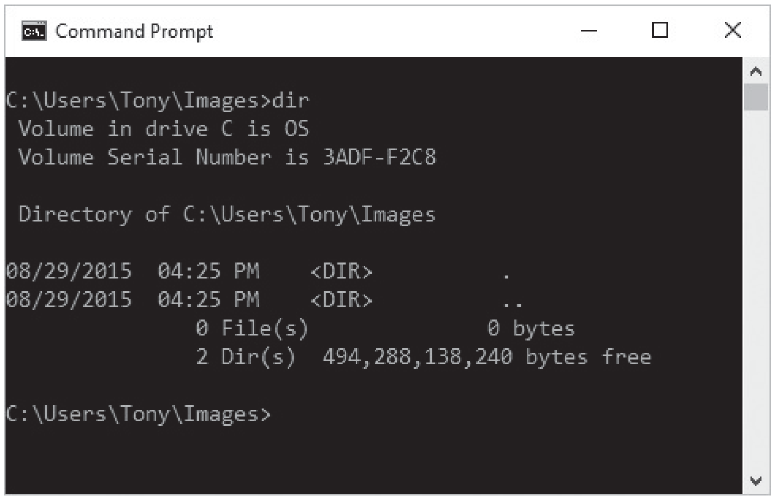
13.1 Graphical User Interfaces

Concept:

A graphical user interface allows the user to interact with the operating system and other programs using graphical elements such as icons, buttons, and dialog boxes.

A computer's *user interface* is the part of the computer with which the user interacts. One part of the user interface consists of hardware devices, such as the keyboard and the video display. Another part of the user interface lies in the way that the computer's operating system accepts commands from the user. For many years, the only way that the user could interact with an operating system was through a *command line interface,* such as the one shown in Figure 13-1. A command line interface typically displays a prompt, and the user types a command, which is then executed.

Figure 13-1 A command line interface



```
C:\Users\Tony\Images>dir
 Volume in drive C is OS
 Volume Serial Number is 3ADF-F2C8

 Directory of C:\Users\Tony\Images

08/29/2015  04:25 PM    <DIR>          .
08/29/2015  04:25 PM    <DIR>          ..
               0 File(s)              0 bytes
               2 Dir(s)  494,288,138,240 bytes free

C:\Users\Tony\Images>
```

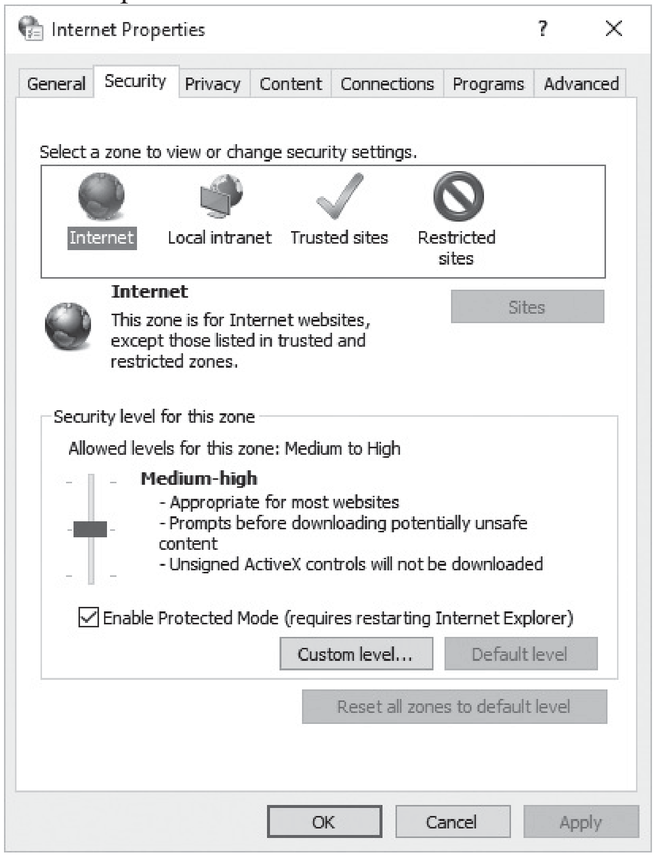
Many computer users, especially beginners, find command line interfaces difficult to use. This is because there are many commands to be learned, and each command has its own syntax, much like a programming statement. If a command isn't entered correctly, it will not work.

In the 1980s, a new type of interface known as a graphical user interface came into use in commercial operating systems. A *graphical user interface* (*GUI*; pronounced "gooey"), allows the user to interact with the operating system and other programs through graphical elements on the screen. GUIs also popularized the use of the mouse as an input device. Instead of requiring the user to type commands on the keyboard, GUIs allow the user to point at graphical elements and click the mouse button to activate them.

Much of the interaction with a GUI is done through *dialog boxes,* which are small windows that display information and allow the user to perform actions. Figure 13-2 shows an example of a dialog box from the Windows operating system that allows the user to change the system's Internet settings. Instead of typing commands according to a specified syntax, the user interacts with graphical elements such as icons, buttons, and slider bars.

Figure 13-2 A dialog box

▶ Description



Select a zone to view or change security settings.

Internet
This zone is for Internet websites, except those listed in trusted and restricted zones.

Security level for this zone
Allowed levels for this zone: Medium to High

Medium-high
- Appropriate for most websites
- Prompts before downloading potentially unsafe content
- Unsigned ActiveX controls will not be downloaded

☑ Enable Protected Mode (requires restarting Internet Explorer)

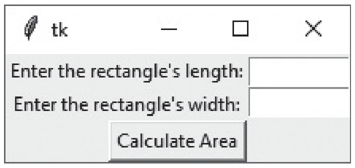
GUI Programs Are Event-Driven

In a text-based environment such as a command line interface, programs determine the order in which things happen. For example, consider a program that calculates the area of a rectangle. First, the program prompts the user to enter the rectangle's width. The user enters the width, then the program prompts the user to enter the rectangle's length. The user enters the length, then the program calculates the area. The user has no choice but to enter the data in the order that it is requested.

