6 Event-Driven Programming

When a program runs in Python the Python interpreter scans the program from top to bottom executing the first statement that is not part of a function definition. The program proceeds by executing the next statement and the next. Sequential execution is redirected by iteration (i.e. *for* and *while* loops) and function calls. Nevertheless, the program sequentially executes until Python interprets the last statement at which point the program terminates.

In an event-driven program sequential execution is in response to events happening while the program is executing. Event-driven programs arise in many areas of programming including Operating Systems, Internet Programming, Distributed Computing, and Graphical User Interfaces, often abbreviated GUI programs. An event-driven application begins as a sequential program executing one statement after another until it enters a never-ending loop. This loop, sometimes called the event dispatch loop looks for an incoming event and then dispatches that event to an event handler. Events come in a wide variety of flavors including:

- An interrupt indicating the completion of a disk operation
- A network packet has become available
- A network connection has become unavailable
- A button was pressed in a GUI application
- A menu item was selected in a GUI application
- An incoming request has been received by a web server.

In an event-driven program, the event dispatch loop looks for events like these. Each event will generally have its own event handler. An event handler is a function that is called to process the event. Each time an event is found, the corresponding event handler is called to process the event. Once the event is processed, the program returns from the event handler to the event dispatch loop to look for the next event. This process repeats forever or until some event is dispatched that causes the program to terminate. For example, if a user chooses to exit a GUI application, the event handler may tell the the event dispatch loop to quit and exit.

K.D. Lee, *Python Programming Fundamentals*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6642-9_6

Tk is a powerful Application Programming Interface, or API, designed to make GUI programming easy on a variety of operating systems including Mac OS X, Windows, and Linux [11]. An API is a set of classes, or types, and functions that can be useful when implementing a program. In the case of Python, the Tkinter API was designed to allow Python programs to work with the Tk package to implement GUI programs that will run on Windows, Mac OS X, or Linux [5]. The Tkinter API is included in a module called *tkinter*. The module is included with most distributions of Python and may be imported to use in your Python programs.

Tk programs use widgets to build a GUI application. The term widget has been used at least since the 1980s to refer to any element of a GUI application including windows, buttons, menus, text entry fields, frames, listboxes, etc. There are many different widgets available in tkinter. Typically, any element you can see (and some you can't see, like frames) in a GUI application is a widget. The next sections will introduce several widgets while building a *Reminder!* note application.

6.1 The Root Window

To begin using the Tk API you open a root window. Tk applications can have more than one open window, but the main window is called the root window. It is opened by calling a function called *Tk*().

Example 6.1 Here is code to open a Tk window.

```python
import sys
import tkinter

def main():
    root = tkinter.Tk()

    root.title("Reminder!")
    root.resizable(width=False,height=False)

    tkinter.mainloop()

if __name__ == "__main__":
    main()
```

The code in Example 6.1 opens a window as pictured in Fig. 6.1. The call to the *title* method sets the title of the window. The call to *resizable* makes the window a non-resizable window. The *Tkinter.mainloop*() calls the Tk event dispatch loop to process events from the windowing application. Even with a simple window like this, the call to mainloop is required because there are events that even a simple

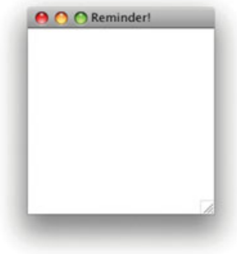


Fig. 6.1 A Tk root window

window must respond to. For example, when a window is moved on the screen it must respond to its redraw event. Redrawing the window is done automatically by the Tk code once the mainloop function is called.

Python 2 ⇝ 3

In Python 2 the module name for Tkinter was *Tkinter*. In Python 3 the module name become *tkinter*. If you are using Tkinter in Python 2.6 you write:

```
import Tkinter
```

to import the Tkinter module.

6.2 Menus

A menu can be added to the application by creating a Menu widget and adding it to the root window. On Windows and Linux the menu will appear right at the top of the window. On a Mac, the menu appears at the top of the screen on the menu bar. This menu contains a *File->Exit* menu item that quits the application when selected.

