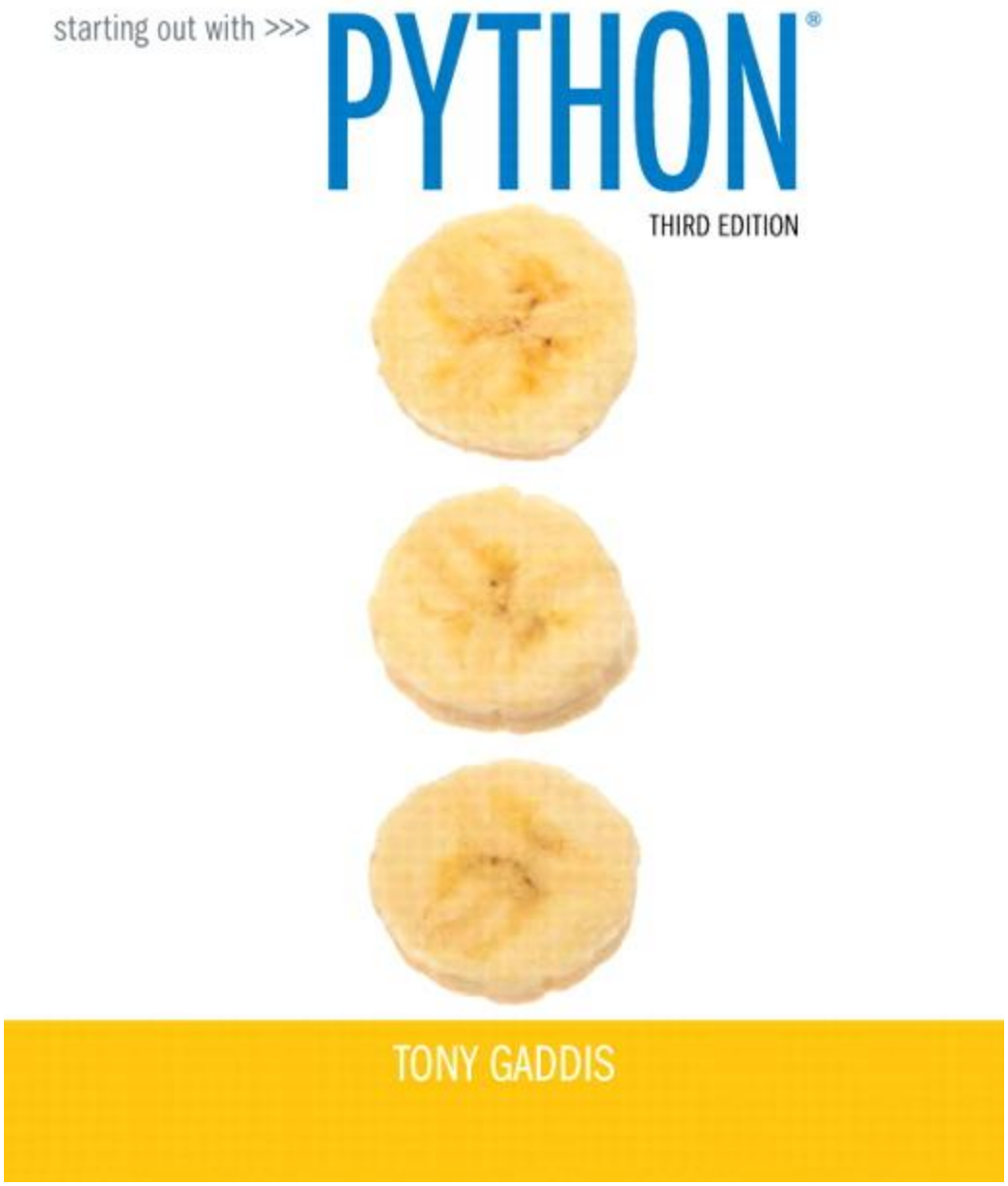
CHAPTER 13

GUI Programmin g



Topics

- Graphical User Interfaces
- Using the `tkinter` Module
- Display Text with `Label` Widgets
- Organizing Widgets with Frames
- `Button` Widgets and Info Dialog Boxes
- Getting Input with the `Entry` Widget
- Using Labels as Output Fields
- Radio Buttons and Check Buttons

