

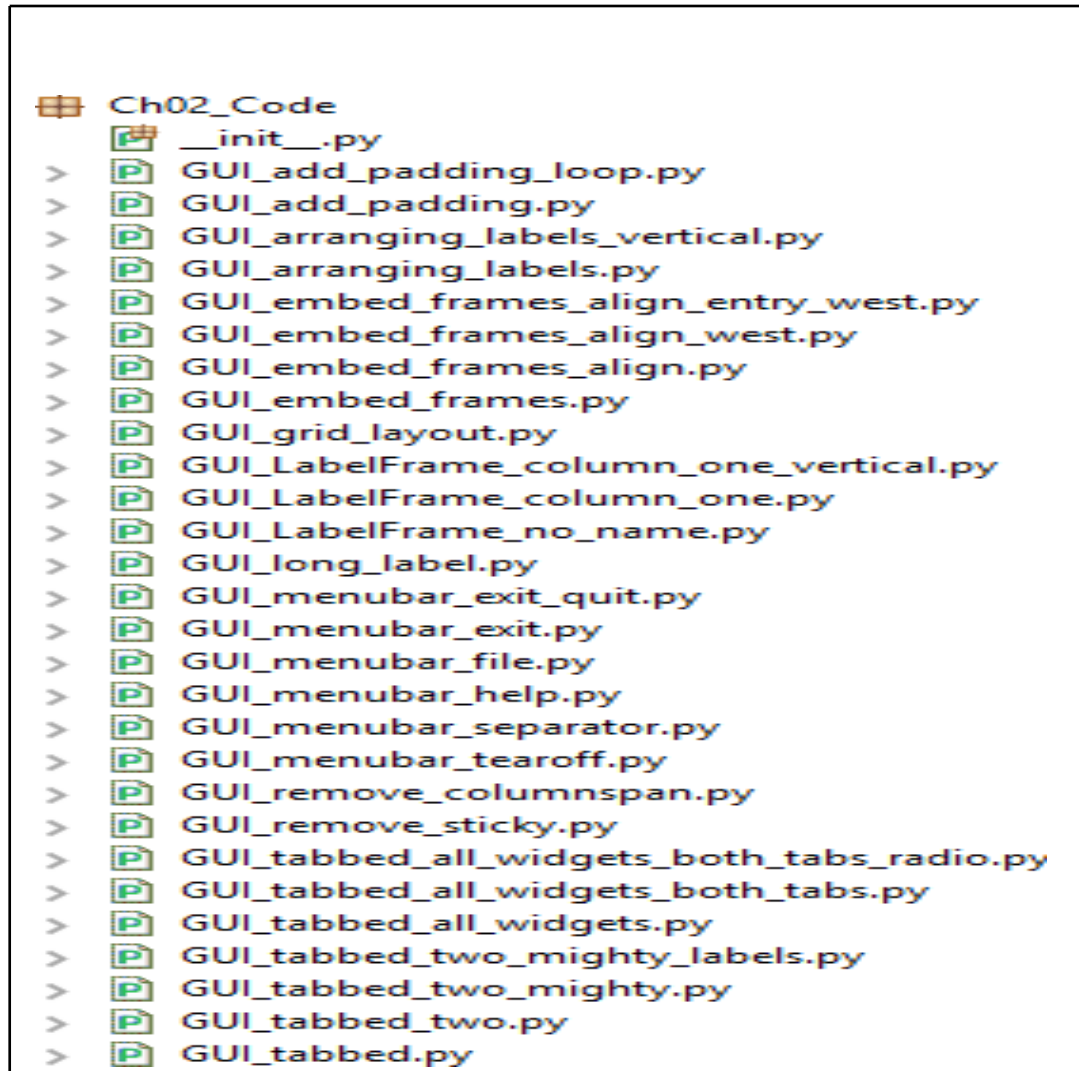
Chapter 2

Layout Management

Content

- Arranging several labels within a label frame Widget
- Using padding to add space around widgets
- Dynamically expanding the GUI using widgets
- Aligning GUI widgets by embedding frames within frames
- Creating menu bars
- Creating tabbed widgets
- Using the grid layout manager

The following screenshot provides an overview of the Python modules that will be used in this chapter:



Arranging several labels within a label frame widget

The `LabelFrame` widget allows us to design our GUI in an organized fashion. We are still using the grid layout manager as our main layout design tool, but by using `LabelFrame` widgets, we get much more control over our GUI's design.

Getting ready

We will start by adding more widgets to our GUI. We will make the GUI fully functional in upcoming recipes.

Here, we will start to use the `LabelFrame` widget. We will reuse the GUI from the *Adding several widgets to a loop* recipe in **Chapter 1, Creating the GUI Form and Adding Widgets**.

How to do it...

1. Open `GUI_adding_widgets_in_loop.py` from Chapter 1, *Creating the GUI Form and Adding Widgets*, and save the module as `GUI_LabelFrame_column_one.py`.
2. Create a `ttk.LabelFrame` and position it in the grid:

```
buttons_frame = ttk.LabelFrame(win, text=
'Labels in a Frame')
buttons_frame.grid(column=0, row=7)
# button_frame.grid(column=1, row=7)
```

How to do it...

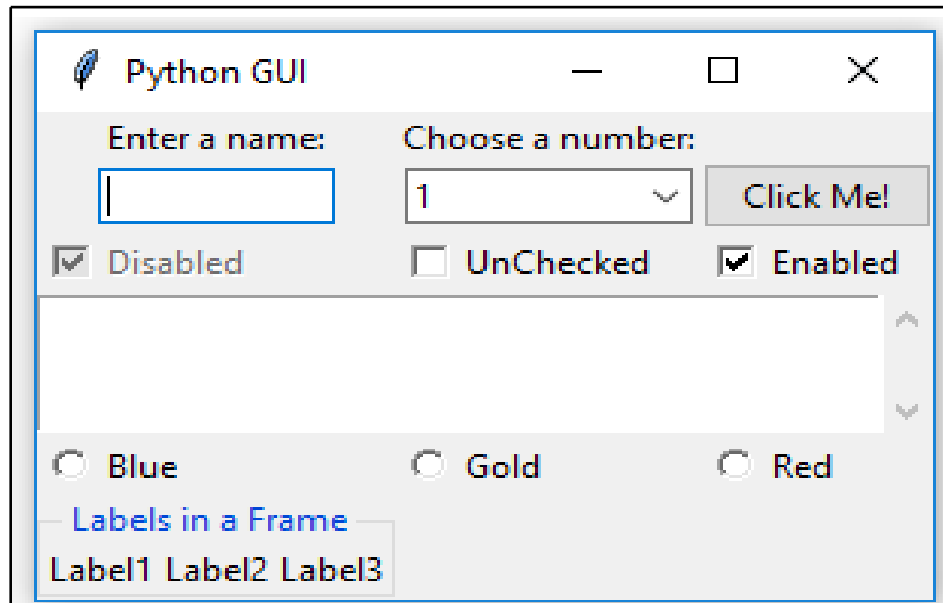
3. Create three `ttk` Labels, set their text attributes, and position them in the grid:

```
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)
```

The preceding instructions produce the following code from the `GUI_LabelFrame_column_one.py` file:

```
108 # Create a container to hold labels
109 buttons_frame = ttk.LabelFrame(win, text=' Labels in a Frame ')
110 buttons_frame.grid(column=0, row=7)
111 # buttons_frame.grid(column=1, row=7)           # now in col 1
112
113 # Place labels into the container element
114 ttk.Label(buttons_frame, text="Label1").grid(column=0, row=0, sticky=tk.W)
115 ttk.Label(buttons_frame, text="Label2").grid(column=1, row=0, sticky=tk.W)
116 ttk.Label(buttons_frame, text="Label3").grid(column=2, row=0, sticky=tk.W)
117
118 name_entered.focus()      # Place cursor into name Entry
119 #=====
120 # Start GUI
121 #=====
122 win.mainloop()
```


4. Run the code. It will result in the following GUI:



In addition, we can easily align the labels vertically by changing our code. To do this perform the following steps:

