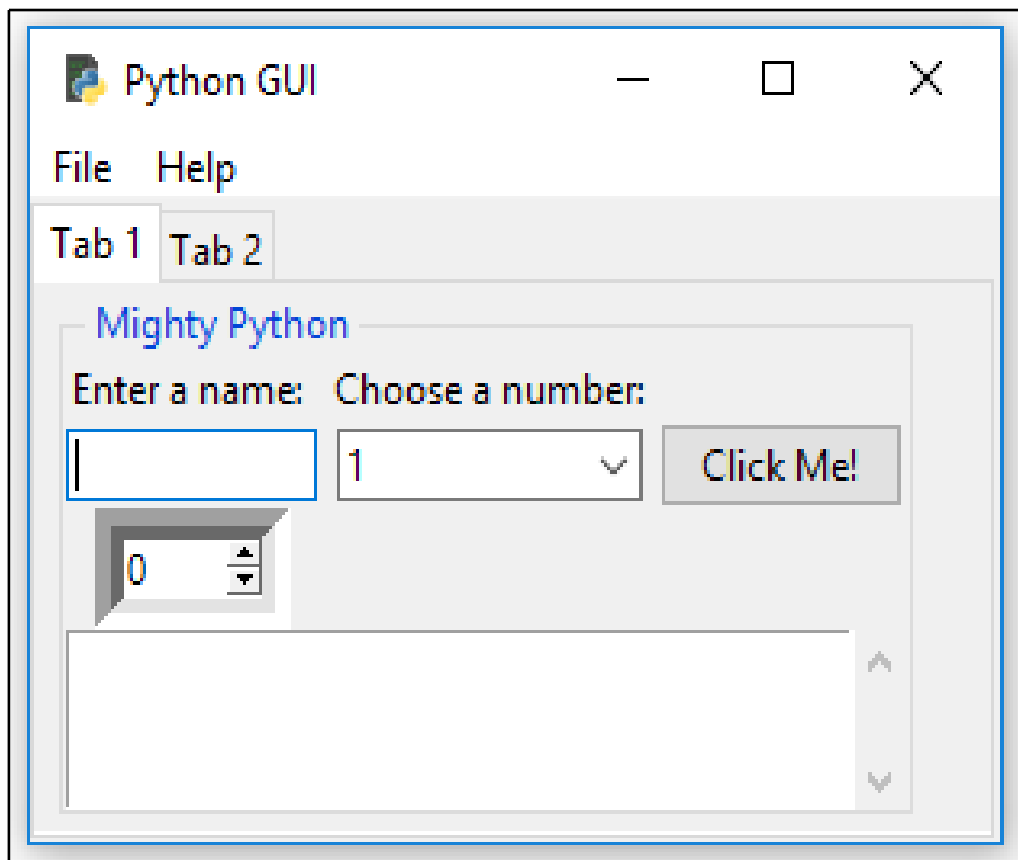


Next, we add another attribute to customize our widget further; `bd` is short-hand notation for the `borderwidth` attribute, and changes the width of the border surrounding the spin box:

1. Open `GUI_spinbox_small.py` and save the module as `GUI_spinbox_small_bd.py`.
2. Add a `bd` attribute, giving it a size of 8:

```
spin = Spinbox(mighty, from_=0, to=10,  
               width=5 , bd=8)
```

3. Running the preceding code results in the following GUI:



Next, we add functionality to the widget by creating a callback and linking it to the control.