Graphical User Interfaces

- **User Interface**: the part of the computer with which the user interacts
- **Command line interface**: displays a prompt and the user types a command that is then executed
- **Graphical User Interface (GUI)**: allows users to interact with a program through graphical elements on the screen

Graphical User Interfaces

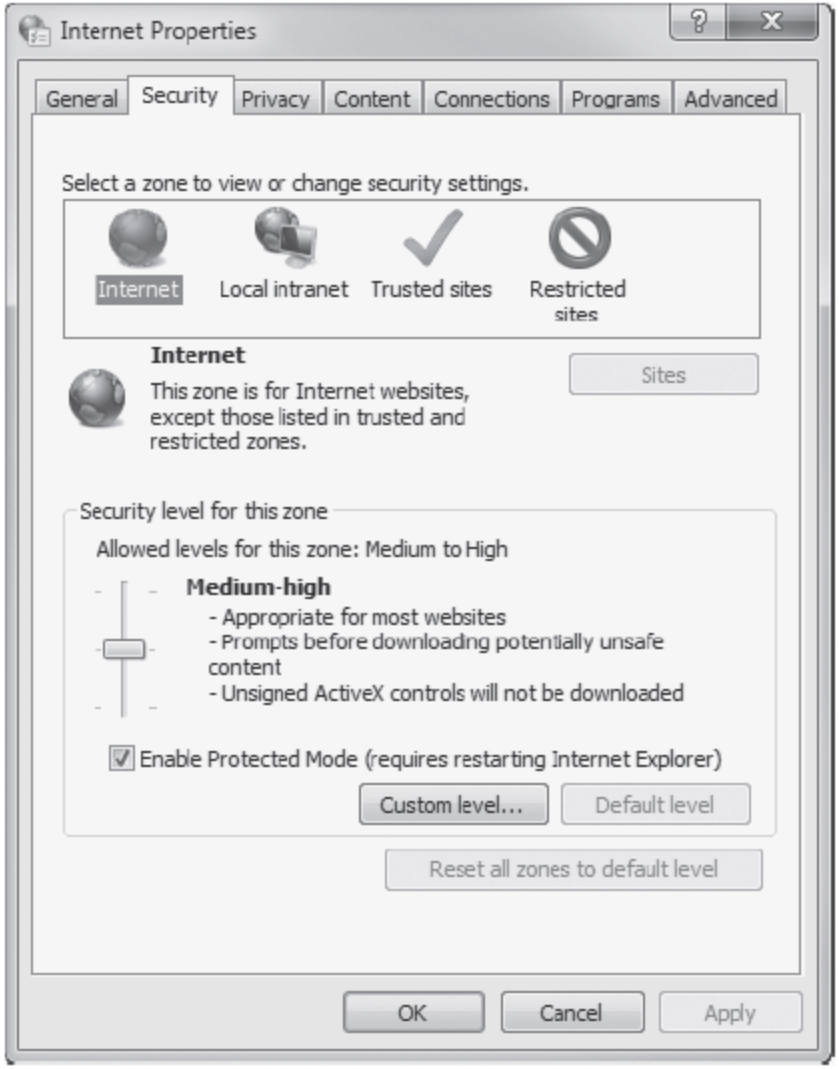
Figure 13-1 A command line interface

```
C:\MyPrograms>dir
 Volume in drive C has no label.
 Volume Serial Number is 2414-0000

 Directory of C:\MyPrograms

01/18/2008  08:10 AM    <DIR>          .
01/18/2008  08:10 AM    <DIR>          ..
04/17/2007  03:23 PM               250 payroll.py
               1 File(s)            250 bytes
               2 Dir(s)  21,691,060,224 bytes free

C:\MyPrograms>
```

Graphical User Interfaces

- **<u>Dialog boxes</u>: small windows that display information and allow the user to perform actions**
  - Responsible for most of the interaction through GUI
  - User interacts with graphical elements such as icons, buttons, and slider bars