In a GUI environment, however, the user determines the order in which things happen. For example, Figure 13-3 shows a GUI program (written in Python) that calculates the area of a rectangle. The user can enter the length and the width in any order he or she wishes. If a mistake is made, the user can delete, the data that was entered and retype it. When the user is ready to calculate the area, he or she clicks the *Calculate Area* button, and the program performs the calculation. Because GUI programs must respond to the actions of the user, it is said that they are *event-driven*. The user causes events to take place, such as the clicking of a button, and the program must respond to the events.

Figure 13-3 A GUI program

▶ Description



Checkpoint

1. 13.1 What is a user interface?
2. 13.2 How does a command line interface work?
3. 13.3 When the user runs a program in a text-based environment, such as the command line, what determines the order in which things happen?
4. 13.4 What is an event-driven program?

13.2 Using the `tkinter` Module

Concept:

In Python, you can use the tkinter module to create simple GUI programs.

Python does not have GUI programming features built into the language itself. However, it comes with a module named `tkinter` that allows you to create simple GUI programs. The name “tkinter” is short for “Tk interface.” It is named this because it provides a way for Python programmers to use a GUI library named Tk. Many other programming languages use the Tk library as well.

Note:

There are numerous GUI libraries available for Python. Because the `tkinter` module comes with Python, we will use it only in this chapter.

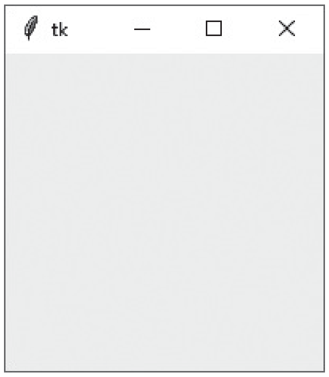
A GUI program presents a window with various graphical *widgets* with which the user can interact or view. The `tkinter` module provides 15 widgets, which are described in Table 13-1. We won’t cover all of the `tkinter` widgets in this chapter, but we will demonstrate how to create simple GUI programs that gather input and display data.

Table 13-1 `tkinter` widgets

Widget	Description
`Button`	A button that can cause an action to occur when it is clicked.
`Canvas`	A rectangular area that can be used to display graphics.
`Checkbutton`	A button that may be in either the “on” or “off” position.
`Entry`	An area in which the user may type a single line of input from the keyboard.
`Frame`	A container that can hold other widgets.
`Label`	An area that displays one line of text or an image.
`Listbox`	A list from which the user may select an item
`Menu`	A list of menu choices that are displayed when the user clicks a `Menubutton` widget.
`Menubutton`	A menu that is displayed on the screen and may be clicked by the user
`Message`	Displays multiple lines of text.
`Radiobutton`	A widget that can be either selected or deselected. `Radiobutton` widgets usually appear in groups and allow the user to select one of several options.
`Scale`	A widget that allows the user to select a value by moving a slider along a track.
`Scrollbar`	Can be used with some other types of widgets to provide scrolling ability.
`Text`	A widget that allows the user to enter multiple lines of text input.
`Toplevel`	A container, like a `Frame`, but displayed in its own window.

The simplest GUI program that we can demonstrate is one that displays an empty window. Program 13-1 shows how we can do this using the `tkinter` module. When the program runs, the window shown in Figure 13-4 is displayed. To exit the program, simply click the standard Windows close button (×) in the upper right corner of the window.

Figure 13-4 Window displayed by Program 13-1



Note:

Programs that use `tkinter` do not always run reliably under IDLE. This is because IDLE itself uses `tkinter`. You can always use IDLE's editor to write GUI programs, but for the best results, run them from your operating system's command prompt.

Program 13-1 (empty_window1.py)

```
 1  # This program displays an empty window.
 2
 3  import tkinter
 4
 5  def main():
 6      # Create the main window widget.
 7      main_window = tkinter.Tk()
 8
 9      # Enter the tkinter main loop.
10      tkinter.mainloop()
11
12  # Call the main function.
13  main()
```

Line 3 imports the `tkinter` module. Inside the `main` function, line 7 creates an instance of the `tkinter` module's `Tk` class and assigns it to the `main_window` variable. This object is the root widget, which is the main window in the program. Line 10 calls the `tkinter` module's `mainloop` function. This function runs like an infinite loop until you close the main window.

Most programmers prefer to take an object-oriented approach when writing a GUI program. Rather than writing a function to create the on-screen elements of a program, it is a common practice to write a class with an `__init__` method that builds the GUI. When an instance of the class is created, the GUI appears on the screen. To demonstrate, Program 13-2 shows an object-oriented version of our program that displays an empty window. When this program runs it displays the window shown in Figure 13-4.

Program 13-2 (`empty_window2.py`)

```
 1  # This program displays an empty window.
 2
 3  import tkinter
 4
 5  class MyGUI:
 6    def __init__(self):
 7      # Create the main window widget.
 8      self.main_window = tkinter.Tk()
 9
10      # Enter the tkinter main loop.
11      tkinter.mainloop()
12
13  # Create an instance of the MyGUI class.
14  my_gui = MyGUI()
```

Lines 5 through 11 are the class definition for the `MyGUI` class. The class's `__init__` method begins in line 6. Line 8 creates the root widget and assigns it to the class attribute `main_window`. Line 11 executes the `tkinter` module's `mainloop` function. The statement in line 14 creates an instance of the `MyGUI` class. This causes the class's `__init__` method to execute, displaying the empty window on the screen.

Checkpoint

1. 13.5 Briefly describe each of the following `tkinter` widgets:
    1. `Label`
    2. `Entry`
    3. `Button`
    4. `Frame`
2. 13.6 How do you create a root widget?
3. 13.7 What does the `tkinter` module's `mainloop` function do?

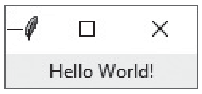
13.3 Display Text with `Label` Widgets

Concept:

You use the Label widget to display text in a window.

You can use a `Label` widget to display a single line of text in a window. To make a `Label` widget you create an instance of the `tkinter` module's `Label` class. Program 13-3 creates a window containing a `Label` widget that displays the text "Hello World!" The window is shown in Figure 13-5.