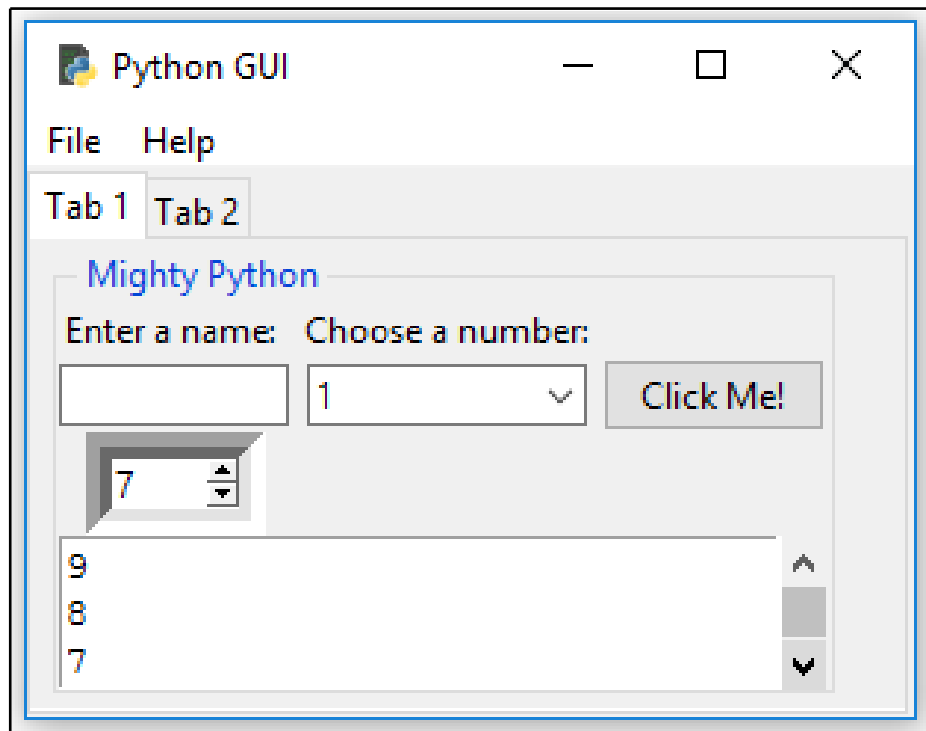
The following steps show how to print the selection of the Spinbox widget into ScrolledText as well as onto stdout. The variable named `scrol` is our reference to the ScrolledText widget:

1. Open `GUI_spinbox_small_bd.py` and save the module as `GUI_spinbox_small_bd_scrol.py`.
2. Write a callback function right above the creation of the Spinbox widget and assign it to the `command` attribute of the Spinbox: widget:

```
# spinbox callback
def _spin():
    value = spin.get()
    print(value)
    scrol.insert(tk.INSERT, value + '\n')
    # <-- add a newline

spin = Spinbox(mighty, from_=0, to=10,
width=5, bd=8, command=_spin)
# <-- command=_spin
```

3. Running the
GUI_spinbox_small_bd_scrol.py
file results in the following GUI when
clicking the Spinbox arrows:



Instead of using a range, we can also specify a set of values by performing the following instructions:

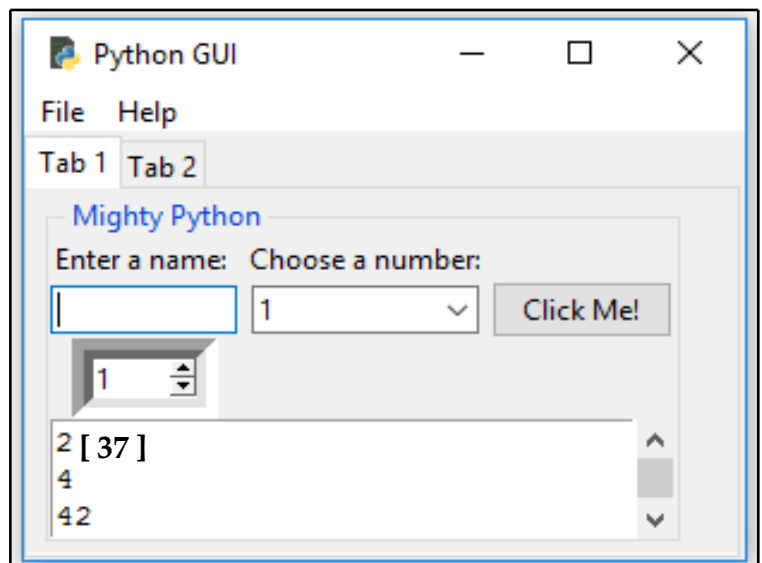
1. Open GUI_spinbox_small_bd_scrol.py and save the module as

GUI_spinbox_small_bd_scrol_values.py.

2. Add the values attribute, replacing from_=0, to=10, and assign it a tuple of numbers during the creation of the Spinbox widget:

```
# Adding a Spinbox widget using a set of values
spin = Spinbox(mighty, values=(1, 2, 4, 42,
100), width=5, bd=8, command=_spin)
spin.grid(column=0, row=2)
```

3. Run the code. This will create the following GUI output:



How it works...

Note how, in the first Python module, `GUI_spinbox.py`, our new Spinbox control defaulted to a width of 20, pushing out the column width of all controls in this column. This is not what we want. We gave the widget a range from 0 to 10.