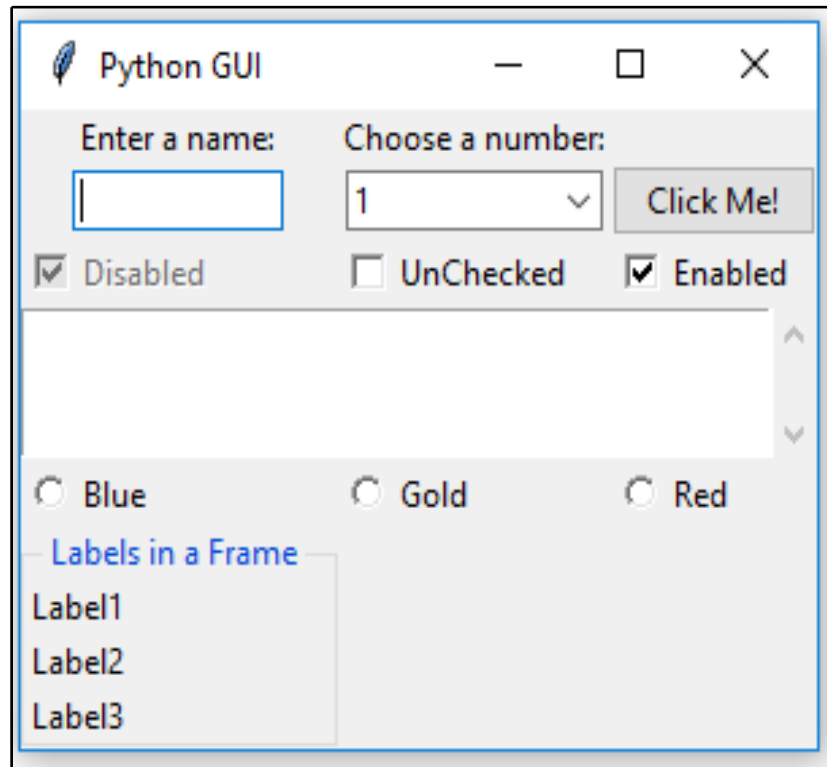
1. Open `GUI_LabelFrame_column_one.py` and save the module as

`GUI_LabelFrame_column_one_vertical.py`.

2. Change the column and row values, as follows:

```
ttk.Label(button_frame,  
text="Label1").grid(column=0,row=0)  
ttk.Label(button_frame,  
text="Label2").grid(column=0,row=1)  
ttk.Label(button_frame,  
text="Label3").grid(column=0,row=2)
```

3. Run the `GUI_LabelFrame_column_one_vertical.py` file. Now the GUI label frame will look as follows:



Using padding to add space around widgets

We will improve the visual aspects of our widgets by adding a little space around them so that they can breathe.

Getting ready

While tkinter might have had a reputation for creating not-so-pretty GUIs, this has dramatically changed since version 8.5.

tkinter version 8.6 ships with Python 3.7. There's no need to install anything other than Python in order to use it.

A simple way of adding spacing around widgets will be shown first, and then we will use a loop to achieve the same thing in a much better way.

Our LabelFrame looks a bit tight as it blends into the main window toward the bottom.

How to do it...

Follow these steps to add padding around the widgets:

1. Open `GUI_LabelFrame_column_one.py` and save it as `GUI_add_padding.py`.
2. Add `padx` and `pady` to the `grid` method:

```
buttons_frame.grid(column=0, row=7,  
padx=20, pady=40) # padx, pady
```

3. Run the code. Now our `LabelFrame` has some breathing space. We can see this in the following screenshot:



We can use a loop to add space around the labels contained within `LabelFrame`. Follow these steps to do so:

1. Open `GUI_add_padding.py` and save it as `GUI_add_padding_loop.py`.
2. Add the following loop below the creation of the three Labels:

```
for child in
    buttons_frame.winfo_children():
    child.grid_configure(padx=8,
        pady=4)
```


The preceding instructions produce the following code:

```
113 # Place labels into the container element
114 ttk.Label(buttons_frame, text="Label1").grid(column=0, row=0)
115 ttk.Label(buttons_frame, text="Label2").grid(column=0, row=1)
116 ttk.Label(buttons_frame, text="Label3").grid(column=0, row=2)
117
118 for child in buttons_frame.winfo_children():
119     child.grid_configure(padx=8, pady=4)
120
121 name_entered.focus()      # Place cursor into name Entry
122 #=====
123 # Start GUI
124 #=====
125 win.mainloop()
```

3. Run the `GUI_add_padding_loop.py` file code. Now the labels within the `LabelFrame` widget have some space around them too:



To see this effect better, let's do the following:

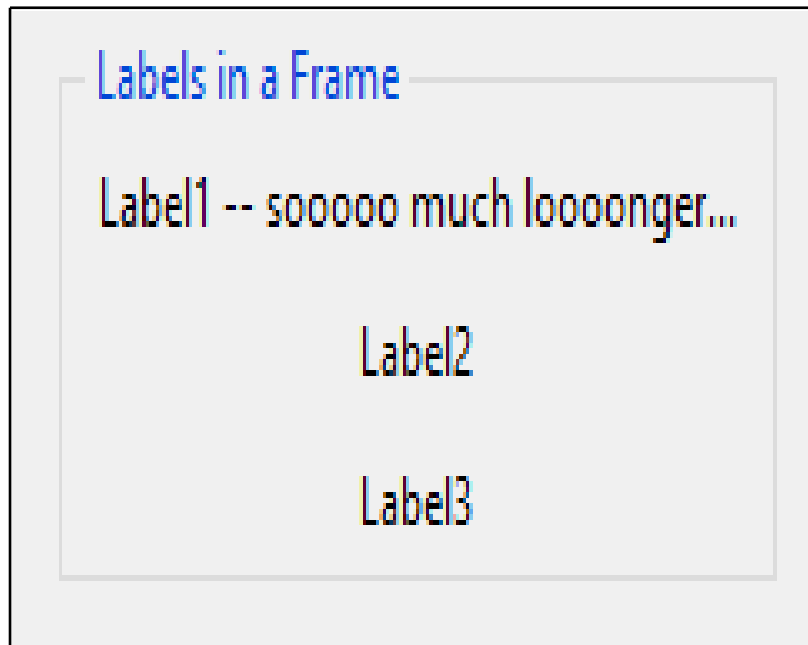
1. Open `GUI_add_padding_loop.py` and save it as `GUI_long_label.py`.

2. Change the text of `Label1`, like so:

```
ttk.Label(buttons_frame, text="Label1 – so much  
longer...").grid(column=0, row=0)
```

To see this effect better, let's do the following:

3. Run the code. This will generate what's shown in the following screenshot, which shows our GUI. Note how there is now space to the right of the long label, next to the dots. The last dot doesn't touch `LabelFrame`, which it otherwise would have without the added space:

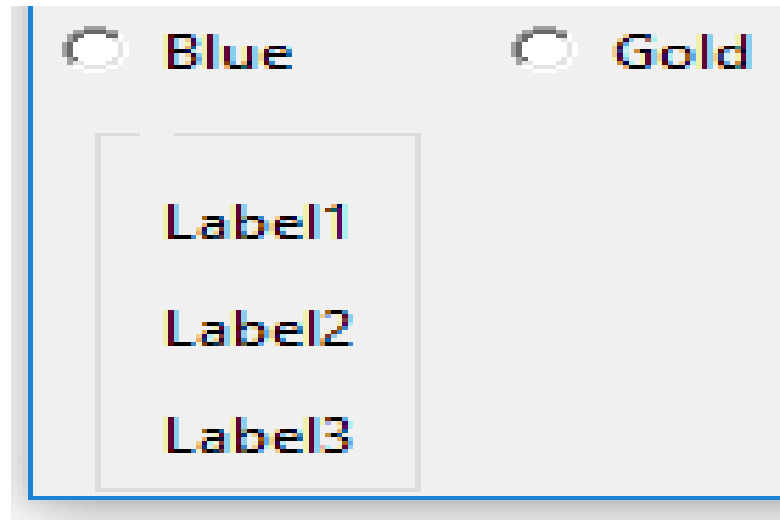


We can also remove the `LabelFrame` name to see the effect `padx` has on the position of our labels. Let's get started:

1. Open `GUI_add_padding_loop.py` and save it as `GUI_LabelFrame_no_name.py`.
2. When creating the button, set the text attribute to an empty string:

```
buttons_frame = ttk.LabelFrame(win, text="")
```

3. Run the code. By setting the `text` attribute to an empty string, we remove the name that was previously displayed for `Label1Frame`. This can be seen in the following screenshot:



Dynamically expanding the GUI using widgets

You may have noticed from the previous screenshots and by running the preceding code that the widgets can extend themselves to take up as much space as they need in order to visually display their text.

Using tkinter, this dynamic capability creates both an advantage and a little bit of a challenge because, sometimes, our GUI dynamically expands when we would like it not to be so dynamic! Well, we are dynamic Python programmers, so we can figure out how to make the best use of this fantastic behavior!

Getting ready

At the beginning of the previous recipe, *Using padding to add space around widgets*, we added a `LabelFrame` widget. This moved some of our controls to the center of column 0. We might not want this modification in our GUI layout. We will explore some ways to solve this in this recipe.