Figure 13-5 Window displayed by Program 13-3



Hello World!

VideoNote

Creating a Simple GUI application

Program 13-3 (`hello_world.py`)

```
 1  # This program displays a label with text.
 2
 3  import tkinter
 4
 5  class MyGUI:
 6    def __init__(self):
 7        # Create the main window widget.
 8        self.main_window = tkinter.Tk()
 9
10        # Create a Label widget containing the
11        # text 'Hello World!'
12        self.label = tkinter.Label(self.main_window,
13                                   text='Hello World!')
14
15        # Call the Label widget's pack method.
16        self.label.pack()
17
18        # Enter the tkinter main loop.
19        tkinter.mainloop()
20
21  # Create an instance of the MyGUI class.
22  my_gui = MyGUI()
```

The `MyGUI` class in this program is very similar to the one you saw previously in Program 13-2. Its `__init__` method builds the GUI when an instance of the class is created. Line 8 creates a root widget and assigns it to `self.main_window`. The following statement appears in lines 12 and 13:

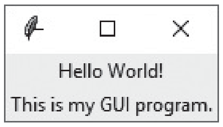
```
self.label = tkinter.Label(self.main_window,
                           text='Hello World!')
```

This statement creates a `Label` widget and assigns it to `self.label`. The first argument inside the parentheses is `self.main_window`, which is a reference to the root widget. This simply specifies that we want the `Label` widget to belong to the root widget. The second argument is `text='Hello World!'`. This specifies the text that we want to be displayed in the label.

The statement in line 16 calls the `Label` widget's `pack` method. The `pack` method determines where a widget should be positioned and makes the widget visible when the main window is displayed. (You call the `pack` method for each widget in a window.) Line 19 calls the `tkinter` module's `mainloop` method, which displays the program's main window, shown in Figure 13-5.

Let's look at another example. Program 13-4 displays a window with two `Label` widgets, shown in Figure 13-6.

Figure 13-6 Window displayed by Program 13-4



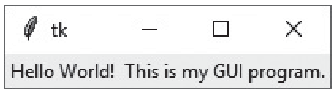
```
Hello World!
This is my GUI program.
```

Program 13-4 (hello_world2.py)

```
# This program displays two labels with text.

import tkinter

class MyGUI:
    def __init__(self):
        # Create the main window widget.
        self.main_window = tkinter.Tk()

        # Create two Label widgets.
        self.label1 = tkinter.Label(self.main_window,
                                    text='Hello World!')
        self.label2 = tkinter.Label(self.main_window,
                           text='This is my GUI program.')

        # Call both Label widgets' pack method.
        self.label1.pack()
        self.label2.pack()

        # Enter the tkinter main loop.
        tkinter.mainloop()

# Create an instance of the MyGUI class.
my_gui = MyGUI()
```

Notice the two `Label` widgets are displayed with one stacked on top of the other. We can change this layout by specifying an argument to `pack` method, as shown in Program 13-5. When the program runs, it displays the window shown in Figure 13-7.

Figure 13-7 Window displayed by Program 13-5



Hello World! This is my GUI program.

Program 13-5 (`hello_world3.py`)

```
 1  # This program uses the side='left' argument with
 2  # the pack method to change the layout of the widgets.
 3
 4  import tkinter
 5
 6  class MyGUI:
 7      def __init__(self):
 8          # Create the main window widget.
 9          self.main_window = tkinter.Tk()
10
11          # Create two Label widgets.
12          self.label1 = tkinter.Label(self.main_window,
13                                      text='Hello World!')
14          self.label2 = tkinter.Label(self.main_window,
15                        text='This is my GUI program.')
16
17          # Call both Label widgets' pack method.
18          self.label1.pack(side='left')
19          self.label2.pack(side='left')
20
21          # Enter the tkinter main loop.
22          tkinter.mainloop()
23
24  # Create an instance of the MyGUI class.
25  my_gui = MyGUI()
```

In lines 18 and 19, we call each `Label` widget's `pack` method passing the argument `side='left'`. This specifies that the widget should be positioned as far left as possible inside the parent widget. Because the `label1` widget was added to the `main_window` first, it will appear at the leftmost edge. The `label2` widget was added next, so it appears next to the `label1` widget. As a result, the labels appear side by side. The valid `side` arguments that you can pass to the `pack` method are `side='top'`, `side='bottom'`, `side='left'`, and `side='right'`.

Checkpoint

1. 13.8 What does a widget's `pack` method do?
2. 13.9 If you create two `Label` widgets and call their `pack` methods with no arguments, how will the `Label` widgets be arranged inside their parent widget?
3. 13.10 What argument would you pass to a widget's `pack` method to specify that it should be positioned as far left as possible inside the parent widget?

13.4 Organizing Widgets with `Frames`

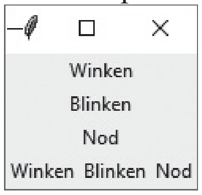
Concept:

A Frame is a container that can hold other widgets. You can use Frames to organize the widgets in a window.

A `Frame` is a container. It is a widget that can hold other widgets. `Frame`s are useful for organizing and arranging groups of widgets in a window. For example, you can place a set of widgets in one `Frame` and arrange them in a particular way, then place a set of widgets in another `Frame` and arrange them in a different way. Program 13-6 demonstrates this. When the program runs it, displays the window shown in Figure 13-8.