- `GUI_spinbox_small.py`, we reduced the width of the Spinbox control, which aligned it in the center of the column.
- `GUI_spinbox_small_bd.py`, we added the `borderwidth` attribute of the Spinbox, which automatically made the entire Spinbox appear no longer flat, but three-dimensional.
- `GUI_spinbox_small_bd_scrol.py`, we added a callback function to display the number chosen in the `ScrolledText` widget and also print it to the standard out stream. We added `\n` to insert the values on new lines within the callback function, `def _spin()`.

Notice, when we click the control that the callback function gets called. By clicking the down arrow with a default of 0, we can print the 0 value.

Lastly, in `GUI_spinbox_small_bd_scrol_values.py`, we restricted the values available to a hardcoded set. This could also be read in the form of a data source (for example, a text or XML file).

6. Applying relief – the sunken and raised appearance of widgets

We can control the appearance of our Spinbox widgets by using an attribute that makes them appear in different formats, such as sunken or raised. This attribute is the relief attribute.

Getting ready

We will add one more Spinbox control to demonstrate the available appearances of widgets, using the relief attribute of the Spinbox control.

How to do it...

While we are creating the second Spinbox, let's also increase borderwidth to distinguish our second Spinbox from the first Spinbox:

1. Open `GUI_spinbox_small_bd_scroll_values.py` and save the module as `GUI_spinbox_two_sunken.py`.
2. Add a second Spinbox just below the first Spinbox and set `bd=20`:

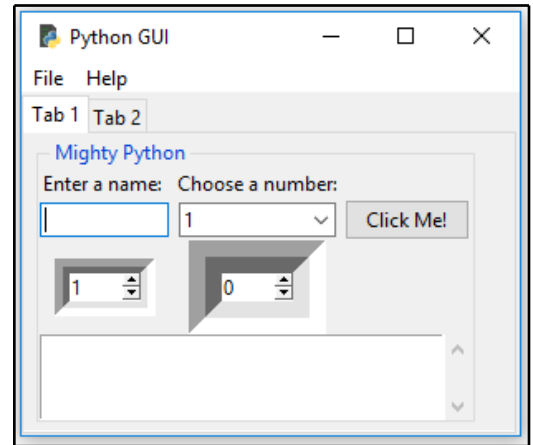
```
# Adding a second Spinbox widget
spin2 = Spinbox(mighty, values=(0, 50,
100), width=5, bd=20, command=_spin2)
# <-- new function spin2.grid(column=1, row=2)
```

3. We will also create a new callback function for the command attribute, `_spin2`. Place this function above the code just shown, where we create the second Spinbox:

```
# Spinbox2 callback function
```

```
def _spin2():
    value = spin2.get()
    print(value)
    scrol.insert(tk.INSERT, value + '\n')
    # <-- write to same ScrolledText
```


4. Run the code. This will create the following GUI output:



Our two spin boxes look different but this is only because of the difference in the `borderwidth` (`bd`) we specified. Both widgets look three-dimensional, and this is much more visible in the second `Spinbox` that we have added.

They actually both have a `relief` style even though we did not specify the `relief` attribute when we created the spin boxes.

+ When not specified,
the relief style defaults to `SUNKEN`.

+ Here are the available relief attribute
options that can be set:

- `tk.SUNKEN`
- `tk.RAISED`
- `tk.FLAT`
- `tk.GROOVE`
- `tk.RIDGE`

By assigning the different available options to the relief attribute , we can create different appearances for this widget.

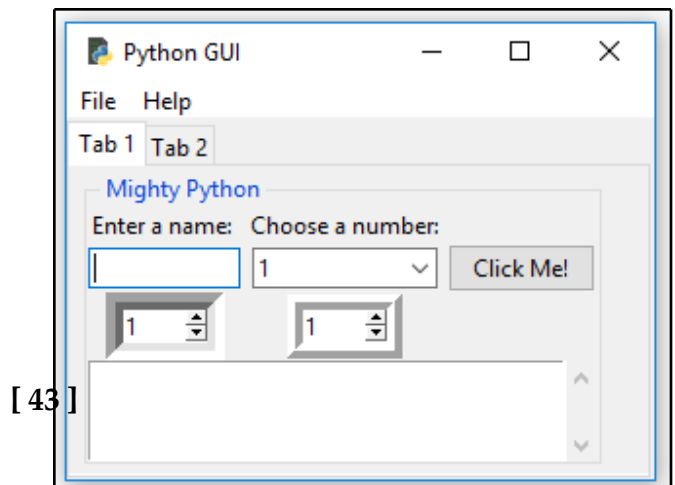
Assigning the `tk.RIDGE` relief and reducing the border width to the same value as our first Spinbox widget results in the following GUI:

1. Open `GUI_spinbox_two_sunken.py` and save the module as `GUI_spinbox_two_ridge.py`.

2. Set relief to `tk.RIDGE`:

```
spin2 = Spinbox(mighty, values=(0, 50,  
100), width=5, bd=9,  
command=_spin2, relief=tk.RIDGE)
```

3. Run the code. The following GUI is obtained after running the code:



Notice the difference in appearance of our second Spinbox widget, on the right.