Example 6.2 Here is the code that, when added right before the call to mainloop, creates a File menu with one menu item to exit.

```
def quit():
    root.destroy()
bar = tkinter.Menu(root)
fileMenu = tkinter.Menu(bar,tearoff=0)
fileMenu.add_command(label="Exit",command=quit)
bar.add_cascade(label="File",menu=fileMenu)
root.config(menu=bar)
```

When adding a menu, you associate a command (i.e. a function) with each menu item added to the menu. The *Exit* menu item is associated with the *quit* function which calls the root's *destroy* method. Notice the *quit* function has no parameters. Most event handlers do not have parameters but do have access to the enclosing scope.

Practice 6.1 Write a Tkinter program that creates a main window with a menu that says *Help*. Within the *Help* menu item should be another menu item that says *About*. When the About menu is selected, your program should print "About was Selected" to the screen.

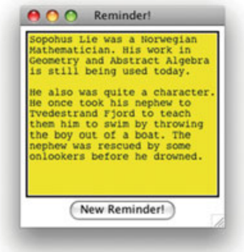
6.3 Frames

A Frame is an invisible widget that can be used as a container for other widgets. Frames are sometimes useful in laying out a GUI application. Layout refers to getting all the widgets in the right place and making them stay there even when the window is resized. We don't have to worry about resizing the window in the Reminder! application so layout will be a little easier.

In Fig. 6.2 there is a Frame widget. The frame is invisible. The text entry area is inside the frame and so is the *New Reminder!* button. Frames can be useful to group widgets together. They can also have a border around them. The border around this frame is 5 pixels wide. Adding the frame with a border gives a little edge to the window.

Example 6.3 This is the code that creates the frame for the Reminder! application.

```
mainFrame = tkinter.Frame(root,borderwidth=1,padx=5,pady=5)
mainFrame.pack()
```



```
Sopohus Lie was a Norwegian
Mathematician. His work in
Geometry and Abstract Algebra
is still being used today.

He also was quite a character.
He once took his nephew to
Tvedestrand Fjord to teach
them him to swim by throwing
the boy out of a boat. The
nephew was rescued by some
onlookers before he drowned.
```

Fig. 6.2 The main Reminder! window [9]

When the frame is created the first parameter to the Frame constructor is the window that the frame is to be packed into. This is true of every widget. The first parameter to the constructor when creating a widget is the widget it belongs to. In this way, widgets can be nested inside of widgets to form the GUI application. So, the *mainFrame* frame is a part of the *root* window. Recall that in Example 6.1 the variable *root* was set to the root Tk window.

Packing the mainFrame means to add it into the root window and make the contents of the frame visible. While a frame itself is invisible, by packing it the contents of the frame will be visible once the window is drawn. Packing is one method of making a widget visible. Other methods of making widgets visible are discussed in Sect. 6.9.

Practice 6.2 Create a frame and pack it in a root window.

6.4 The Text Widget

The Text widget is a powerful multi-line editing window that can embed graphics and other objects within it. In the Reminder! application it holds the message to be posted. The Text widget in this application is added to the *mainFrame*. By creating a Text widget and packing it into the main frame the user can enter text into it. The widget handles all the text entry itself without any intervention by the programmer.

Example 6.4 Here is the code to create a Text widget in the Reminder! application.

```
note = tkinter.Text(mainFrame,bg="yellow",width=30,height=15)
note.pack()
```

Practice 6.3 Create a text widget of 3 rows and 20 columns and place it in your practice GUI’s frame.

6.5 The Button Widget

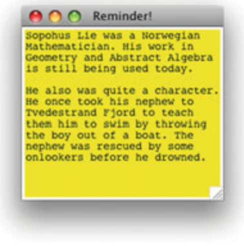
The Button widget is used to get button press input from a user. Buttons appear in the native button format of the operating system you are using so they may not look exactly like the button displayed in Fig. 6.2. Since a button must respond to being pressed, when you create a button you specify an event handler to handle the button presses. An event handler is added to the button in the same way a command was added to a menu item in Sect. 6.2.