Figure 13-8 Window displayed by Program 13-6

▶ Description



Program 13-6 `(frame_demo.py)`

```
# This program creates labels in two different frames.

import tkinter

class MyGUI:
    def __init__(self):
        # Create the main window widget.
        self.main_window = tkinter.Tk()

        # Create two frames, one for the top of the
        # window, and one for the bottom.
        self.top_frame = tkinter.Frame(self.main_window)
        self.bottom_frame = tkinter.Frame(self.main_window)

        # Create three Label widgets for the
        # top frame.
        self.label1 = tkinter.Label(self.top_frame,
                                    text='Winken')
        self.label2 = tkinter.Label(self.top_frame,
                                    text='Blinken')
        self.label3 = tkinter.Label(self.top_frame,
                                    text='Nod')

        # Pack the labels that are in the top frame.
        # Use the side='top' argument to stack them
```

```
26          # one on top of the other.
27          self.label1.pack(side='top')
28          self.label2.pack(side='top')
29          self.label3.pack(side='top')
30
31          # Create three Label widgets for the
32          # bottom frame.
33          self.label4 = tkinter.Label(self.bottom_frame,
34                                      text='Winken')
35          self.label5 = tkinter.Label(self.bottom_frame,
36                                      text='Blinken')
37          self.label6 = tkinter.Label(self.bottom_frame,
38                                      text='Nod')
39
40          # Pack the labels that are in the bottom frame.
41          # Use the side='left' argument to arrange them
42          # horizontally from the left of the frame.
43          self.label4.pack(side='left')
44          self.label5.pack(side='left')
45          self.label6.pack(side='left')
46
47          # Yes, we have to pack the frames too!
48          self.top_frame.pack()
49          self.bottom_frame.pack()
50
51          # Enter the tkinter main loop.
52          tkinter.mainloop()
53
54  # Create an instance of the MyGUI class.
55  my_gui = MyGUI()
```

Take a closer look at lines 12 and 13:

```
self.top_frame = tkinter.Frame(self.main_window)
self.bottom_frame = tkinter.Frame(self.main_window)
```

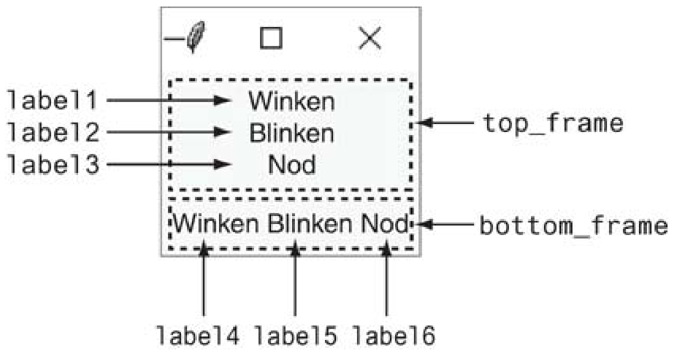
These lines create two `Frame` objects. The `self.main_window` argument that appears inside the parentheses cause the `Frames` to be added to the `main_window` widget.

Lines 17 through 22 create three `Label` widgets. Notice these widgets are added to the `self.top_frame` widget. Then, lines 27 through 29 call each of the `Label` widgets' `pack` method, passing `side='top'` as an argument. As shown in Figure 13-6, this causes the three widgets to be stacked one on top of the other inside the `Frame`.

Lines 33 through 38 create three more `Label` widgets. These `Label` widgets are added to the `self.bottom_frame` widget. Then, lines 43 through 45 call each of the `Label` widgets' `pack` method, passing `side='left'` as an argument. As shown in Figure 13-9, this causes the three widgets to appear horizontally inside the `Frame`.

Figure 13-9 Arrangement of widgets

▶ Description



Lines 48 and 49 call the `Frame` widgets' `pack` method, which makes the `Frame` widgets visible. Line 52 executes the `tkinter` module's `mainloop` function.

13.5 `Button` Widgets and Info Dialog Boxes

Concept:

You use the Button widget to create a standard button in a window. When the user clicks a button, a specified function or method is called.

An info dialog box is a simple window that displays a message to the user, and has an OK button that dismisses the dialog box. You can use the tkinter.messagebox module's showinfo function to display an info dialog box.

VideoNote

Responding to Button Clicks

A `Button` is a widget that the user can click to cause an action to take place. When you create a `Button` widget you can specify the text that is to appear on the face of the button and the name of a callback function. A *callback function* is a function or method that executes when the user clicks the button.

Note:

A callback function is also known as an *event handler* because it handles the event that occurs when the user clicks the button.

To demonstrate, we will look at Program 13-7. This program displays the window shown in Figure 13-10. When the user clicks the button, the program displays a separate *info dialog box*, shown in Figure 13-11. We use a function named `showinfo`, which is in the `tkinter.messagebox` module, to display the info dialog box. (To use the `showinfo` function, you will need to import the `tkinter.messagebox` module.) This is the general format of the `showinfo` function call:

```
tkinter.messagebox.showinfo(title, message)
```

In the general format, *`title`* is a string that is displayed in the dialog box's title bar, and *`message`* is an informational string that is displayed in the main part of the dialog box.

Figure 13-10 The main window displayed by Program 13-7

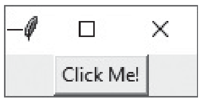
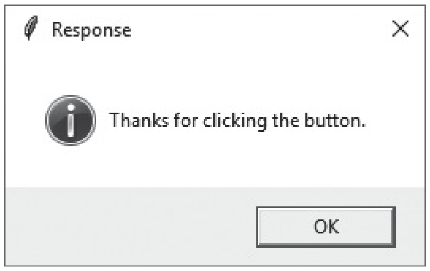


Figure 13-11 The info dialog box displayed by Program 13-7



Thanks for clicking the button.

Program 13-7 (button_demo.py)

```
 1  # This program demonstrates a Button widget.
 2  # When the user clicks the Button, an
 3  # info dialog box is displayed.
 4
 5  import tkinter
 6  import tkinter.messagebox
 7
 8  class MyGUI:
 9      def __init__(self):
10          # Create the main window widget.
11          self.main_window = tkinter.Tk()
12
13          # Create a Button widget. The text 'Click Me!'
14          # should appear on the face of the Button. The
15          # do_something method should be executed when
16          # the user clicks the Button.
17          self.my_button = tkinter.Button(self.main_window,
18                                       text='Click Me!',
19                                       command=self.do_something)
20
21          # Pack the Button.
22          self.my_button.pack()
23
24          # Enter the tkinter main loop.
25          tkinter.mainloop()
26
27    # The do_something method is a callback function
28    # for the Button widget.
29
30    def do_something(self):
31          # Display an info dialog box.
32          tkinter.messagebox.showinfo('Response',
33                                    'Thanks for clicking the button.')
34
35  # Create an instance of the MyGUI class.
36  my_gui = MyGUI()
```

Line 5 imports the `tkinter` module, and line 6 imports the `tkinter.messagebox` module. Line 11 creates the root widget and assigns it to the `main_window` variable.