Now, let's go behind the scenes to understand the code better.

How it works...

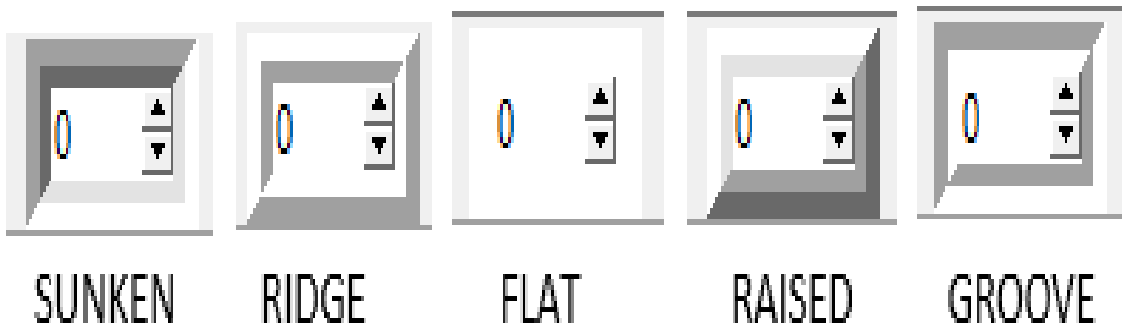
First, we created a second Spinbox aligned in the second column (`index == 1`). It defaults to `SUNKEN`, so it looks similar to our first Spinbox. We distinguished the two widgets by increasing the border width of the second control (the one on the right).

Next, we explicitly set the relief attribute of the Spinbox widget. We made borderwidth the same as our first Spinbox because, by giving it a different relief, the differences became visible without having to change any other attributes.

Here is an example of the different relief options, *GUI_spinbox_two_ridge.py*:

```
# Adding a second Spinbox widget displaying its relief options
# uncomment each next code line to see the different effects
spin2 = Spinbox(mighty, values=(0, 50, 100), width=5, bd=9, command=_spin2, relief=tk.RIDGE)
# spin2 = Spinbox(mighty, values=(0, 50, 100), width=5, bd=9, command=_spin2) # default value is: tk.SUNKEN
# spin2 = Spinbox(mighty, values=(0, 50, 100), width=5, bd=9, command=_spin2, relief=tk.FLAT)
# spin2 = Spinbox(mighty, values=(0, 50, 100), width=5, bd=9, command=_spin2, relief=tk.RAISED)
# spin2 = Spinbox(mighty, values=(0, 50, 100), width=5, bd=9, command=_spin2, relief=tk.SUNKEN) # default
# spin2 = Spinbox(mighty, values=(0, 50, 100), width=5, bd=9, command=_spin2, relief=tk.GROOVE)
```

And here is a screenshot of what those relief attributes create:



We've successfully learned how to use and apply relief, sunken, and raised appearances to widgets. Now, let's move on to the next recipe.

7. Creating tooltips using Python

This recipe will show you how to create tooltips. When the user hovers the mouse over a widget, additional information will be available in the form of a tooltip.

We will code this additional information into our GUI.

Getting ready

We will be adding more useful functionality to our GUI. Surprisingly, adding a tooltip to our controls should be simple, but it is not as simple as we'd want it to be.

In order to achieve this desired functionality, we will place our tooltip code in its own OOP class.

How to do it...

These are the steps to create a tooltip:

1. Open `GUI_spinbox_small_bd_scrol_values.py` and save the module as `GUI_tooltip.py`.

2. Add the following class just below the import statements:

```
class ToolTip(object):
    def init (self, widget, tip_text=None):
        self.widget = widget
        self.tip_text = tip_text
        widget.bind('<Enter>', self.mouse_enter)
        widget.bind('<Leave>', self.mouse_leave)
```

3. Add two new methods to the class below `init`:

```
def mouse_enter(self, _event):
    self.show_tooltip()

def mouse_leave(self, _event):
    self.hide_tooltip()
```