The following is an example of a grid layout manager with two rows and three columns:

Row 0; Col 0	Row 0; Col 1	Row 0; Col 2
Row 1; Col 0	Row 1; Col 1	Row 1; Col 2



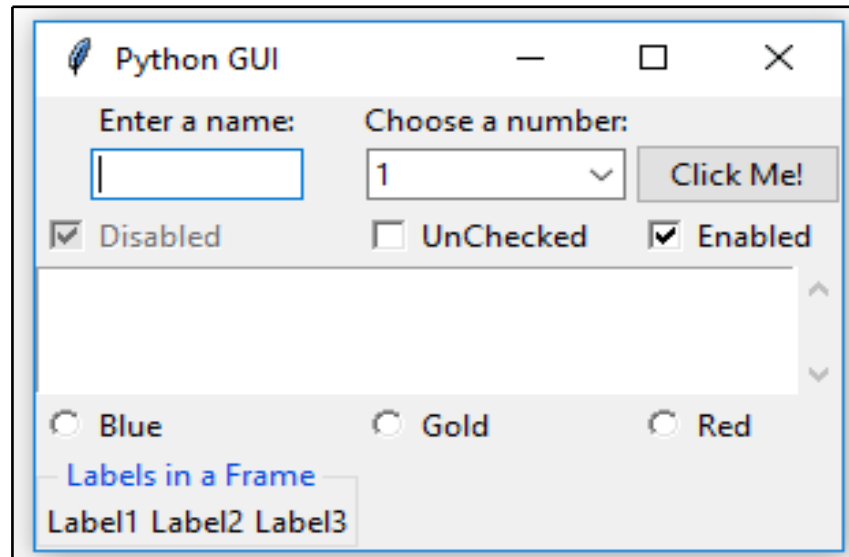
Using the grid layout manager, the width of any given column is determined by the longest name or widget in that column. This, in turn, affects all of the rows.

By adding our `LabelFrame` widget and giving it a title that is longer than a hard-coded size widget, we dynamically move those widgets to the center of column 0. By doing so, we are adding space on the left- and right-hand side of those widgets.

Incidentally, because we used the sticky property for the `Checkbutton` and `ScrolledText` widgets, those remain attached to the left-hand side of the frame.

Let's take a look at the screenshot from the first recipe in this chapter, *Arranging several labels within a label frame widget*, in more detail.

Since the text property of `LabelFrame`, which is displayed as the title of `LabelFrame`, is longer than both our `Enter a name:` label and the text box entry below it, those two widgets are dynamically centered within the new width of column 0, as shown in the following screenshot:



Notice how both the label and the entry below it are no longer positioned on the left but have been moved to the center within the grid column.

We added the following code to `GUI_LabelFrame_no_name.py` to create a `LabelFrame` and then placed labels in this frame to stretch both the `Label` frame and the widgets contained therein:

```
buttons_frame = ttk.LabelFrame(win,  
text='Labels in a Frame')  
buttons_frame.grid(column=0, row=7)
```

The `Checkbutton` and `Radiobutton` widgets in column 0 did not get centered because we used the `sticky=tk.W` attribute when we created those widgets.

For the `ScrolledText` widget, we also used `sticky=tk.WE`, which binds the widget to both the west (the left) and east (the right) side of the frame.

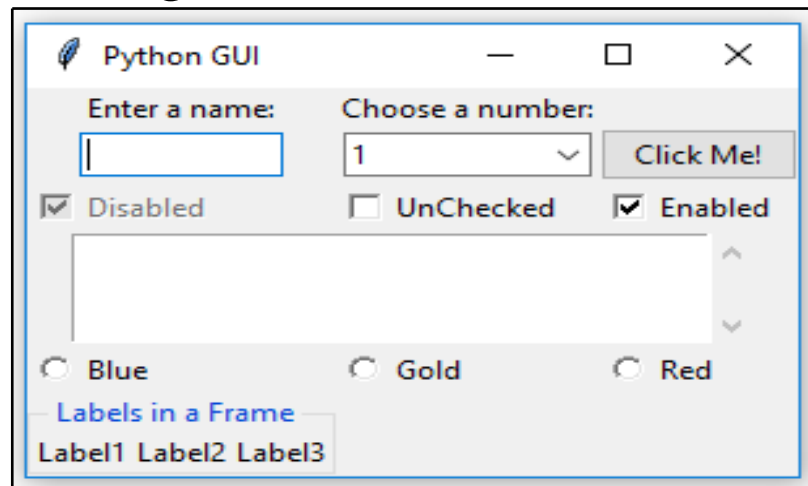
How to do it...

Perform the following steps to complete this recipe:

1. Open `GUI_arranging_labels.py` and save it as `GUI_remove_sticky.py`.
2. Remove the `sticky` attribute from the `ScrolledText` widget and observe the effect this change has.

```
# Using a scrolled Text control
scrol_w = 30
scrol_h = 3
scr = scrolledtext.ScrolledText(win, width=scrol_w, height=scrol_h, wrap=tk.WORD)
#### scr.grid(column=0, row=5, sticky='WE', columnspan=3)
scr.grid(column=0, row=5, columnspan=3)           # sticky property removed
```

3. Run the code. Now our GUI has a new space around the `ScrolledText` widget, both on the left- and right-hand sides. Because we used the `columnspan=3` property, our `ScrolledText` widget still spans all three columns. This is shown in the following screenshot:

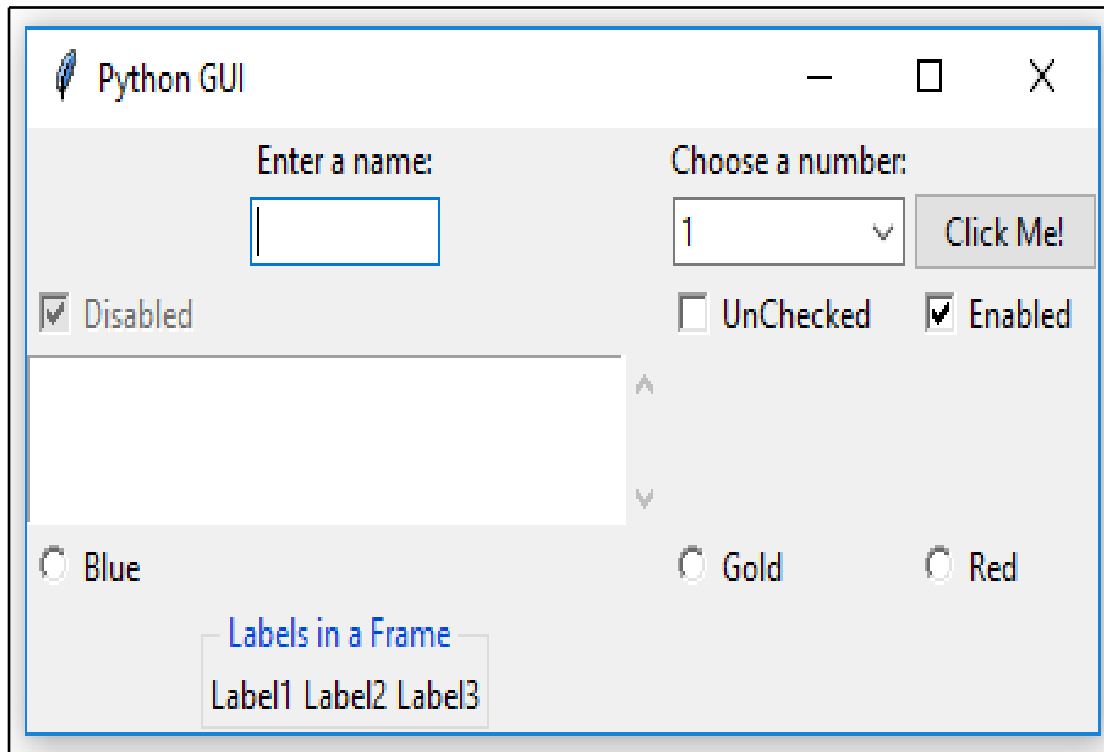


Using `columnspan` is necessary to arrange our GUI in the way we desire it to look.

Let's take a look at how *not* using the `columnspan` attribute could screw up our nice GUI design by doing the following modifications:

1. Open `GUI_remove_sticky.py` and save it as `GUI_remove_columnspan.py`.
2. If we remove `columnspan=3`, we'll get the GUI that's shown in the following screenshot, which is not what we want. Now `ScrolledText` only occupies column 0 and, because of its size, stretches the layout.