Figure 13-2 A dialog box

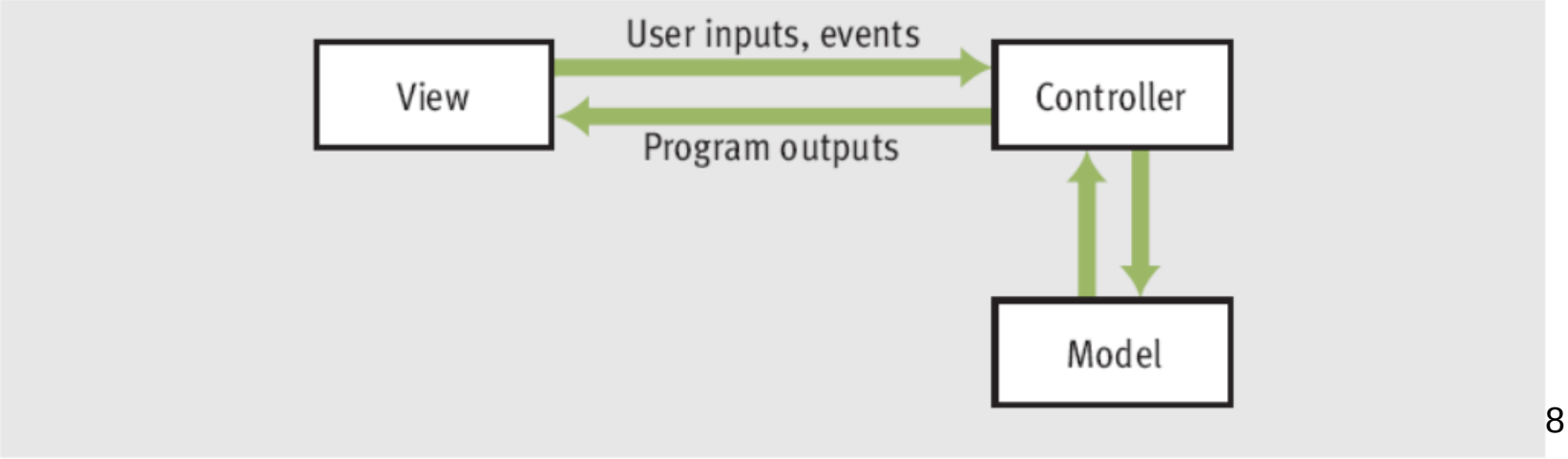


GUI Programs Are Event-Driven

- **In text-based environments, programs determine the order in which things happen**
  - The user can only enter data in the order requested by the program
- **GUI environment is event-driven**
  - The user determines the order in which things happen
    - User causes events to take place and the program responds to the events

GUI Programs Are Event-Driven

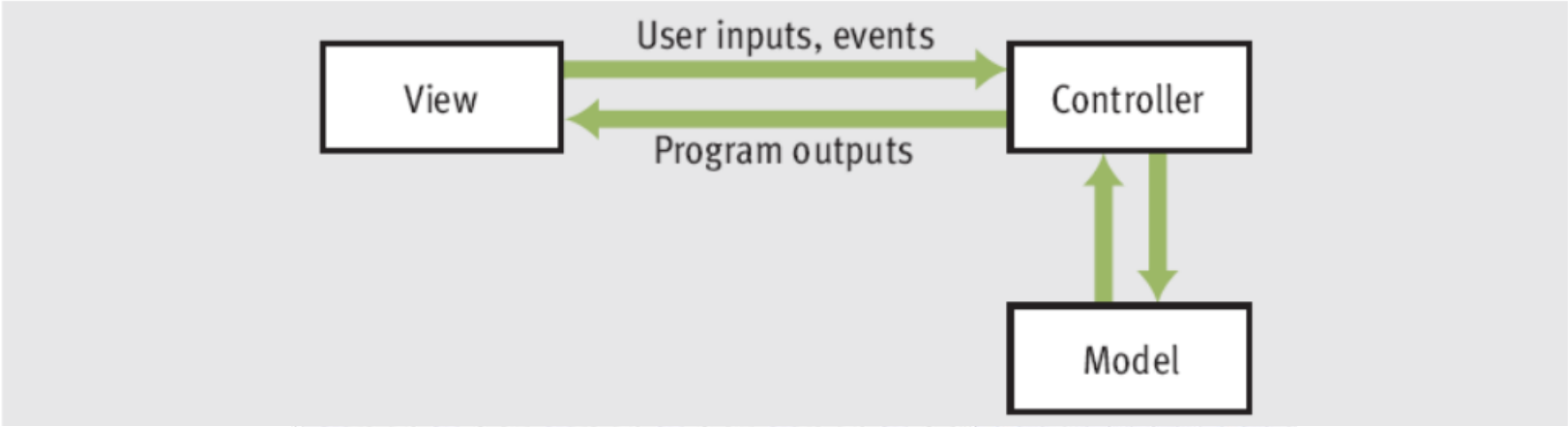
- User-generated events (mouse clicks, button clicks) trigger operations in program to respond by pulling in inputs, processing them, and displaying results
  - Event-driven software
  - Event-driven programming