4. Add another method below these two,
and name the method `show_tooltip`:

```
def show_tooltip(self):  
    if self.tip_window:  
        x_left = self.widget.winfo_rootx()  
        y_top = self.widget.winfo_rooty() - 18  
        self.tip_window = tk.Toplevel(self.widget)  
        self.tip_window.overridereDIRECT(True)  
        self.tip_window.geometry("+%d+%d" % (x_left,  
        y_top))  
        label = tk.Label(self.tip_window, text=self.tip_text,  
        justify=tk.LEFT, background="#ffffe0",  
        relief=tk.SOLID, borderwidth=1, font=("tahoma",  
        "8", "normal"))  
        label.pack(ipadx=1)
```

5. Add another method below `show_tooltip`,
and name it `hide_tooltip`:

```
def hide_tooltip(self):  
    if self.tip_window: [48]  
        self.tip_window.destroy()
```


6. Below the class and below the code where we create the Spinbox widget, create an instance of the ToolTip class, passing in the Spinbox variable, spin:

```
# Adding a Spinbox widget
```

```
spin = Spinbox(mighty, values=(1, 2, 4, 42, 100),  
width=5, bd=9, command=_spin)  
spin.grid(column=0, row=2)
```

```
# Add a Tooltip to the Spinbox
```

```
ToolTip(spin, 'This is a Spin control')# <-- add this code
```

7. Perform the same step for the ScrolledText widget just below the Spinbox widget:

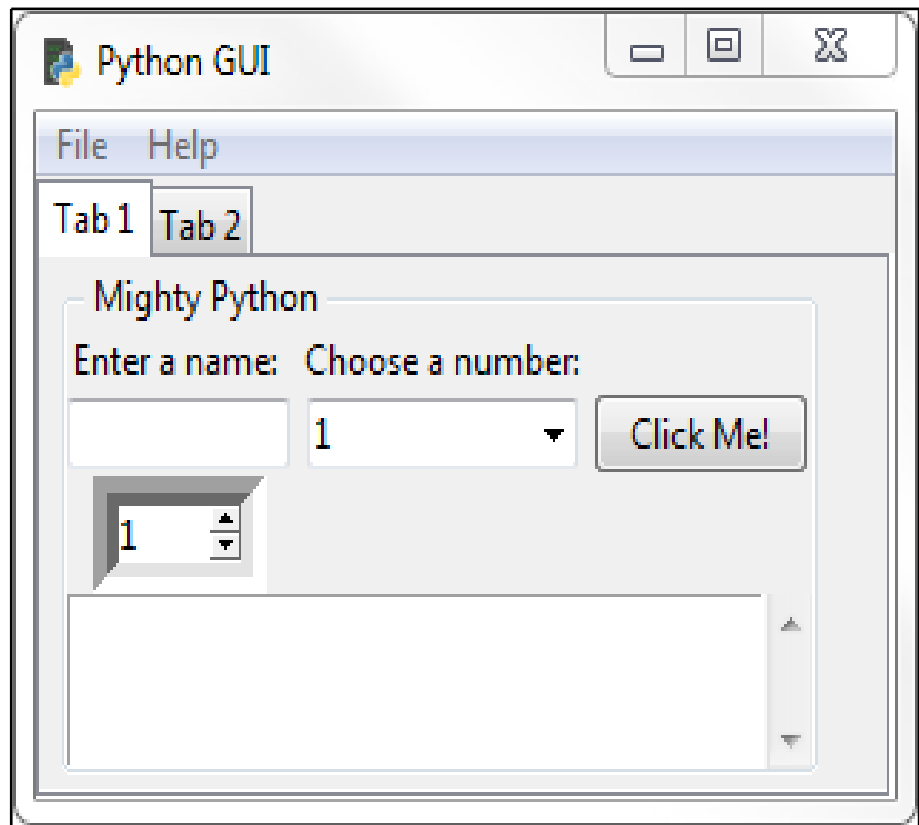
```
scrol = scrolledtext.ScrolledText(mighty,  
width=scrol_w, height=scrol_h, wrap=tk.WORD)  
scrol.grid(column=0, row=3, sticky='WE',  
columnspan=3)
```

```
# Add a Tooltip to the ScrolledText widget
```

```
ToolTip(scrol, 'This is a ScrolledText widget')
```

```
# <-- add this code
```

8. Run the code and hover the mouse over the `ScrolledText` widget:



Now, let's go behind the scenes to understand the code better.

8. Adding Progressbar to the GUI

In this recipe, we will add a Progressbar to our GUI. It is very easy to add a `ttk.Progressbar`, and we will demonstrate how to start and stop a Progressbar. This recipe will also show you how to delay the stopping of a Progressbar, and how to run it in a loop.