3. Run the `GUI_remove_columnspan.py` file and observe the output:



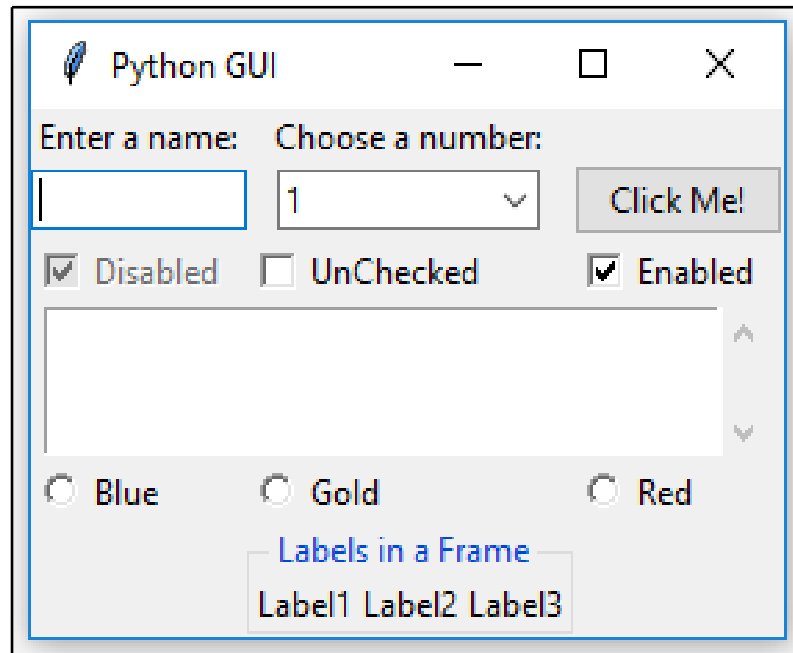
One way to get our layout back to where it was before adding `LabelFrame` is to adjust the `grid` column position. Let's get started:

1. Open `GUI_remove_columnspan.py` and save it as `GUI_LabelFrame_column_one.py`.

2. Change the column value from 0 to 1:

```
buttons_frame.grid(column=1, row=7)
```

3. Run the code. Now our GUI will look as follows:



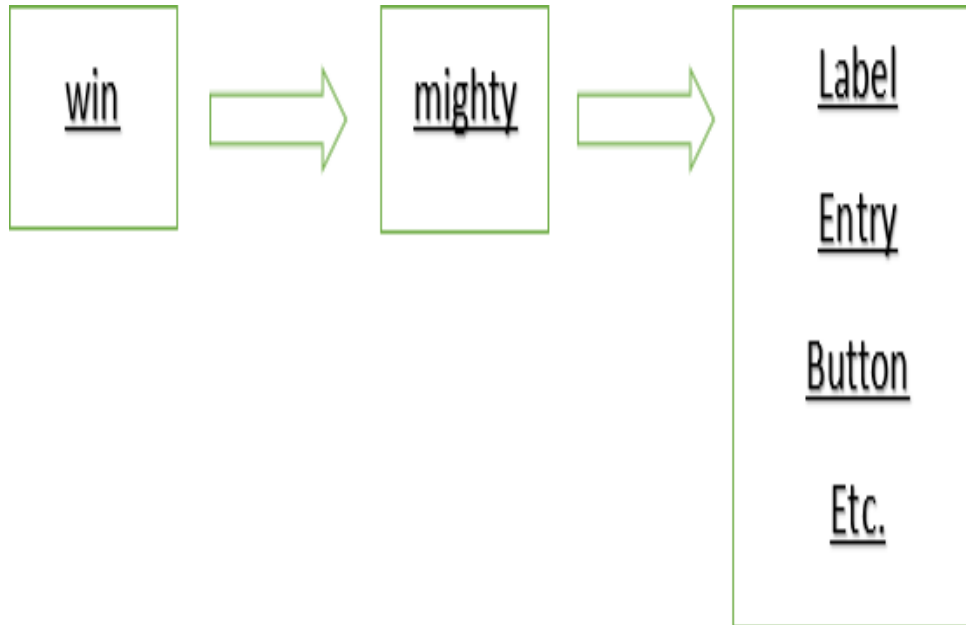
Aligning GUI widgets by embedding frames within frames

We'll have better control of our GUI layout if we embed frames within frames. This is what we will do in this recipe.

Getting ready

We will continue using the GUI we created in the previous recipe, *Dynamically expanding the GUI using widgets*.

Creates the following hierarchy in our GUI layout:



How to do it...

Perform the following steps to complete this recipe:

1. Open `GUI_LabelFrame_column_one.py` and save it as `GUI_embed_frames.py`.
2. Add the following code toward the top of our Python module:

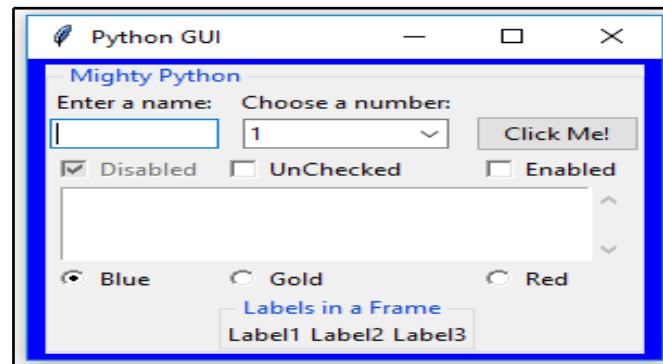
```
mighty = ttk.LabelFrame(win, text='  
Mighty Python ') mighty.grid(column=0,  
row=0, padx=8, pady=4)
```

Next, we will modify the following controls to use `mighty` as the parent, replacing `win`.

3. Change the Label parent from win to mighty:

```
a_label = ttk.Label(mighty,  
text="Enter a name:")  
a_label.grid(column=0, row=0)
```

4. Run the `GUI_embed_frames.py` file. This results in the GUI shown in the following screenshot:



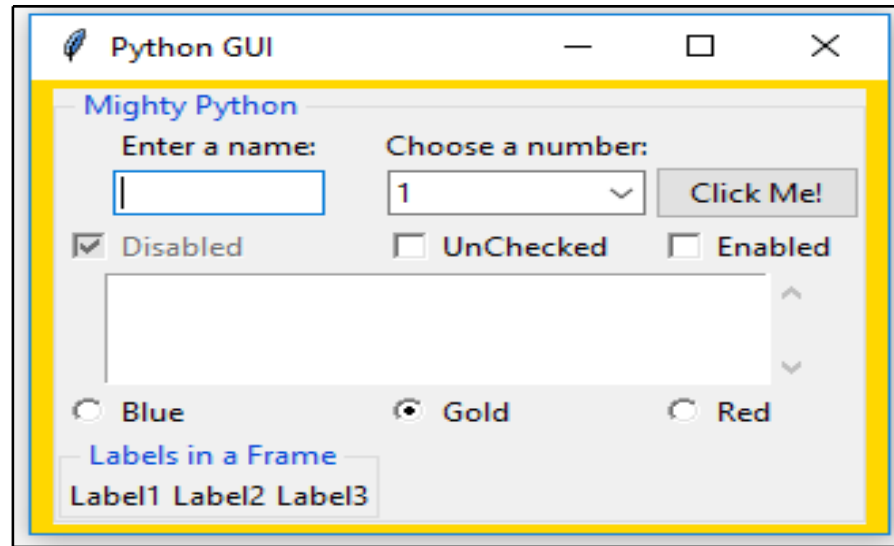
Note how all the widgets are now contained in the **Mighty Python LabelFrame**, which surrounds all of them with a barely visible thin line. Next, we can reset the **Labels in a Frame** widget to the left without messing up our GUI layout:

1. Open `GUI_embed_frames.py` and save it as `GUI_embed_frames_align.py`.

2. Change `column` to 0:

```
buttons_frame = ttk.LabelFrame(mighty, text='
Labels in a Frame ')
buttons_frame.grid(column=0, row=7)
```

3. Run the `GUI_embed_frames_align.py` file. This results in the GUI shown in the following screenshot:



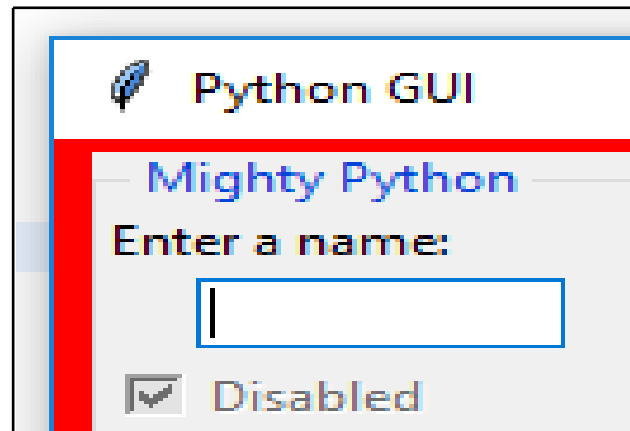
To align them to the left, we have to force our GUI layout by using the `sticky` property. By assigning it `'W'` (west), we can force the widget to be left-aligned. Perform the following steps:

1. Open `GUI_embed_frames_align.py` and save it as `GUI_embed_frames_align_west.py`.
2. Add the `sticky` attribute to the label:

```
a_label = ttk.Label(mighty,  
text="Enter a name:")  
a_label.grid(column=0,  
row=0, sticky='w')
```