Example 6.5 Here is the code to create a Button and its associated event handler.

```
def post():
    print("Post")
    addReminder(note.get("1.0",tkinter.END), \
       root.winfo_rootx()+5,root.winfo_rooty()+5, \
       notes,reminders)
    note.delete("1.0",tkinter.END)


tkinter.Button(mainFrame,text="New Reminder!", \
    command=post).pack()
```

Example 6.5 shows a button being created, being added to the main frame, and then being packed within the frame. The keyword argument *text* specifies the text to go on the button. The keyword *command* is used to specify a parameterless function to call when the button is pressed. The function *post* is a parameterless function and is defined in the same scope as the Button. Normally, a function is not defined within the scope of another function. However, in Tk programming it is much more common. Event handlers are almost always nested functions. By nesting the event handler in the main function, it has access to all the variables defined in the main function. In this example the *post* function needs to have access to the *root* variable as well as the *notes* and *reminders* variables. By defining *post* within the same scope as the *root* variable, the *post* function can use these values as needed. Since the function *post* cannot have any parameters as dictated by Tkinter API, the *post* function must access the *root* variable from the enclosing scope. To see the whole program in context refer to Chap. 15.

The *post* function gets the contents of the text field, called *note*, by using the *get* method on the note. Calling the *get* method with "1.0" and *tkinter.END* gets the text from beginning to end. The *winfo_rootx*() and *winfo_rooty*() methods get the x and y coordinates for the upper left corner of the root window. The post function then passes that information along with a couple of lists called *notes* and *reminders* to the *addReminder* function. The *addReminder* function adds a new reminder note to the screen as appears in Fig. 6.3.



```
Sopohus Lie was a Norwegian
Mathematician. His work in
Geometry and Abstract Algebra
is still being used today.

He also was quite a character.
He once took his nephew to
Tvedestrand Fjord to teach
them him to swim by throwing
the boy out of a boat. The
nephew was rescued by some
onlookers before he drowned.
```

Fig. 6.3 A Reminder!

Notice that when a command like *post* is provided to a button it is not written *post*(). This is because we are not calling post when the button is created. Instead, we are specifying that when the button is pressed the *post* function should be called. By providing the function name *post* to the button widget it can remember to call that function when it is pressed.

Practice 6.4 Create a button that says "Now!" on it. Connect it to a command that prints "Oh, now you've done it!" to the screen.

6.6 Creating a Reminder!

To create a Reminder! window another top level window is created. To do this, the button calls the *addReminder* function. There are two parts to a reminder, the window itself and the Text widget within the window. A list of reminder windows is maintained in a list called *notes*. A list of the text widgets is maintained in a list called *reminders*. These lists are *parallel* lists. This means that the first entry in both lists corresponds to the first reminder, the second element in both lists is the second reminder and so on. Parallel lists were first introduced in Sect. 4.9 on p. 103. Both the window and the Text widget are needed to maintain the information about a *reminder* in the program.

Example 6.6 Here is the code that adds reminders to the screen. The *notes* and *reminders* lists keep track of the windows and Text widgets.

```
    def addReminder(text,x,y,notes,reminders):
        notewin = tkinter.Toplevel()
        notewin.resizable(width=False,height=False)
        notewin.geometry("+"+str(x)+"+"+str(y))

        reminder = tkinter.Text(notewin,bg="yellow", \
           width=30,height=15)

        reminder.insert(tkinter.END,text)
        reminder.pack()

       notes.append(notewin)
       reminders.append(reminder)

       def deleteWindowHandler():
           print("Window Deleted")
           notewin.withdraw()
           notes.remove(notewin)
           reminders.remove(reminder)

       notewin.protocol("WM_DELETE_WINDOW", deleteWindowHandler)
```

To add a *reminder* to the screen a toplevel window is created, the new window is not resizable and is positioned over the top of the existing window using the *geometry* method. Calling geometry on a window with a string like "+10+10" positions the window at (10,10) pixels measured from the upper left corner of the screen. Since the root window's coordinates were passed to the function, the new window is positioned approximately on top of the root window.

The text is copied into the reminder. Then the window and the Text widget are copied into the notes and reminders lists, respectively. The last line of the method adds an event handler for the window deletion event. If the reminder window is closed, the user is getting rid of that reminder. In that case, the reminder window and corresponding Text widget are removed from the notes and reminders lists. The remove method looks for a matching element of the list and removes it. The only matching element of a window or Text entry widget is the original window or widget added to the list.