A Progressbar is typically used to show the current status of a long-running process.

Getting ready

We will add Progressbar to Tab 2 of the GUI that we developed in a previous recipe:

Using a spin box control.

How to do it...

Here are the steps to create a Progressbar and some new Buttons that start and stop the Progressbar:

1. Open `GUI_spinbox_small_bd_scrol_values.py` and save the module as `GUI_progressbar.py`.

2. At the top of the module, add sleep to the imports:

```
from time import sleep
```

```
# careful - this can freeze the GU
```

3. Add Progressbar below the code where we create the three Radiobutton widgets:

```
# Now we are creating all three Radiobutton widgets  
within one loop
```

```
for col in range(3):  
    curRad = tk.Radiobutton(mighty2,  
        text=colors[col], variable=radVar,  
        value=col, command=radCall)  
    curRad.grid(column=col, row=1,  
        sticky=tk.W) # row=6
```

```
# Add a Progressbar to Tab 2 # <--- add this code here
```

```
progress_bar = ttk.Progressbar(tab2,  
    orient='horizontal', length=286, mode='determinate')  
progress_bar.grid(column=0, row=3, pady=2)
```

4. Next, we write a callback function to update Progressbar:

```
# update progressbar in callback loop
def run_progressbar():
    progress_bar["maximum"] = 100
    for i in range(101):
        sleep(0.05)
        progress_bar["value"] = i # increment progressbar
        progress_bar.update()     # have to call update() in loop
    progress_bar["value"] = 0     # reset/clear progressbar
```

5. We then write the following three functions below the preceding code:

```
def start_progressbar():  
    progress_bar.start()
```

```
def stop_progressbar():  
    progress_bar.stop()
```

```
def progressbar_stop_after(wait_ms=1000):  
    win.after(wait_ms, progress_bar.stop)
```

6. We will reuse `buttons_frame` and `LabelFrame`, but replace the labels with new code. Change the following code:

```
# PREVIOUS CODE -- REPLACE WITH BELOW CODE
```

```
# Create a container to hold labels
```

```
buttons_frame = ttk.LabelFrame(mighty2, text='  
Labels in a Frame ')
```

```
buttons_frame.grid(column=0, row=7)
```

```
# NEW CODE
```

```
# Create a container to hold buttons
```

```
buttons_frame = ttk.LabelFrame(mighty2, text='  
ProgressBar ')
```

```
buttons_frame.grid(column=0, row=2, sticky='W',  
columnspan=2)
```


7.Delete the previous labels that resided in buttons_frame:

```
# DELETE THE LABELS BELOW
# Place labels into the container element
ttk.Label(buttons_frame,
text="Label1").grid(column=0, row=0, sticky=tk.W)

ttk.Label(buttons_frame,
text="Label2").grid(column=1, row=0, sticky=tk.W)

ttk.Label(buttons_frame,
text="Label3").grid(column=2, row=0, sticky=tk.W)
```

8. Create four new buttons. `buttons_frame` is their parent:

Add Buttons for Progressbar commands

```
ttk.Button(buttons_frame, text=" Run Progressbar ",  
command=run_progressbar).grid(column=0, row=0,  
sticky='W')
```

```
ttk.Button(buttons_frame, text=" Start Progressbar ",  
command=start_progressbar).grid(column=0, row=1,  
sticky='W')
```

```
ttk.Button(buttons_frame, text=" Stop immediately ",  
command=stop_progressbar).grid(column=0, row=2,  
sticky='W')
```

```
ttk.Button(buttons_frame, text=" Stop after second ",  
command=progressbar_stop_after).grid(column=0,  
row=3, sticky='W')
```