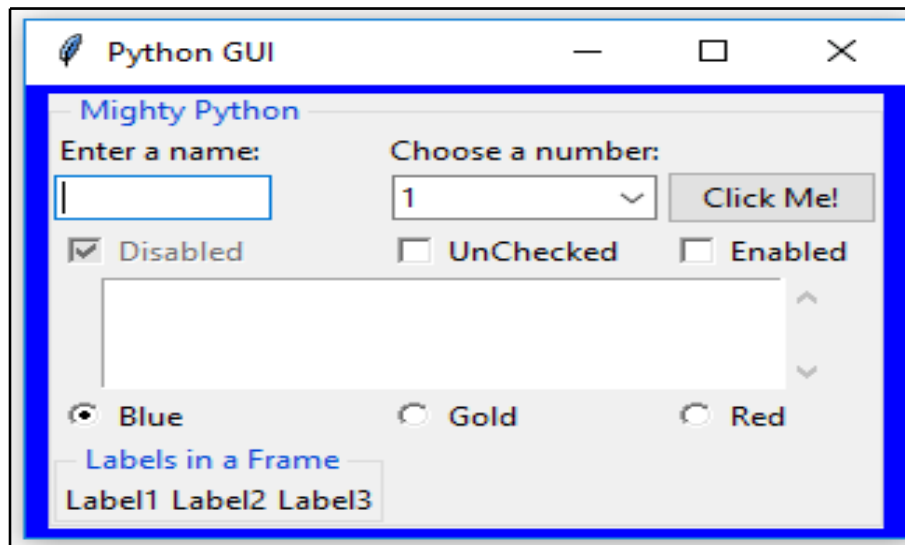
To align them to the left, we have to force our GUI layout by using the `sticky` property. By assigning it 'W' (west), we can force the widget to be left-aligned. Perform the following steps:

3.Run the code. This gives us the following GUI:



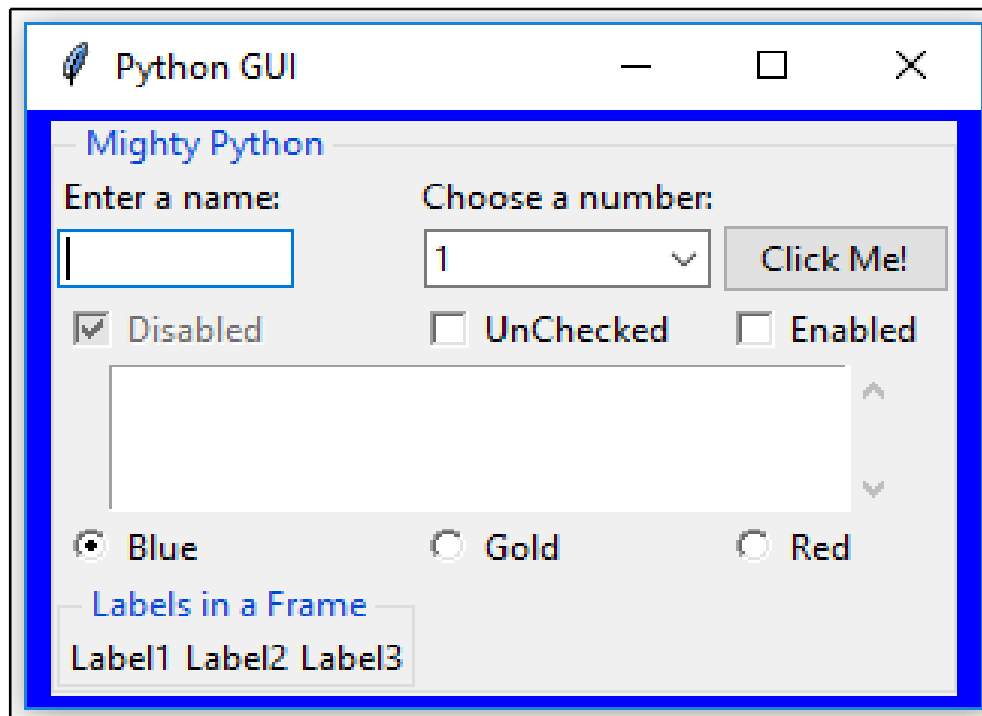
Let's align the **Entry** widget in column 0 to the left:

1. Open `GUI_embed_frames_align_west.py` and save it as `GUI_embed_frames_align_entry_west.py`.[#]
2. Use the `sticky` attribute to align the **Entry** left:



3. Run the `GUI_embed_frames_align_entry_west.py` file. Now both the label and the entry are aligned toward the west (left):

#



Creating menu bars

In this recipe, we will add a menu bar to our main window, add menus to the menu bar, and then add menu items to the menus.

Getting ready

We will start by learning how to add a menu bar, several menus, and a few menu items. In the beginning, clicking on a menu item will have no effect. We will add functionality to the menu items later, for example, closing the main window when clicking the **Exit** menu item and displaying a **Help | About** dialog.

We will continue to extend the GUI we created in the previous recipe, *Aligning GUI widgets by embedding frames within frames*.

How to do it...

To create a menu bar, follow these steps:

1. Open `GUI_embed_frames_align_entry_west.py` and save it as `GUI_menubar_file.py`.

2. Import the `Menu` class from `tkinter`:

```
import tkinter as tk

from tkinter import ttk
from tkinter import scrolledtext
from tkinter import Menu
```

How to do it...

3. Next, we will create the menu bar. Add the following code toward the bottom of the module, just above where we created the main event loop:

```
# Creating a Menu Bar
```

```
menu_bar = Menu(win)
```

```
win.config(menu=menu_bar)
```

```
# Create menu and add menu items
```

```
file_menu = Menu(menu_bar) # create File menu
```

```
file_menu.add_command(label="New") # add File
```

```
menu item
```

```
118 # Creating a Menu Bar
```

```
119 menu_bar = Menu(win)
```

```
120 win.config(menu=menu_bar)
```

```
121
```

```
122 # Create menu and add menu items
```

```
123 file_menu = Menu(menu_bar) # create File menu
```

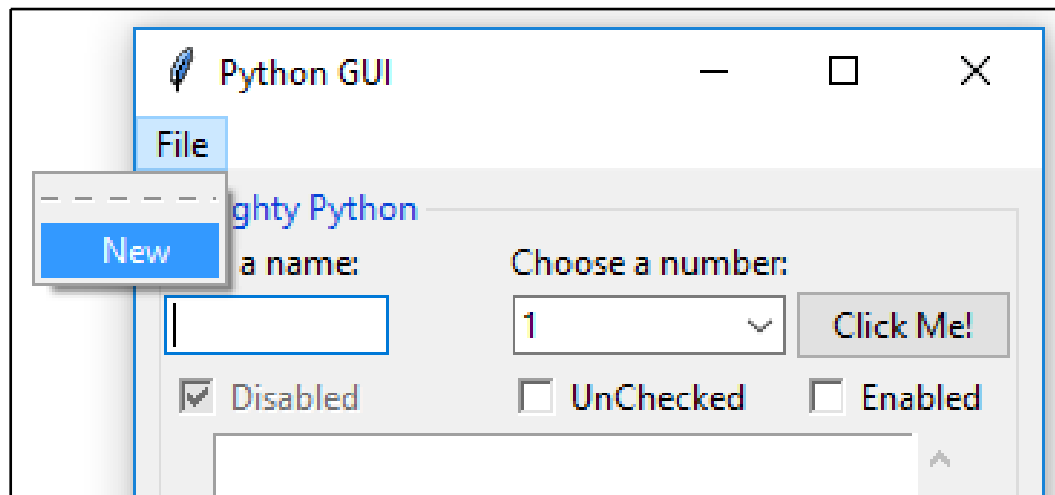
```
124 file_menu.add_command(label="New") # add File menu item
```

To make this work, we also have to add the menu to the menu bar and give it a label.