The *deleteWindowHandler* function is a case where accessing the enclosing scope is exactly what we want. We can't pass parameters to the *deleteWindowHandler* function, but we can access the notes, reminders, reminder, and notewin variables from the enclosing scope to remove the window from the program when it is closed.

6.7 Finishing up the Reminder! Application

There is only a little more code needed to finish the Reminder! application. It is more interesting if the reminders are saved to a file when the program is closed. Then the reminder windows can be redisplayed when the program is started again. The application saves the information in a file called *reminders.txt*. The file starts with the X,Y coordinate of the root window on the screen. Then, each reminder record starts with an X,Y coordinate of the reminder window followed by some text on multiple lines followed by a line of underscores and periods in a pattern that should never be seen by accident. The application reads from the file until this special line is found and then makes a reminder out of the text it just read. Then it continues reading the file looking for the next reminder.

Example 6.7 Here is the code that reads and writes the *reminders.txt* file.

```
    try:
        print("reading reminders.txt file")
        file = open("reminders.txt","r")
        x = int(file.readline())
        y = int(file.readline())
        root.geometry("+"+str(x)+"+"+str(y))

        line = file.readline()
        while line.strip() != "":
            x = int(line)
            y = int(file.readline())
            text = ""
```

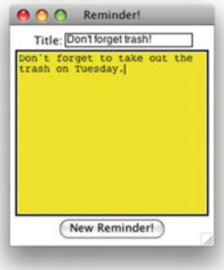
```
           line = file.readline()
           while line.strip() != "____...____._._._":
              text = text + line
              line = file.readline()

           text = text.strip()
           addReminder(text,x,y,notes,reminders)
           line = file.readline()
     except:
         print("reminders.txt not found")

     def appClosing():
         print("Application Closing")
         file = open("reminders.txt","w")
         file.write(str(root.winfo_x())+"\n")
         file.write(str(root.winfo_y())+"\n")

        for i in range(len(notes)):
           print(notes[i].winfo_rootx())
           print(notes[i].winfo_rooty())
           print(reminders[i].get("1.0",tkinter.END))
           file.write(str(notes[i].winfo_rootx())+"\n")
           file.write(str(notes[i].winfo_rooty())+"\n")
           file.write(reminders[i].get("1.0",tkinter.END)+"\n")
           file.write("____...____._._._\n")

        file.close()
        root.destroy()
        root.quit() # May or may not be necessary
        sys.exit()

    root.protocol("WM_DELETE_WINDOW", appClosing)
```

The code in the *try…except* block attempts to read the information when the application starts. This code is located in the main function of the application. When the window deletion event occurs for the main window, the appClosing handler is called. The appClosing function writes the file, overwriting any file that was read when the application started. The complete code for the Reminder! application can be found in Chap. 15.

6.8 Label and Entry Widgets

Assume we wish to enhance the Reminder! application by allowing the user to set the title of each reminder. Instead of the reminder note just having *Reminder!* as its title, it could have a user-defined title. So when the *New Reminder!* button was pressed for the application in Fig. 6.4 a new window would appear with “Don’t forget trash!” as its title. This can be done by adding a label and an entry widget to the application.

The Label widget is the text “Title:” that appears in the figure. The Entry widget is the one line text field. While a Text widget can handle multiple lines, an Entry widget holds just one line of text.



Don't forget trash!

Don't forget to take out the trash on Tuesday.

Fig. 6.4 A titled Reminder! application

Example 6.8 Here is the code for the Entry and Text widgets in this application.

```
titleFrame = tkinter.Frame(mainFrame)
titleFrame.pack()

noteTitle = tkinter.StringVar()
titleLabel = tkinter.Label(titleFrame,text="Title:")
titleLabel.grid(row=1,column=1,sticky=tkinter.E)
titleText = tkinter.Entry(titleFrame,textvariable=noteTitle)
titleText.grid(row=1,column=2,columnspan=2, \
     sticky=tkinter.E+tkinter.W)
```

A new frame is created because it will need to contain the two elements on one line in the application. Without a new frame, the "Title:" label would be packed above the Entry widget. Within the *titleFrame* frame, the *titleLabel* and *titleText* widgets are added using the *grid* layout instead of the *pack* layout. In a grid layout you specify which row and column of the grid the widget should be placed in. The *columnspan* argument specifies that the *titleText* widget should span 2 of the three columns of the row.