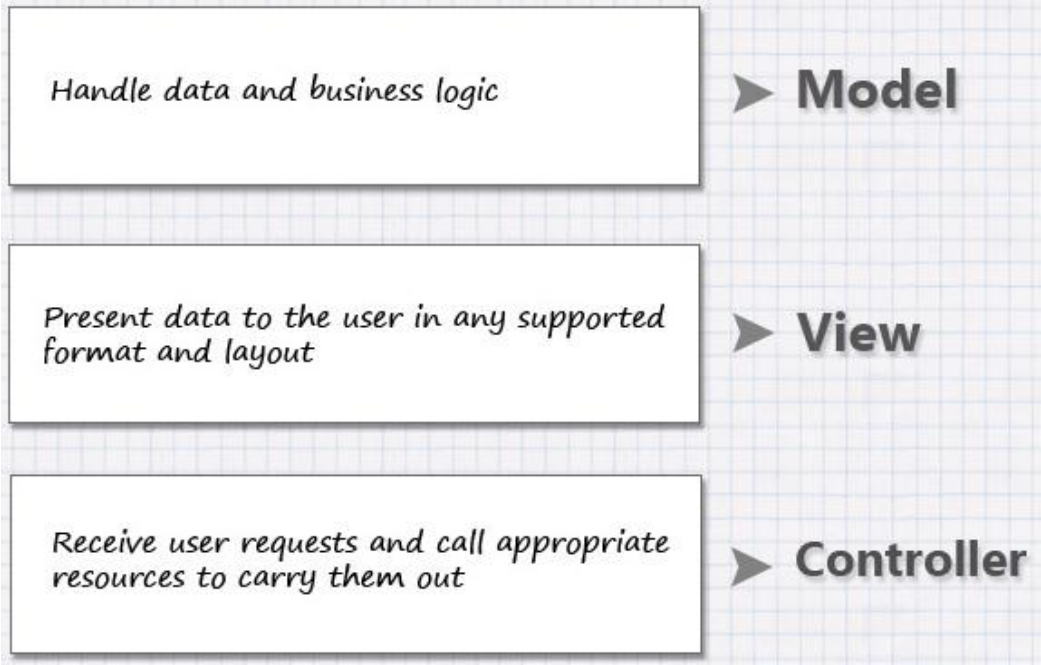


Model-View-Controller (MVC)

- Pattern used in creating graphic user interfaces (guis)
  - Separate three different aspects of the GUI:
    - The data (model)
    - The visual representation of the data (view)
    - The interface between the view and the model (controller)
  - Primary idea:
    - Keep MVC components separate
    - Each one is as independent of the others
    - Changes made to one will not affect changes made to the others
    - For example: GUI can be updated with a new look or visual style without having to change the data model or the controller

Model-View-Controller (MVC)