4. The menu item was already added to the menu, but we still have to add the menu to the menu bar:

```
menu_bar.add_cascade(label="File", menu=file_menu)  
# add File menu to menu bar and give it a label
```

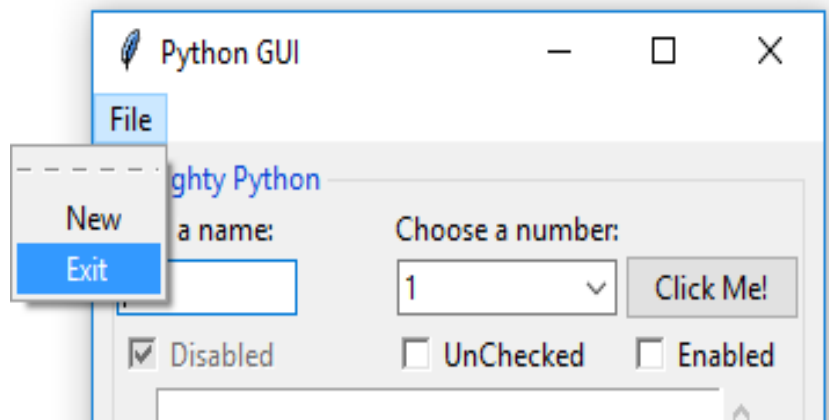
5. Running the preceding code adds a menu bar with a menu that has a menu item. This is shown in the following screenshot:



Next, we'll add a second menu item to the first menu that we added to the menu bar. This can be done by performing the following steps:

1. Open `GUI_menubar_file.py` and save it as `GUI_menubar_exit.py`.
2. Add the **Exit** menu item:
`file_menu.add_command(label="Exit")`

3. Running the preceding code produces the following result, that is, `GUI_menubar_exit.py`:



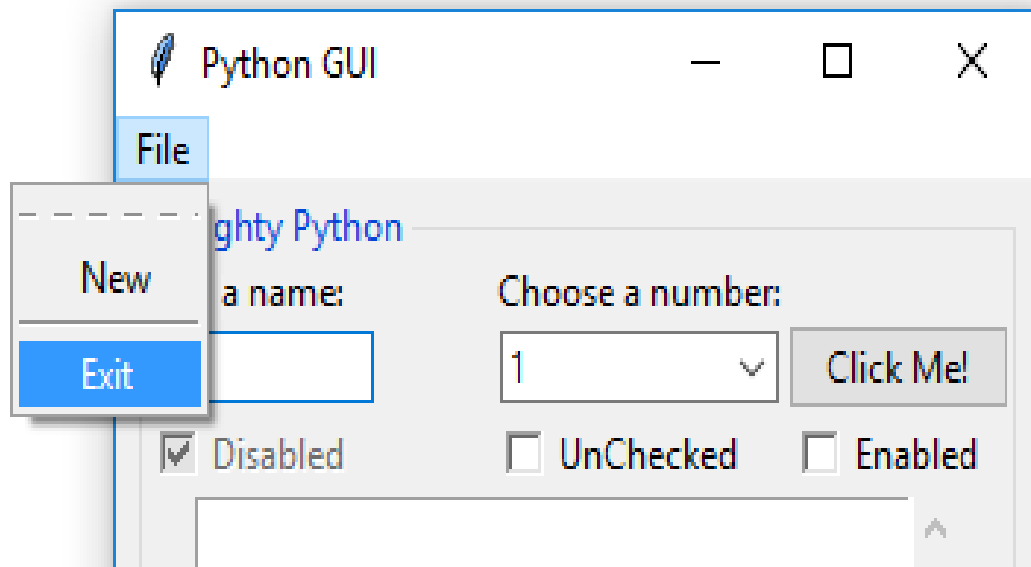
We can add separator lines between the menu items by adding a line of code in-between the existing menu items. This can be done by performing the following steps:

1. Open `GUI_menubar_exit.py` and save it as `GUI_menubar_separator.py`.

2. Add a separator, as follows:

```
file_menu.add_separator()
```

3. Run the preceding code. In the following screenshot, we can see that a separator line has been added in-between our two menu items:

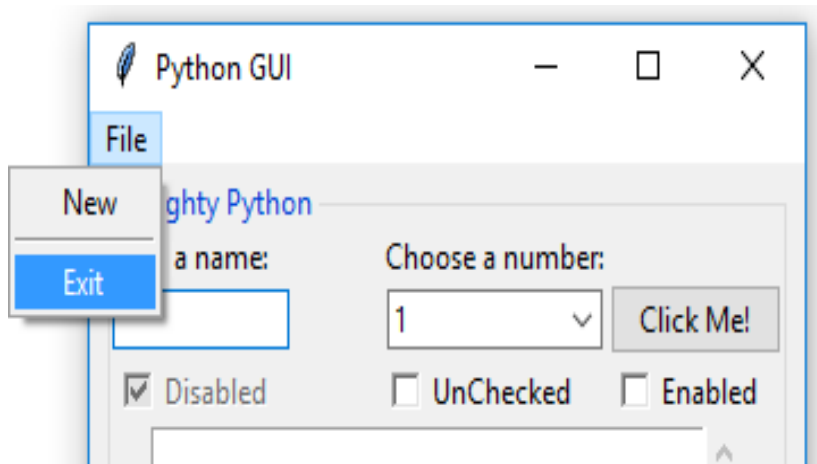


By passing in the `tearoff` property to the constructor of the menu, we can remove the first dashed line that, by default, appears above the first menu item in a menu. This can be done by performing the following steps:

1. Open `GUI_menubar_separator.py` and save it as `GUI_menubar_tearoff.py`.
2. Set the `tearoff` attribute to 0:

```
file_menu = Menu(menu_bar, tearoff=0)
```

3. Run the preceding code. In the following screenshot, the dashed line no longer appears, and our GUI looks so much better:



Next, we'll add a second menu, **Help**, which will be placed horizontally, to the right of the first menu. We'll give it one menu item, named **About**, and add this second menu to the menu bar.

File and **Help | About** are very common Windows GUI layouts we are all familiar with, and we can create these same menus using Python and `tkinter`:

1. Open `GUI_menubar_tearoff.py` and save it as `GUI_menubar_help.py`.
2. Add a second menu with a menu item:

```
help_menu = Menu(menu_bar, tearoff=0)
menu_bar.add_cascade(label="Help",
menu=help_menu)
help_menu.add_command(label="About")
```

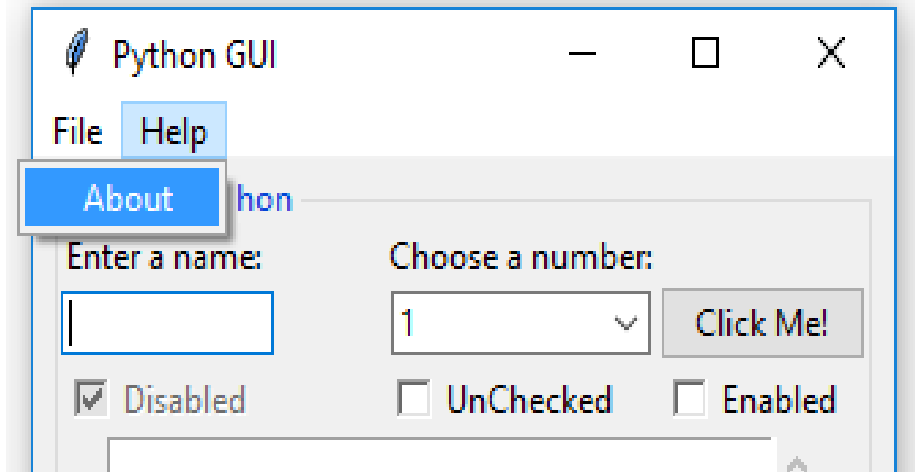
```
# Creating a Menu Bar
menu_bar = Menu(win)
win.config(menu=menu_bar)

# Add menu items
file_menu = Menu(menu_bar, tearoff=0)
file_menu.add_command(label="New")
file_menu.add_separator()
file_menu.add_command(label="Exit")
menu_bar.add_cascade(label="File", menu=file_menu)

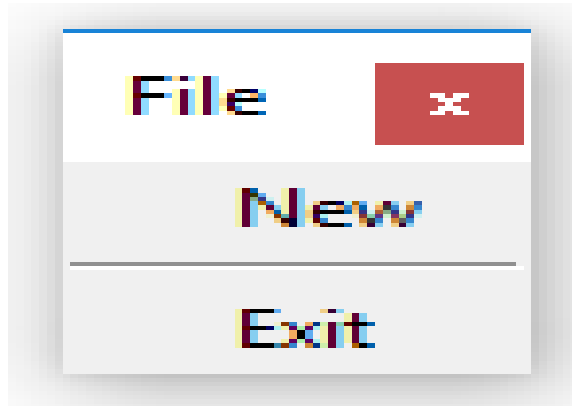
# Add another Menu to the Menu Bar and an item
help_menu = Menu(menu_bar, tearoff=0)
menu_bar.add_cascade(label="Help", menu=help_menu)
help_menu.add_command(label="About")

name_entered.focus()      # Place cursor into name Entry
#=====
# Start GUI
#=====
win.mainloop()
```

3. Run the preceding code. As shown in the following screenshot, we have a second menu with a menu item in the menu bar:



4. Run the code and click the **Exit** menu item. The following GUI shows the output of the code we run:



Creating tabbed widgets

In this recipe, we will create tabbed widgets to further organize our expanding GUI written in `tkinter`.

Getting ready

To improve our Python GUI using tabs, we will start at the beginning, using as little code as possible. In this recipe, we will create a simple GUI and then add widgets from the previous recipes, placing them in this new tabbed layout.

How to do it...