A *StringVar* is an object with a *get* and a *set* method. The *titleText* Entry widget is created specifying a textvariable called *noteTitle* which is required to be of type *StringVar*. To retrieve the text of the Entry widget we can write *noteTitle.get*() and to set the text of the widget we can write *noteTitle.set*(*"Whatever Text We Want"*). StringVars make it easy to set and retrieve text from an Entry widget.

There is a little more code to write to complete the extension of this application to include the title information in the reminders and in the text file that stores the reminders. This code is left as an exercise.

Practice 6.5 Add a label that says "What do you want?" to the practice Tk application from this chapter.

6.9 Layout Management

When widgets are packed or gridded in an application, their appearance within the application is called their layout. Sometimes, when widgets are placed within an application they appear in the right place when the application starts, but if the window is resized, they don't look right. Understanding something about layout management can help you correctly plan your application's layout and avoid these kinds of problems.

Packing widgets places them one above another in what is sometimes called a flow layout. Each widget appears above the next when packed. The Tk packer is responsible for packer layout management. There are some options that can affect how packing is done. Normally the packer places one widget above another in a flow layout. But these options let the programmer have some control about how that flow is managed.

- **fill =** You can specify that if a widget can use the extra space, then it should fill the available space. Valid values for fill are *tkinter.X*, *tkinter.Y*, or *tkinter.BOTH*. *X* means to fill in the horizontal direction, *Y* means to fill in the vertical direction, *BOTH* means to fill in both directions. For a label to fill in the horizontal direction you would write:

  ```
  titleLabel = tkinter.Label(titleFrame,text="Title:", \
      bg="green",fg="blue")
  titleLabel.pack(fill=tkinter.X)
  ```

  The *bg* and *fg* parameters set the background and foreground color, respectively.
- **side =** This specifies which side to flow from. For example, writing *titleLabel.pack*(*side=tkinter.LEFT*) will flow from the left rather than the top. Other valid values are *TOP*, *BOTTOM*, or *RIGHT*.

The Tk gridder is responsible for grid layout management. Grid layout allows widgets to be placed in a specific column and/or row of a container widget. As we have seen, it is possible for one widget to span more than one column or row in a grid. The *rowspan* parameter sets the number of rows a widget should span. The *columnspan* option was used in Example 6.8. It is also possible to tell the gridder how it should use the space within a row and column. Normally a widget is centered within the available space. But, if the widget can use it, the gridder can be told to expand the widget to take up the available space. The *sticky* option tells the gridder to stick the widget to one or more sides of the available area. The *tkinter.E* and *tkinter.W* constants stand for east and west. By adding east and west together in Example 6.8 the entry widget will expand to the full width of its allowable size. In that example it has no affect on the layout, since the window cannot be resized anyway, but nonetheless it demonstrates its use.

While packing and gridding are the two most common forms of layout management, there is also a placer. The placer places widgets explicitly within the X,Y plane of the application. The packer, gridder, and placer are the three layout managers for

Tkinter. Each of these layout managers have more options available for layout that are not discussed here but can be found by searching for "tkinter layout management" on the internet.

Practice 6.6 Make the entry widget and the button widget in your practice application appear next to each other at the bottom of the window.

6.10 Message Boxes

Sometimes it is necessary to pop up a message box in a GUI application to warn the user of some invalid operation they are trying to perform. Sometimes the application just needs to provide some quick feedback, like "Job Completed" or some other status. Tk provides a few message boxes for these occasions. To use the message boxes you must import *tkinter.messagebox*.
Here are three examples.

- **tkinter.messagebox.showinfo("Invalid Entry", "Type a reminder first.")**

  This displays an informational box with an informational icon. You can change the icon displayed in the box by specifying the *icon* = parameter. More information is available online. The dialog box appears on the screen and the application waits for *OK* to be pressed.
- **tkinter.messagebox.showwarning("Invalid Entry", "Type a reminder first.")**

  This works the same as the *showinfo* dialog box but displays a warning icon instead of an informational icon.
- **answer = tkinter.messagebox.askyesno("Really?", "Are you sure you want to create a blank reminder?")**

  This displays a dialog with *Yes* and *No* buttons. If *Yes* is pressed, the function call returns *True*. If *No* is pressed, the function returns *False*.