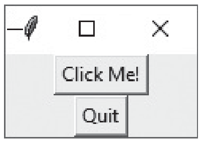
The statement in lines 17 through 19 creates the `Button` widget. The first argument inside the parentheses is `self.main_window`, which is the parent widget. The `text='Click Me!'` argument specifies that the string ‘Click Me!’ should appear on the face of the button. The `command='self.do_something'` argument specifies the class's `do_something` method as the callback function. When the user clicks the button, the `do_something` method will execute.

The `do_something` method appears in lines 31 through 33. The method simply calls the `tkinter.messagebox.showinfo` function to display the info box shown in Figure 13-11. To dismiss the dialog box, the user can click the OK button.

Creating a Quit Button

GUI programs usually have a *Quit button* (or an Exit button) that closes the program when the user clicks it. To create a Quit button in a Python program, you simply create a `Button` widget that calls the root widget's `destroy` method as a callback function. Program 13-8 demonstrates how to do this. It is a modified version of Program 13-7, with a second `Button` widget added as shown in Figure 13-12.

Figure 13-12 The info dialog box displayed by Program 13-8



Program 13-8 (`quit_button.py`)

```
# This program has a Quit button that calls
# the Tk class's destroy method when clicked.

import tkinter
import tkinter.messagebox

class MyGUI:
    def __init__(self):
        # Create the main window widget.
        self.main_window = tkinter.Tk()

        # Create a Button widget. The text 'Click Me!'
        # should appear on the face of the Button. The
        # do_something method should be executed when
        # the user clicks the Button.
        self.my_button = tkinter.Button(self.main_window,
                                        text='Click Me!',
                                        command=self.do_something)

        # Create a Quit button. When this button is clicked
        # the root widget's destroy method is called.
        # (The main_window variable references the root widget,
        # so the callback function is self.main_window.destroy.)
        self.quit_button = tkinter.Button(self.main_window,
                                          text='Quit',
                                           command=self.main_window.destroy)


```

```
29          # Pack the Buttons.
30          self.my_button.pack()
31          self.quit_button.pack()
32
33          # Enter the tkinter main loop.
34          tkinter.mainloop()
35
36      # The do_something method is a callback function
37      # for the Button widget.
38
39      def do_something(self):
40          # Display an info dialog box.
41          tkinter.messagebox.showinfo('Response',
42                                     'Thanks for clicking the button.')
43
44  # Create an instance of the MyGUI class.
45  my_gui = MyGUI()
```

The statement in lines 24 through 26 creates the Quit button. Notice the `self.main_window.destroy` method is used as the callback function. When the user clicks the button, this method is called and the program ends.

13.6 Getting Input with the Entry Widget

Concept:

An `Entry` widget is a rectangular area that the user can type input into. You use the `Entry` widget's `get` method to retrieve the data that has been typed into the widget.

An `Entry` widget is a rectangular area that the user can type text into. Entry widgets are used to gather input in a GUI program. Typically, a program will have one or more `Entry` widgets in a window, along with a button that the user clicks to submit the data that he or she has typed into the `Entry` widgets. The button's callback function retrieves data from the window's `Entry` widgets and processes it.

You use an `Entry` widget's `get` method to retrieve the data that the user has typed into the widget. The `get` method returns a string, so it will have to be converted to the appropriate data type if the `Entry` widget is used for numeric input.

To demonstrate, we will look at a program that allows the user to enter a distance in kilometers into an `Entry` widget then click a button to see that distance converted to miles. The formula for converting kilometers to miles is:

Miles = Kilometers × 0.6214

Figure 13-13 shows the window that the program displays. To arrange the widgets in the positions shown in the figure, we will organize them in two frames, as shown in Figure 13-14. The label that displays the prompt and the `Entry` widget will be stored in the `top_frame`, and their `pack` methods will be called with the `side='left'` argument. This will cause them to appear horizontally in the frame. The Convert button and the Quit button will be stored in the `bottom_frame`, and their `pack` methods will also be called with the `side='left'` argument.

Figure 13-13 The `kilo_converter` program's window

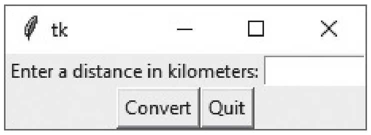
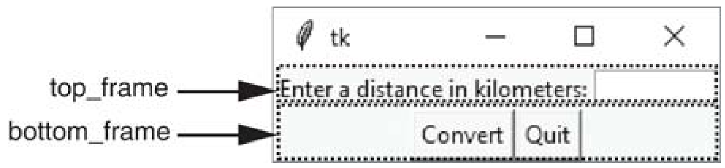


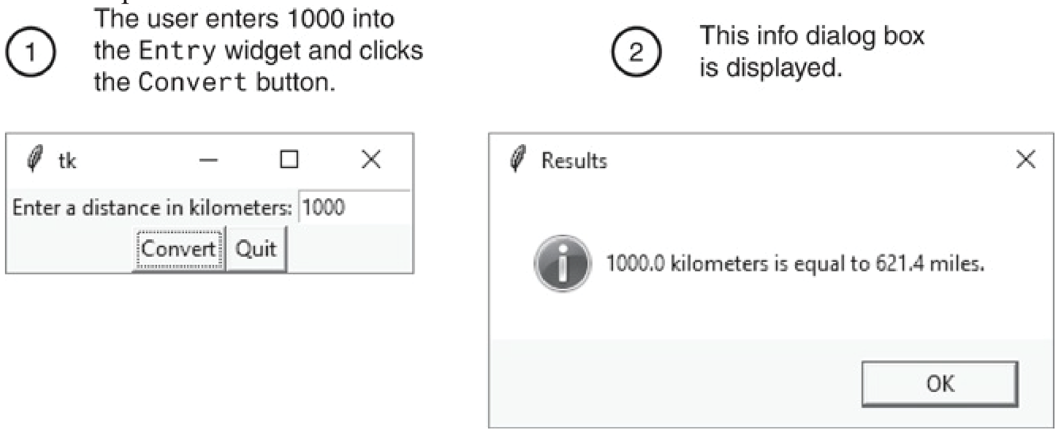
Figure 13-14 The window organized with frames



Program 13-9 shows the code for the program. Figure 13-15 shows what happens when the user enters 1000 into the `Entry` widget and clicks the Convert button.