Follow these steps to create *Tab* controls, which in `tkinter` are called **Notebook**:

1. Create a new Python module and name it `GUI_tabbed.py`.
2. At the top of the module, import `tkinter`:

```
import tkinter as tk
from tkinter import ttk
```


3. Create an instance of the Tk class:

```
win = tk.Tk()
```

4. Add a title via the `title` attribute:

```
win.title("Python GUI")
```

5. Create `tabControl` using the `ttk Notebook`:

```
tabControl = ttk.Notebook(win)
```

6. Add the tab to `tabControl`:

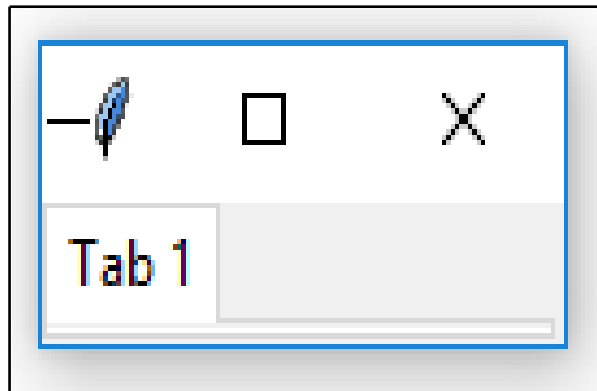
```
tabControl.add(tab1, text='Tab 1')
```

7. Use `pack` to make the control visible inside the GUI:

The preceding instructions produce the following code, which can be found in the `GUI_tabbed.py` file:

```
6 #=====
7 # imports
8 #=====
9 import tkinter as tk
10 from tkinter import ttk
11
12 win = tk.Tk()                # Create instance
13 win.title("Python GUI")     # Add a title
14 tabControl = ttk.Notebook(win) # Create Tab Control
15 tab1 = ttk.Frame(tabControl) # Create a tab
16 tabControl.add(tab1, text='Tab 1') # Add the tab
17 tabControl.pack(expand=1, fill="both") # Pack to make visible
18
19 #=====
20 # Start GUI
21 #=====
22 win.mainloop()
```

8. Run the preceding code. The following screenshot shows the GUI after running the code:

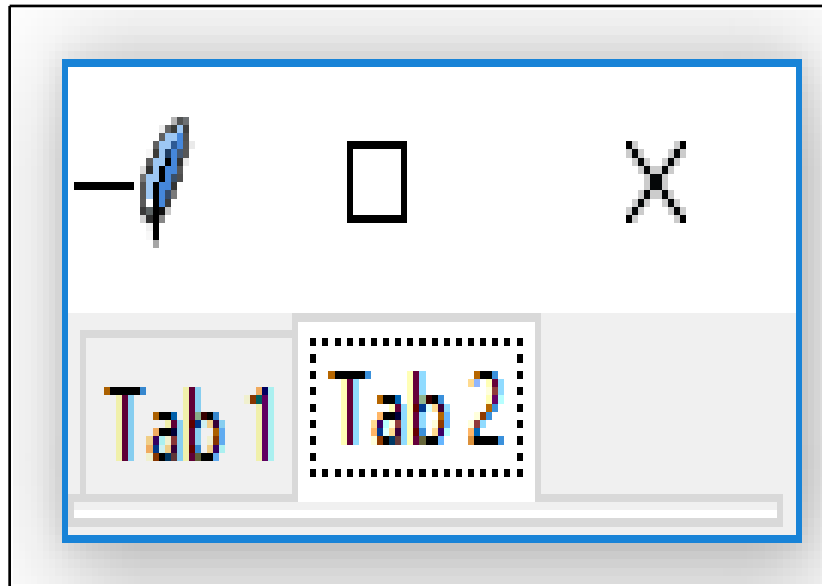


In the preceding code, we pack the `tabControl` and `ttk.Notebook` widgets into the main GUI form, expanding the notebook-tabbed control to fill in all the sides. We can add a second tab to our control and click between them by performing the following steps:

1. Open `GUI_tabbed.py` and save it as `GUI_tabbed_two.py`.
2. Add a second tab:

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

3. Run the preceding code. In the following screenshot, we have two tabs. Click on **Tab 2** to give it focus:



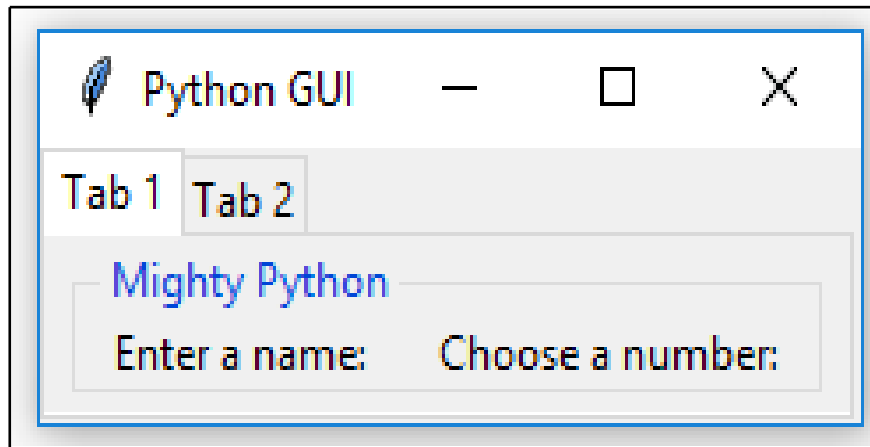
After adding a second label plus some spacing around them, we stretch the layout enough so we can see our GUI title again:

1. Open `GUI_tabbed_two_mighty.py` and save it as `GUI_tabbed_two_mighty_labels.py`.
2. Add a second label and spacing via a loop:

```
# Add another label
ttk.Label(mighty, text="Choose a
number:").grid(column=1, row=0)

# Add some space around each label
for child in mighty.winfo_children():
    child.grid_configure(padx=8)
```

3. Run the preceding code. The following screenshot shows the output from running this code, which can also be found in the `GUI_tabbed_two_mighty_labels.py` file:

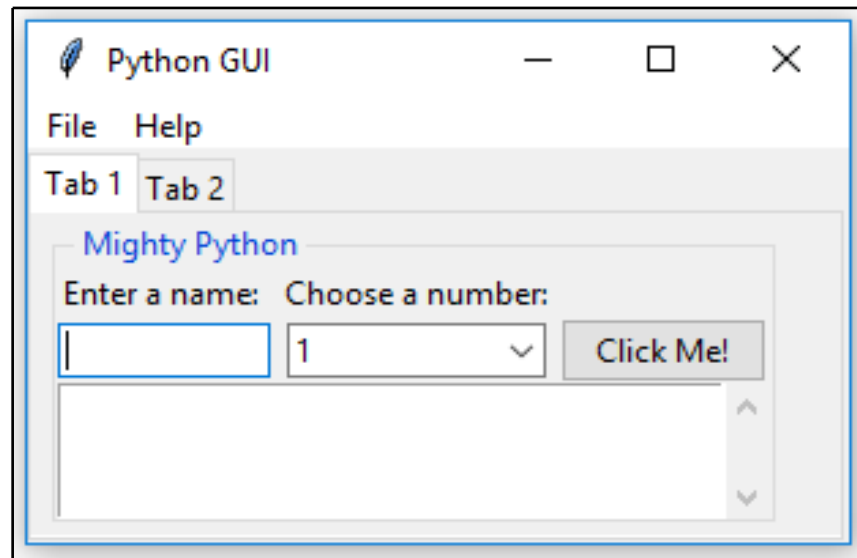


As you can see, all the widgets reside inside **Tab 1**. Let's move some of them to **Tab 2**:

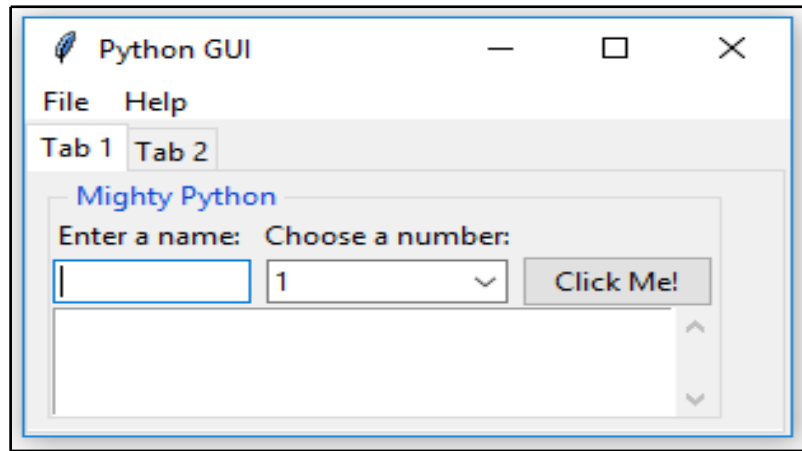
1. Create a second `LabelFrame`, which will be the container of the widgets we will be relocating to **Tab 2**:

```
mighty2 = ttk.LabelFrame(tab2, text=' The Snake ')  
mighty2.grid(column=0, row=0, padx=8, pady=4)
```