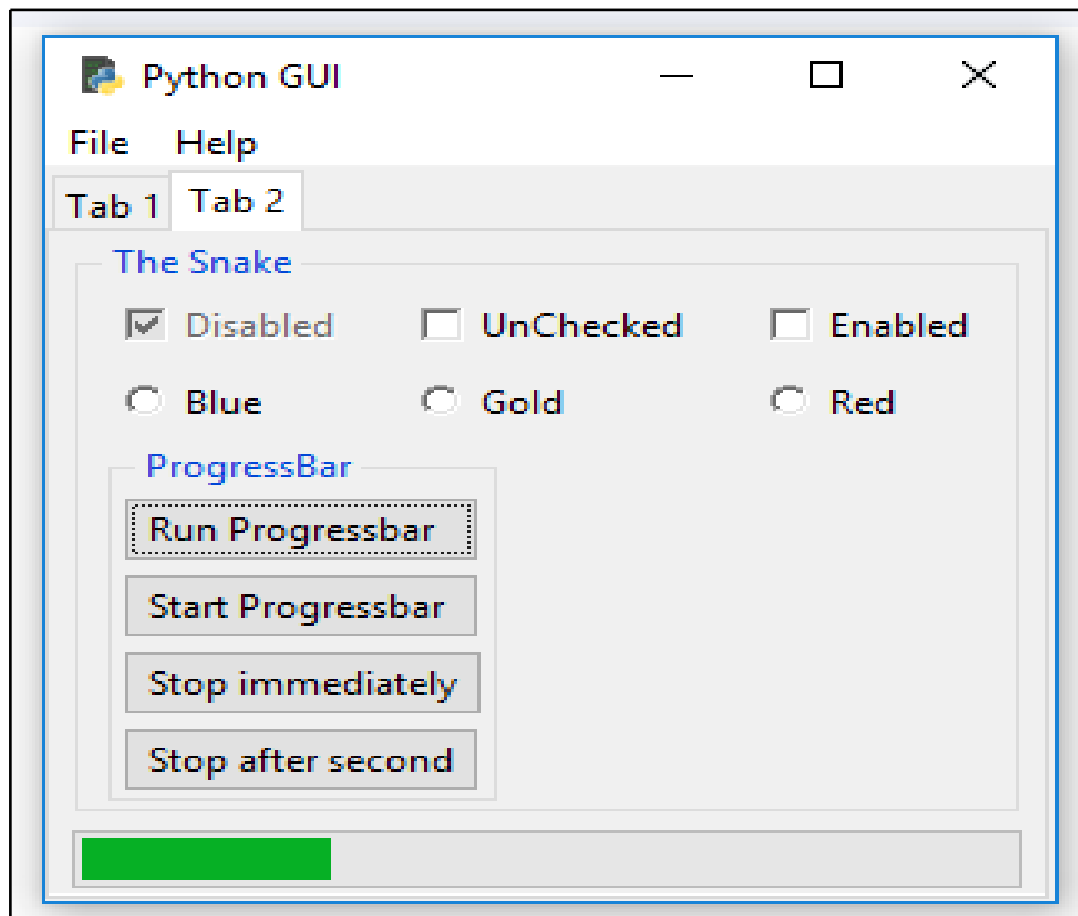
9. Add additional padding for the children of `buttons_frame` in a loop:

```
for child in buttons_frame.winfo_children():  
    child.grid_configure(padx=2, pady=2)
```

10. Add additional padding for all children of `Tab2`:

```
for child in mighty2.winfo_children():  
    child.grid_configure(padx=8, pady=2)
```

11. Run the code. The following GUI is obtained after clicking the Run Progressbar button:



9. Use the canvas widget

This recipe shows how to add dramatic color effects to our GUI by using the tkinter canvas widget.

Getting ready

We will improve our previous code from `GUI_tooltip.py`, and we'll improve the look of our GUI by adding some more colors to it.

How to do it...

First, we will create a third tab in our GUI in order to isolate our new code.

Here is the code to create the new third tab:

1. Open GUI_tooltip.py

and save the module as GUI_canvas.py.

2. Create a third tab control:

```
tabControl = ttk.Notebook(win) # Create Tab Control

tab1 = ttk.Frame(tabControl)      # Create a tab
tabControl.add(tab1, text='Tab 1') # Add the tab

tab2 = ttk.Frame(tabControl)
tabControl.add(tab2, text='Tab 2') # Add a second tab

tab3 = ttk.Frame(tabControl)
tabControl.add(tab3, text='Tab 3') # Add a third tab

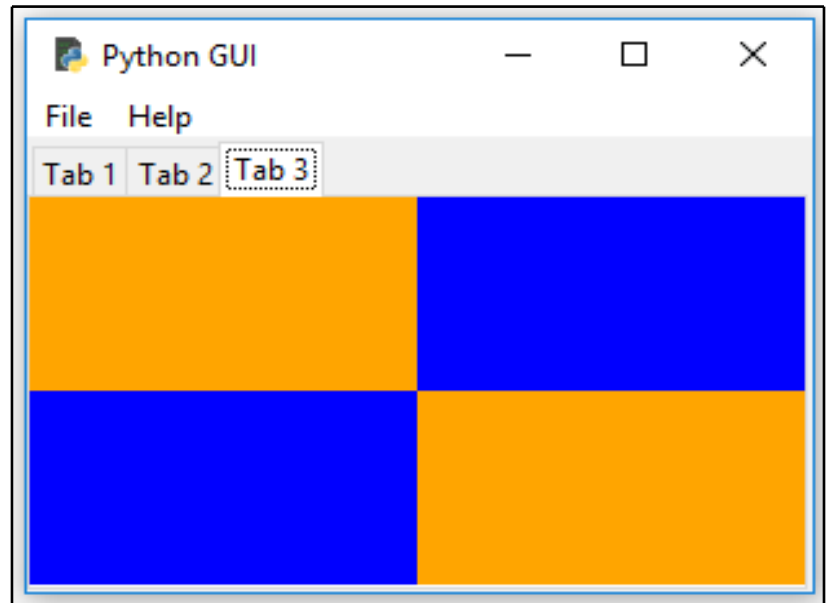
tabControl.pack(expand=1, fill="both")
# Pack to make tabs visible
```