Python 2 $\rightsquigarrow$ 3

In Python 2 the module name for message boxes was *tkMessageBox*. In Python 3 the module name became *tkinter.messagebox*. If you are using Tkinter in Python 2.6 you write:

```
import tkMessageBox
```

to import the Tkinter message box module.

There are other dialogs available including a color chooser and file chooser. There are also several other options that are possible with each of these dialogs. Again, more information can be found online.

Practice 6.7 When the button of your practice application is pressed, take the information in the entry widget and display it in a message box of your choice with some appropriate text to go with it.

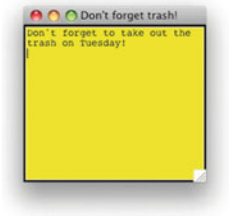
6.11 Review Questions

1. How are a event-driven program and simple sequential program the same?
2. What distinguishes an event-driven program from a sequential program?
3. What is an API?
4. Name two APIs that are available in Python. What does each API do for you as a programmer?
5. What is a widget?
6. When writing a Tkinter application, what is the purpose of the call to *mainloop*?
7. What is the purpose of a frame in Tkinter?
8. What does the term *layout* refer to in a GUI application? Be complete in your answer.
9. What is the purpose of the StringVar class in Tkinter applications?
10. Why are event handlers generally defined within the scope of the main function?
11. What are two methods of arranging widgets in a Tkinter application? Describe the differences between the two methods.

6.12 Exercises

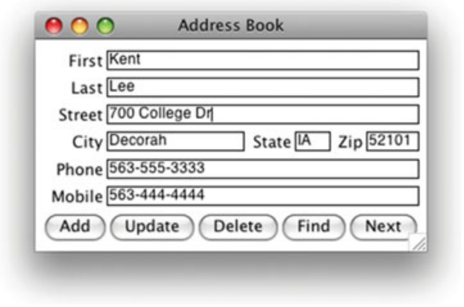
1. Extend the Reminder! application so that each Reminder! is given the title assigned in the main application window. For example, if the *New Reminder!* button is pressed for the application as it appears in Fig. 6.4, the reminder window would appear as shown in Fig. 6.5. Be sure to clear both the text and the title from the root application window after the *New Reminder!* button is pressed.
2. Implement a GUI front-end to the address book application. The GUI should be similar to that presented in Fig. 6.6. Each of the buttons in the application should work as described here.

   (a) The add button should add a new entry to the phonebook. This must append an entry to the phonebook. The event handler for this function should look something like this (depending on how you write the rest of your program).



Don't forget to take out the
trash on Tuesday!

Fig. 6.5 A titled Reminder!



First Kent
Last Lee
Street 700 College Dr
City Decorah State IA Zip 52101
Phone 563-555-3333
Mobile 563-444-4444

Fig. 6.6 A GUI for the addressbook application

```
def addAddress():
    print "Add"

    if lname.get().strip() == "":
        tkMessageBox.showwarning("Missing Last Name", \
            "You must enter a non-empty last name.")
        return

    if fname.get().strip() == "":
        tkMessageBox.showwarning("Missing First Name", \
            "You must enter a non-empty first name.")
        return

    file = open("addressbook.txt","a")

    file.write(lname.get().strip()+"\n")
    file.write(fname.get().strip()+"\n")
    file.write(street.get().strip()+"\n")
    file.write(city.get().strip()+","+state.get().strip()+\
        " "+zip.get().strip()+"\n")
    file.write(phone.get().strip()+"\n")
    file.write(mobile.get().strip()+"\n")

    file.close()

    tkMessageBox.showinfo("Entry Added", \
        "The entry was successfully added.")
```

(b) The update button should update an existing entry or display a message saying the entry was not found. Update must find an entry that matches the

first and last name displayed in the GUI. If found, the entry in the file is updated to reflect the new information found in the GUI. You find an entry by matching the first and last name in the address book so updating the name will not work. In that case a new entry needs to be added and the old one deleted. If the entry is not found a warning message should be displayed.
Since entries cannot be deleted from files, to update an entry you must open a new file for writing. Then you copy all the entries to the new file that don't match the entry to be updated. Once you find the entry to be updated you write the GUI information to the new file. Finally, you must write the rest of the non-matching entries to the new file. After you are done, you can remove the old file and rename the new file to the *addressbook.txt* file name.
The following lines of code will delete the addressbook.txt file and rename a file called *__newbook.txt* to *addressbook.txt*.

```
os.remove("addressbook.txt")
os.rename(".__newbook.txt","addressbook.txt")
```