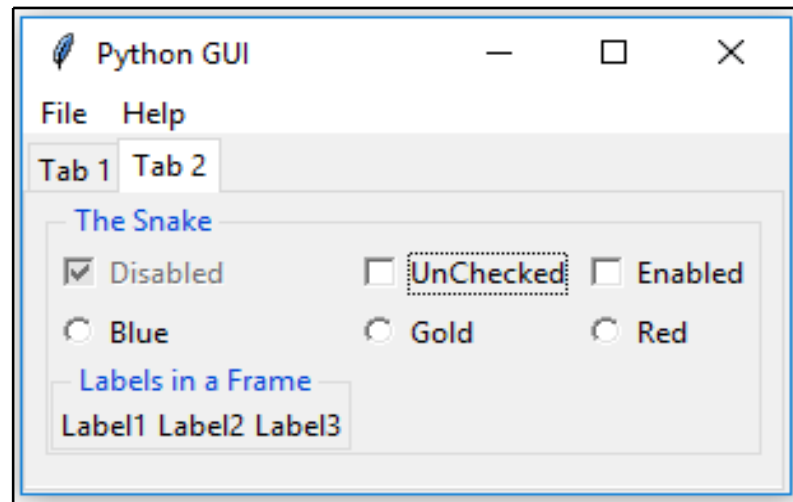
2. Next, we move the **Check** and **Radio** buttons to **Tab 2** by specifying the new parent container, which is a new variable that we name **mighty2**. The following is an example that we will apply to all the controls that relocate to **Tab 2**:



3. Run the `GUI_tabbed_all_widgets_both_tabs.py` file. The following screenshot shows the output we receive after running the preceding code:



We can now click on **Tab 2** and see our relocated widgets:



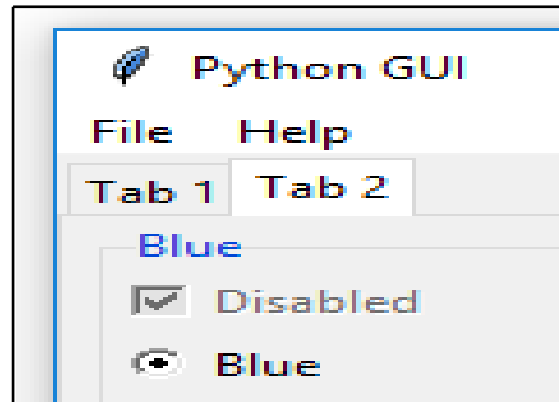
After running the preceding code, our GUI looks different. **Tab 1** has fewer widgets than it had before when it contained all of our previously created widgets.

Clicking the relocated **RadioButton** no longer has any effect, so we will change their actions to renaming the text attribute, from the title of the **LabelFrame** widget to the name the **Radiobuttons** display. When we click the **Gold RadioButton**, we no longer set the background of the frame to the color gold. Instead, we replace the **LabelFrame** text title. Python's **The Snake** now becomes **Gold**:

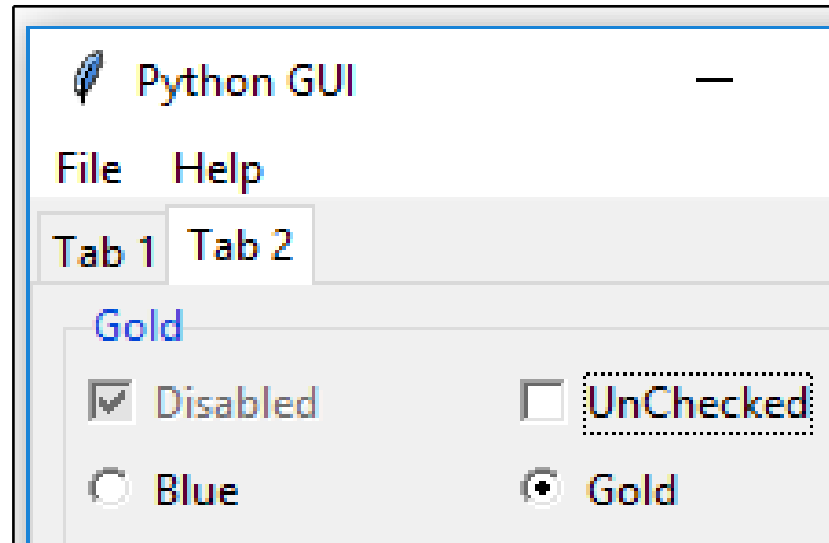
Def radCall():

```
radSel=radVar.get()
if radSel == 0:
    mighty2.configure(text='Blue')
if radSel == 1:
    mighty2.configure(text='Gold')
if radSel == 0:
    mighty2.configure(text='Red')
```

4. Now selecting any of the `RadioButton` widgets will change the name of the `LabelFrame`.
5. Run the `GUI_tabbed_all_widgets_both_tabs_radio.py` file. The following screenshot shows the output of running the code in this file:



Notice how the label frame is now titled **Blue**. Clicking on the **Gold** radio button changes this title to **Gold**, as shown in the following screenshot:



Using the grid layout manager

The grid layout manager is one of the most useful layout tools at our disposal. While layout tools such as `pack` are simple and easy to use, `grid` gives us a lot of control over our layout

–especially when we combine `grid` with embedded frames.

We have already used it in many recipes, for example, because it is just so powerful.

Getting ready...

In this recipe, we will review some grid layout manager techniques. We have already used them, but we will explore them in more detail here.

How to do it...

In this chapter, we have created rows and columns, which is the database approach to GUI design (MS Excel does the same). We hard-coded the first rows. However, if we forget to specify where we went the next row to reside, `tkinter` fills this in without us even noticing.

To observe this, let's take the code from a recipe we previously worked on:

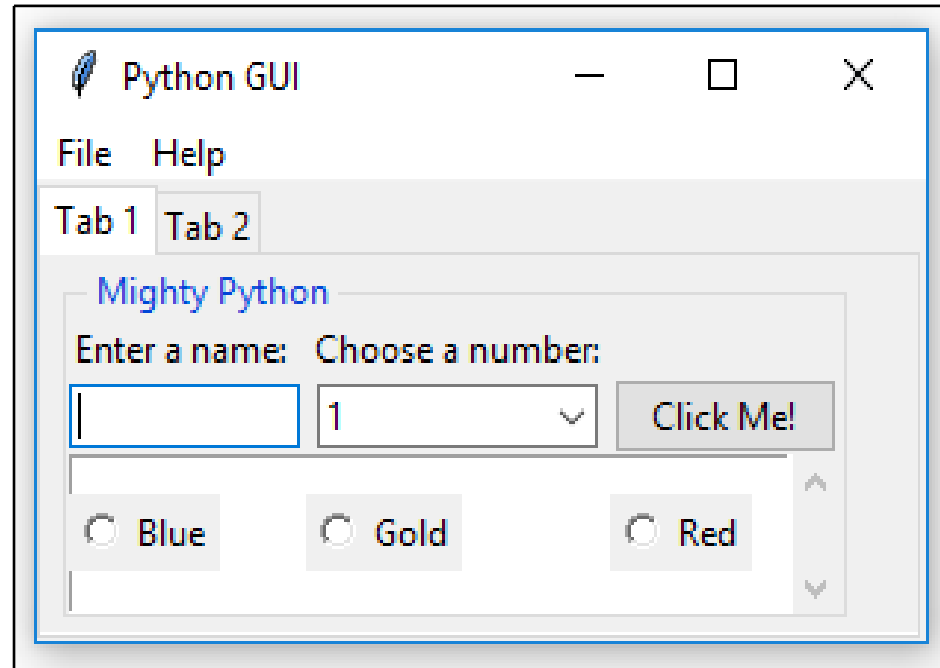
1. Open

`GUI_tabbed_all_widgets_both_tabs_radio.py`.

2. Comment out the `scr.grid` line, as follows:

```
# Using a scrolled Text control
scrol_w = 30
scrol_h = 3
scr = scrolledtext.ScrolledText(mighty, width=scrol_w, height=scrol_h, wrap=tk.WORD)
# scr.grid(column=0, row=2, sticky='WE', columnspan=3)
scr.grid(column=0, sticky='WE', columnspan=3) # row not specified
```

3. Run the code and notice how our radio buttons suddenly ended up in the middle of the Text widget!



We also used the `columnspan` property to make sure our widgets did not get limited to just one column, as shown in the following screenshot:

```
scr = scrolledtext.ScrolledText(mighty, width=scrol_w, height=scrol_h, wrap=tk.WORD)
scr.grid(column=0, row=5, sticky='WE', columnspan=3)
```