3. Next, we use another built-in widget of tkinter, called Canvas. A lot of people like this widget because it has powerful capabilities:

Tab Control 3

```
tab3_frame = tk.Frame(tab3, bg='blue')
tab3_frame.pack()
for orange_color in range(2):
    canvas = tk.Canvas(tab3_frame, width=150,
                        height=80, highlightthickness=0, bg='orange')
    canvas.grid(row=orange_color,
                column=orange_color)
```

4. Run the `GUI_canvas.py` file. The following GUI is obtained after running the code:



How it works...

After we have created the new tab, we place a regular `tk.Frame` into it and assign it a background color of blue. In the loop, we create two `tk.Canvas` widgets, making their color orange and assigning them to the grid coordinates 0,0 and 1,1. This also makes the blue background color of the `tk.Frame` visible in the two other grid locations.

BÀI TẬP THỰC HÀNH LAB 3

Trên trang elearning:

<https://elearning.vanlanguni.edu.vn/course/view.php?id=13473>

Lập trình Python nâng cao - 213_DIT0540_01, 02, 03

[Home](#) / [My courses](#) / [Lập trình Python nâng cao - 213_DIT0540](#) (1/10/02)


Administrator

Trang tin về khóa học LẬP TRÌNH PYTHON NÂNG CAO



I. MỤC TIÊU HỌC HUẤN

Kiến thức: Môn học Lập trình Python Nâng cao sẽ trang bị và củng cố kiến thức cho sinh viên đã học xong môn học Cơ sở Lập trình Kỹ thuật Lập trình, bổ sung thêm kỹ năng lập trình khi học các môn học về mạng máy tính và hệ thống, cũng như vận dụng vào thực tiễn công việc của sinh viên lập trình ứng dụng các bài toán thực tế, đạt trình độ cao hơn và kiến thức kỹ năng lập trình Python.

Kỹ năng: Sinh viên có thể phân tích, phát triển và cải tiến một số chương trình máy tính với bằng ngôn ngữ lập trình Python sử dụng thư viện cơ bản của bộ thư viện lập trình Python để giải quyết vấn đề trong thực tiễn.

Thái độ: Có học tập nghiêm túc, tự giác, không ngại khó khăn, vượt qua những trở ngại để hoàn thành tốt nhiệm vụ được giao.

2. THÔNG TIN GIẢNG VIÊN	
Họ và Tên	Email
Tạ Xuân Thu Mỹ Linh	linh.amt@vnu.edu.vn
Trương Khắc Tùng	tung.tr@vnu.edu.vn
Vũ Ngọc Văn	viet.ng@vnu.edu.vn
Tổng Hưng Anh	anh.th@vnu.edu.vn
Vũ Kim Mỹ Vân	myvan.km@vnu.edu.vn
Đinh Minh Tuấn	minhtuan.dinh@vnu.edu.vn

NOTE:

+ Sinh viên điểm danh trên online Lab

<https://fit-lab.vlu.edu.vn>

+ SV nộp bài tập trên lớp vào online lab:

Blended Classes/ Lập trình Python NC/

Kho Học liệu **Lập trình Python nâng cao. Thư mục DIT0540 (FITLAB-02)**

Điểm danh sinh viên

Tạo Thư mục mới

Cung cấp Học liệu

QL Học liệu

Back

Home

213_DIT0540_01

Tuần 1 HK213

Choose Files

No file chosen

Kho Học liệu. Thư mục DIT0540

Bài giảng

Sinh viên

Thực hành

Tài liệu

Videos