(c) The delete button deletes an existing entry. To delete an existing entry the last and first name should match the entry being deleted. Since you cannot delete a record from a file, you must create a new file, writing all records to the new file except for the one to be deleted. Then remove the old file and rename the new file to *addressbook.txt*. See the description of the *update* button implementation to see how to delete and rename the files.
(d) The find button finds the entry with the same first and last name as typed. It should at least work when both last and first name are supplied by the user. However, you can extend this by making it work if the last name is empty. Then it should match only on first name. Likewise, if the first name is empty then it should only match on last name. In either case it should display the first matching entry in the address book.
(e) The next button displays the next address after the current entry and wraps around to the beginning when the last entry was displayed.

3. Implement a GUI front-end for the addressbook application as described in Exercise 2, but use parallel lists to hold the fields of each record instead of reading from and writing to the file immediately. You should write code to read the entire file when the application starts and it should be written again when the application closes.
Each of the buttons should be implemented but instead of reading or writing to the file, the buttons should use the parallel lists as the source of the addressbook entries.
4. Using the Reminder! application code from Appendix 15 as a reference, rewrite the code so that the reminders are read from an XML file when the application starts and are written to an XML file when the application terminates. To write an XML file you open a text file for writing and you write the data and the XML tags for each XML element.

5. Implement a GUI front-end for the addressbook application but in this version of the application define an XML file format to hold the data. Then, write the program to read the XML file when the application starts and write the XML file when the application terminates. Use parallel lists to hold the fields of each record while the application is running. To write an XML file you open a text file for writing and you write the data and the XML tags for each XML element.

6.13 Solutions to Practice Problems

These are solutions to the practice problems in this chapter. You should only consult these answers after you have tried each of them for yourself first. Practice problems are meant to help reinforce the material you have just read so make use of them.

6.13.1 Solutions to Practice Problem 6.1

```
import tkinter

def main():
    def about():
        print("About was Selected")

    root = tkinter.Tk()

    root.title("Silly Program")

    bar = tkinter.Menu(root)

    fileMenu = tkinter.Menu(bar,tearoff=0)
    fileMenu.add_command(label="About",command=about)
    bar.add_cascade(label="Help",menu=fileMenu)
    root.config(menu=bar)

if __name__ == "__main__":
    main()
    tkinter.mainloop()
```

6.13.2 Solutions to Practice Problem 6.2

The window will probably resize to a very tiny window when run because there isn't anything in the frame yet.

```
mainFrame = tkinter.Frame(root,borderwidth=1,padx=5,pady=5)
mainFrame.pack()
```

6.13.3 Solutions to Practice Problem 6.3

```
note = tkinter.Text(mainFrame, width=20,height=3)
note.pack()
```

6.13.4 Solutions to Practice Problem 6.4

```
def pressedIt():
    print("Oh, now you've done it!")

tkinter.Button(mainFrame,text="Now!", \
    command=pressedIt).pack()
```

6.13.5 Solutions to Practice Problem 6.5

```
titleLabel = tkinter.Label(mainFrame, \
    text="What do you want?")
titleLabel.pack()
```

6.13.6 Solutions to Practice Problem 6.6

```
bottomFrame = tkinter.Frame(root,borderwidth=1, \
    padx=5,pady=5)
bottomFrame.pack()

titleLabel = tkinter.Label(bottomFrame, \
    text="What do you want?")
titleLabel.grid(column=1,row=1)

tkinter.Button(bottomFrame,text="Now!", \
    command=pressedIt).grid(column=2,row=1)
```

6.13.7 Solutions to Practice Problem 6.7

```
import tkinter.messagebox

    def pressedIt():
        print("Oh, now you've done it!")
        tkinter.messagebox.showinfo("Okey dokey", \
            "Well let me get"+note.get("1.0",tkinter.END)+ \
            "for you!")
```