Figure 13-15 The info dialog box

▶ Description


1
The user enters 1000 into the Entry widget and clicks the Convert button.
2
This info dialog box is displayed.
tk
Enter a distance in kilometers:
1000
Convert
Quit
Results
OK

1000.0 kilometers is equal to 621.4 miles.

Program 13-9 (`kilo_converter.py`)

```
# This program converts distances in kilometers
# to miles. The result is displayed in an info
# dialog box.

import tkinter
import tkinter.messagebox

class KiloConverterGUI:
    def __init__(self):

        # Create the main window.
        self.main_window = tkinter.Tk()

        # Create two frames to group widgets.
        self.top_frame = tkinter.Frame(self.main_window)
        self.bottom_frame = tkinter.Frame(self.main_window)

        # Create the widgets for the top frame.
        self.prompt_label = tkinter.Label(self.top_frame,
                    text='Enter a distance in kilometers:')
        self.kilo_entry = tkinter.Entry(self.top_frame,
                                        width=10)

        # Pack the top frame's widgets.
        self.prompt_label.pack(side='left')
        self.kilo_entry.pack(side='left')

        # Create the button widgets for the bottom frame.
        self.calc_button = tkinter.Button(self.bottom_frame,
                                          text='Convert',
31                                         command=self.convert)
32          self.quit_button = tkinter.Button(self.bottom_frame,
33                                         text='Quit',
34                                        command=self.main_window.destroy)
35          # Pack the buttons.
36          self.calc_button.pack(side='left')
37          self.quit_button.pack(side='left')
38
39          # Pack the frames.
40          self.top_frame.pack()
41          self.bottom_frame.pack()
42
43          # Enter the tkinter main loop.
44          tkinter.mainloop()
45
46      # The convert method is a callback function for
47      # the Calculate button.
48
49      def convert(self):
50          # Get the value entered by the user into the
51          # kilo_entry widget.
52          kilo = float(self.kilo_entry.get())
53
54          # Convert kilometers to miles.
55          miles = kilo * 0.6214
56
57          # Display the results in an info dialog box.
58          tkinter.messagebox.showinfo('Results',
59                                    str(kilo) +
60                                   ' kilometers is equal to ' +
61                                    str(miles) + ' miles.')
62
63  # Create an instance of the KiloConverterGUI class.
64  kilo_conv = KiloConverterGUI()
```

The `convert` method, shown in lines 49 through 60 is the Convert button's callback function. The statement in line 52 calls the `kilo_entry` widget's `get` method to retrieve the data that has been typed into the widget. The value is converted to a `float` then assigned to the `kilo` variable. The calculation in line 55 performs the conversion and assigns the results to the `miles` variable. Then, the statement in lines 58 through 61 displays the info dialog box with a message that gives the converted value.

13.7 Using Labels as Output Fields

Concept:

When a StringVar object is associated with a Label widget, the Label widget displays any data that is stored in the StringVar object.

Previously, you saw how to use an info dialog box to display output. If you don't want to display a separate dialog box for your program's output, you can use `Label` widgets in the program's main window to dynamically display output. You simply create empty `Label` widgets in your main window, then write code that displays the desired data in those labels when a button is clicked.

The `tkinter` module provides a class named `StringVar` that can be used along with a `Label` widget to display data. First, you create a `StringVar` object. Then, you create a `Label` widget and associate it with the `StringVar` object. From that point on, any value that is then stored in the `StringVar` object will automatically be displayed in the `Label` widget.

Program 13-10 demonstrates how to do this. It is a modified version of the `kilo_converter` program that you saw in Program 13-9. Instead of popping up an info dialog box, this version of the program displays the number of miles in a label in the main window.

Program 13-10 `(kilo_converter2.py)`

```
# This program converts distances in kilometers
# to miles. The result is displayed in a label
# on the main window.

import tkinter

class KiloConverterGUI:
    def __init__(self):

        # Create the main window.
        self.main_window = tkinter.Tk()

        # Create three frames to group widgets.
        self.top_frame = tkinter.Frame()
        self.mid_frame = tkinter.Frame()
        self.bottom_frame = tkinter.Frame()

        # Create the widgets for the top frame.
        self.prompt_label = tkinter.Label(self.top_frame,
                    text='Enter a distance in kilometers:')
        self.kilo_entry = tkinter.Entry(self.top_frame,
                                        width=10)

        # Pack the top frame's widgets.
        self.prompt_label.pack(side='left')
        self.kilo_entry.pack(side='left')

        # Create the widgets for the middle frame.
        self.descr_label = tkinter.Label(self.mid_frame,
                                         text='Converted to miles:')

        # We need a StringVar object to associate with
        # an output label. Use the object's set method
        # to store a string of blank characters.
        self.value = tkinter.StringVar()

        # Create a label and associate it with the
        # StringVar object. Any value stored in the
        # StringVar object will automatically be displayed
        # in the label.
        self.miles_label = tkinter.Label(self.mid_frame,
                          textvariable=self.value)

        # Pack the middle frame's widgets.
        self.descr_label.pack(side='left')
        self.miles_label.pack(side='left')

        # Create the button widgets for the bottom frame.
        self.calc_button = tkinter.Button(self.bottom_frame,
                                       text='Convert',
                                       command=self.convert)
        self.quit_button = tkinter.Button(self.bottom_frame,
                                       text='Quit',
                                       command=self.main_window.destroy)

        # Pack the buttons.
        self.calc_button.pack(side='left')
        self.quit_button.pack(side='left')

        # Pack the frames.
        self.top_frame.pack()
        self.mid_frame.pack()
        self.bottom_frame.pack()

        # Enter the tkinter main loop.
        tkinter.mainloop()

    # The convert method is a callback function for
    # the Calculate button.

    def convert(self):
        # Get the value entered by the user into the
        # kilo_entry widget.
        kilo = float(self.kilo_entry.get())

        # Convert kilometers to miles.
        miles = kilo * 0.6214

        # Convert miles to a string and store it
        # in the StringVar object. This will automatically
        # update the miles_label widget.
        self.value.set(miles)

# Create an instance of the KiloConverterGUI class.
kilo_conv = KiloConverterGUI()
```

When this program runs, it displays the window shown in Figure 13-16. Figure 13-17 shows what happens when the user enters 1000 for the kilometers and clicks the Convert button. The number of miles is displayed in a label in the main window.

Figure 13-16 The window initially displayed

▶ Description

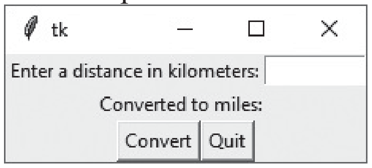
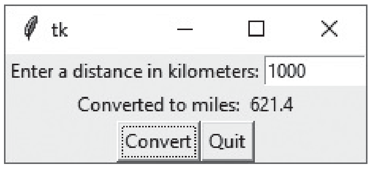


Figure 13-17 The window showing 1000 kilometers converted to miles


tk
Enter a distance in kilometers: 1000
Converted to miles: 621.4
Convert
Quit

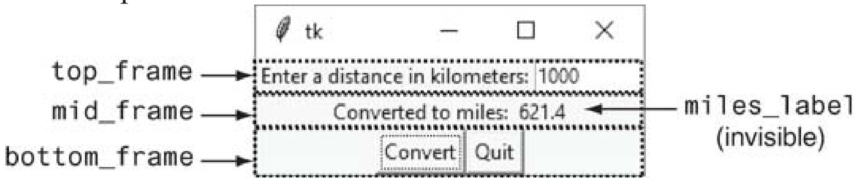
Let's look at the code. Lines 14 through 16 create three frames: `top_frame`, `mid_frame`, and `bottom_frame`. Lines 19 through 26 create the widgets for the top frame and call their `pack` method.

Lines 29 through 30 create the Label widget with the text `'Converted to miles:'` that you see on the main window in Figure 13-16. Then, line 35 creates a `StringVar` object and assigns it to the `value` variable. Line 41 creates a `Label` widget named `miles_label` that we will use to display the number of miles. Notice in line 42, we use the argument `textvariable=self.value`. This creates an association between the `Label` widget and the `StringVar` object that is referenced by the `value` variable. Any value that we store in the `StringVar` object will be displayed in the label.

Lines 45 and 46 pack the two `Label` widgets that are in the `mid_frame`. Lines 49 through 58 create the `Button` widgets and `pack` them. Lines 61 through 63 pack the `Frame` objects. Figure 13-18 shows how the various widgets in this window are organized in the three frames.

Figure 13-18 Layout of the `kilo_converter2` program's main window

▶ Description



The `convert` method, shown in lines 71 through 82 is the Convert button's callback function. The statement in line 74 calls the `kilo_entry` widget's `get` method to retrieve the data that has been typed into the widget. The value is converted to a `float` then assigned to the `kilo` variable. The calculation in line 77 performs the conversion and assigns the results to the `miles` variable. Then the statement in line 82 calls the `StringVar` object's `set` method, passing `miles` as an argument. This stores the value referenced by `miles` in the `StringVar` object, and also causes it to be displayed in the `miles_label` widget.

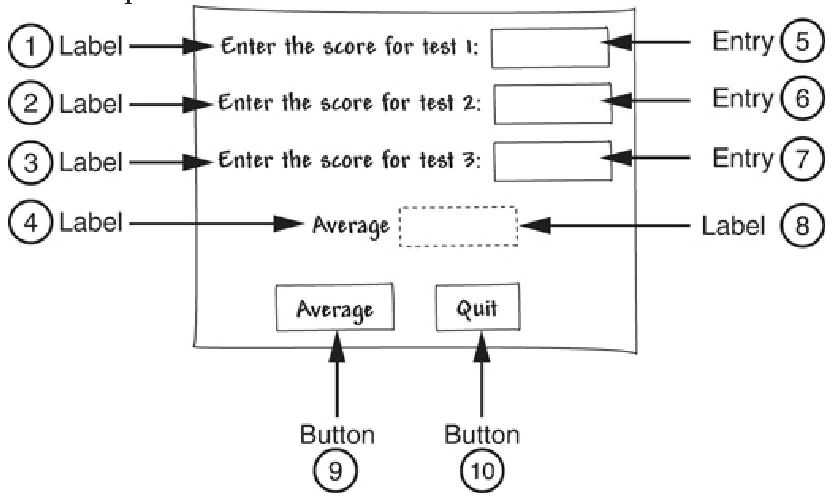
In the Spotlight: Creating a GUI Program

Kathryn teaches a science class. In Chapter 3, we stepped through the development of a program that her students can use to calculate the average of three test scores. The program prompts the student to enter each score, then it displays the average. She has asked you to design a GUI program that performs a similar operation. She would like the program to have three `Entry` widgets into which the test scores can be entered, and a button that causes the average to be displayed when clicked.

Before we begin writing code, it will be helpful if we draw a sketch of the program's window, as shown in Figure 13-19. The sketch also shows the type of each widget. (The numbers that appear in the sketch will help us when we make a list of all the widgets.)

Figure 13-19 A sketch of the window

▶ Description



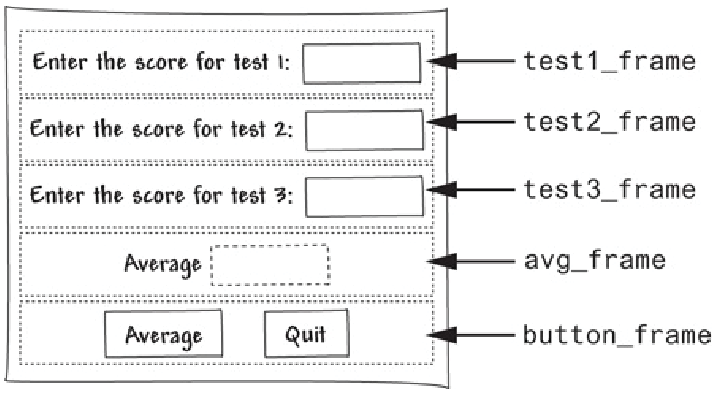
By examining the sketch, we can make a list of the widgets that we need. As we make the list, we will include a brief description of each widget, and a name that we will assign to each widget when we construct it.

Widget Number in Figure 13-19	Widget Type	Description	Name
1	Label	Instructs the user to enter the score for test 1.	`test1_label`
2	Label	Instructs the user to enter the score for test 2.	`test2_label`
3	Label	Instructs the user to enter the score for test 3.	`test3_label`
4	Label	Identifies the average, which will be displayed next to this label.	`result_label`
5	Entry	This is where the user will enter the score for test 1.	`test1_entry`
6	Entry	This is where the user will enter the score for test 2.	`test2_entry`
7	Entry	This is where the user will enter the score for test 3.	`test3_entry`

8	Label	The program will display the average test score in this label.	`avg_label`
9	Button	When this button is clicked, the program will calculate the average test score and display it in the `averageLabel` component.	`calc_button`
10	Button	When this button is clicked the program will end.	`quit_button`

We can see from the sketch that we have five rows of widgets in the window. To organize them, we will also create five `Frame` objects. Figure 13-20 shows how we will position the widgets inside the five `Frame` objects.

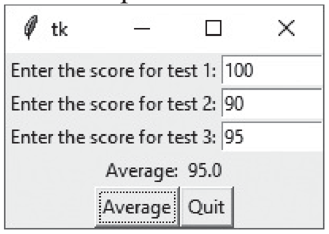
Figure 13-20 Using Frames to organize the widgets


Enter the score for test 1:
Enter the score for test 2:
Enter the score for test 3:
Average
Average
Quit
test1_frame
test2_frame
test3_frame
avg_frame
button_frame

Program 13-11 shows the code for the program, and Figure 13-21 shows the program's window with data entered by the user.

Figure 13-21 The `test_averages` program window

▶ Description



Enter the score for test 1: 100
Enter the score for test 2: 90
Enter the score for test 3: 95
Average: 95.0

Program 13-11 `(test_averages.py)`

```
# This program uses a GUI to get three test
# scores and display their average.

import tkinter
```

```

class TestAvg:
    def __init__(self):
        # Create the main window.
        self.main_window = tkinter.Tk()

        # Create the five frames.
        self.test1_frame = tkinter.Frame(self.main_window)
        self.test2_frame = tkinter.Frame(self.main_window)
        self.test3_frame = tkinter.Frame(self.main_window)
        self.avg_frame = tkinter.Frame(self.main_window)
        self.button_frame = tkinter.Frame(self.main_window)

        # Create and pack the widgets for test 1.
        self.test1_label = tkinter.Label(self.test1_frame,
                                         text='Enter the score for test 1:')
        self.test1_entry = tkinter.Entry(self.test1_frame,
                                         width=10)
        self.test1_label.pack(side='left')
        self.test1_entry.pack(side='left')

        # Create and pack the widgets for test 2.
        self.test2_label = tkinter.Label(self.test2_frame,
                                         text='Enter the score for test 2:')
        self.test2_entry = tkinter.Entry(self.test2_frame,
                                         width=10)
        self.test2_label.pack(side='left')
        self.test2_entry.pack(side='left')

        # Create and pack the widgets for test 3.
        self.test3_label = tkinter.Label(self.test3_frame,
                                         text='Enter the score for test 3:')
        self.test3_entry = tkinter.Entry(self.test3_frame,
                                         width=10)
        self.test3_label.pack(side='left')
        self.test3_entry.pack(side='left')

        # Create and pack the widgets for the average.
        self.result_label = tkinter.Label(self.avg_frame,
                                          text='Average:')
        self.avg = tkinter.StringVar() # To update avg_label
        self.avg_label = tkinter.Label(self.avg_frame,
                                       textvariable=self.avg)
        self.result_label.pack(side='left')
        self.avg_label.pack(side='left')

        # Create and pack the button widgets.
        self.calc_button = tkinter.Button(self.button_frame,
                                          text='Average',
                                          command=self.calc_avg)
        self.quit_button = tkinter.Button(self.button_frame,
                                          text='Quit',
                                          command=self.main_window.destroy)
        self.calc_button.pack(side='left')
        self.quit_button.pack(side='left')

        # Pack the frames.
        self.test1_frame.pack()
        self.test2_frame.pack()
        self.test3_frame.pack()
        self.avg_frame.pack()
        self.button_frame.pack()

        # Start the main loop.
        tkinter.mainloop()

    # The calc_avg method is the callback function for
    # the calc_button widget.

    def calc_avg(self):
        # Get the three test scores and store them
        # in variables.
        self.test1 = float(self.test1_entry.get())
        self.test2 = float(self.test2_entry.get())
        self.test3 = float(self.test3_entry.get())

        # Calculate the average.
        self.average = (self.test1 + self.test2 +
                        self.test3) / 3.0

        # Update the avg_label widget by storing
        # the value of self.average in the StringVar
        # object referenced by avg.
        self.avg.set(self.average)

# Create an instance of the TestAvg class.
test_avg = TestAvg()
```

Checkpoint

1. 13.11 How do you retrieve data from an `Entry` widget?
2. 13.12 When you retrieve a value from an `Entry` widget, of what data type is it?
3. 13.13 What module is the `StringVar` class in?
4. 13.14 What can you accomplish by associating a `StringVar` object with a `Label` widget?