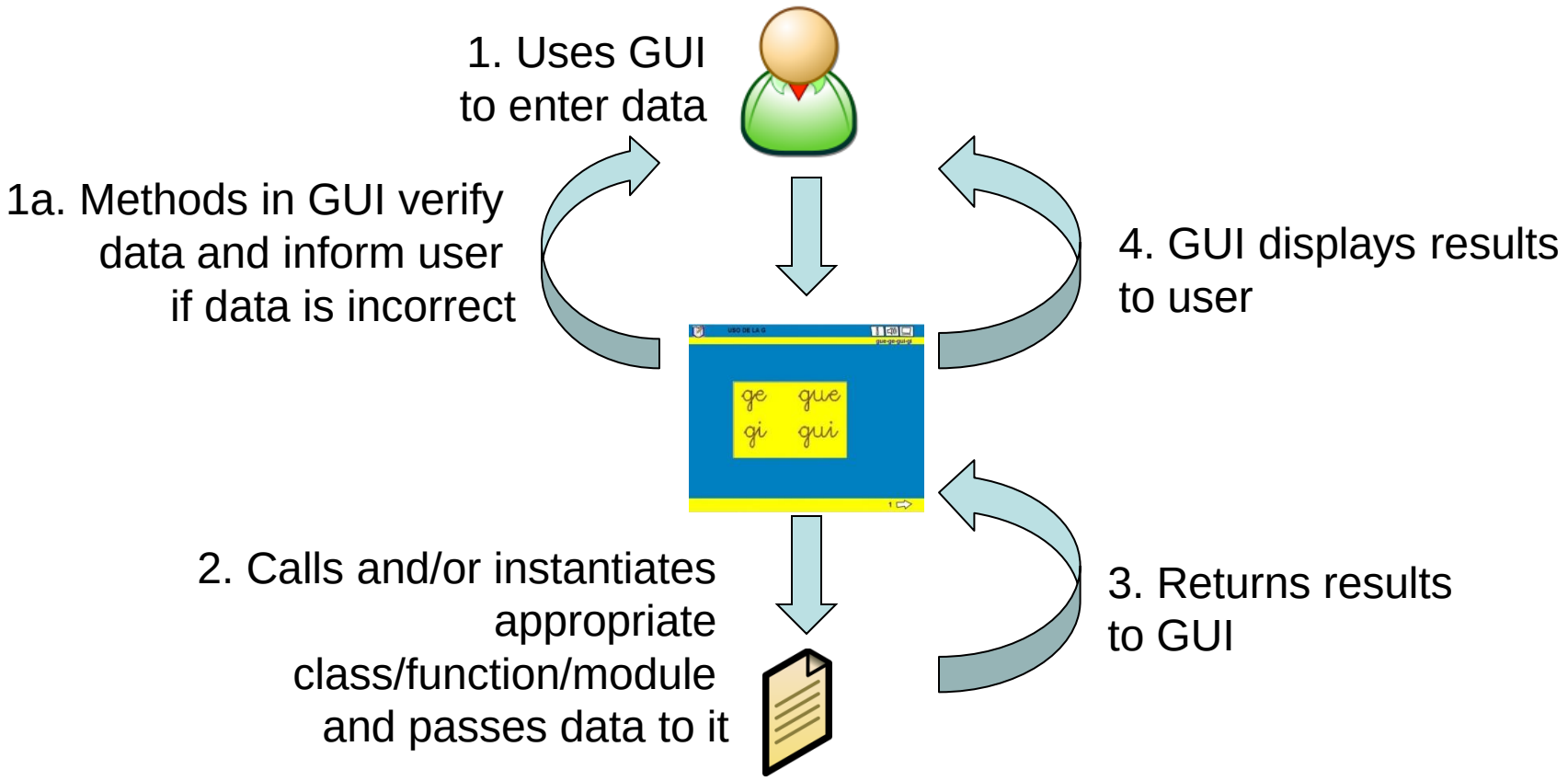


“Simplified” Model-View-Controller (MVC)

- **Simplified MVC for our class**
  1. GUI receives input from user
  2. GUI will pass that input to appropriate module/function/class to be processed
     - There shall be no data processing in GUI functions or classes
     - It is OK to have input validation in GUI
  3. GUI receives processed information from module/function/class and displays it to user
     - There shall be no input validation in this step

“Simplified” Model-View-Controller (MVC)

- **Simplified MVC for our class**



GUI Programs Are Event-Driven

- **Coding phase:**
  - Define a new class to represent the main window
  - Instantiate the classes of window objects needed for this application (e.g., labels, command buttons)
  - Position these components in the window
  - Instantiate the data model and provide for the display of any default data in the window objects
  - Register controller methods with each window object in which a relevant event might occur
  - Define these controller methods
  - Define a **`main`** that launches the GUI

GUI-Based Programs

- **There are many libraries and toolkits of GUI components available to the Python programmer**
  - **`tkinter`** includes classes for windows and numerous types of window objects
  - **`tkinter.messagebox`** includes functions for several standard pop-up dialog boxes

Using the `tkinter` Module

- **No GUI programming features built into Python**
- **`tkinter` module: allows you to create simple GUI programs**
  - Comes with Python
- **Widget: graphical element that the user can interact with or view**
  - Presented by a GUI program

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.

Using the `tkinter` Module

- **Programs that use `tkinter` do not always run reliably under IDLE**
  - 99.99999% they will run just fine
- **Programmers take an object-oriented approach when writing GUI programs**
  - `__init__` method builds the GUI
  - When an instance is created the GUI appears on the screen

EmptyWindow.py

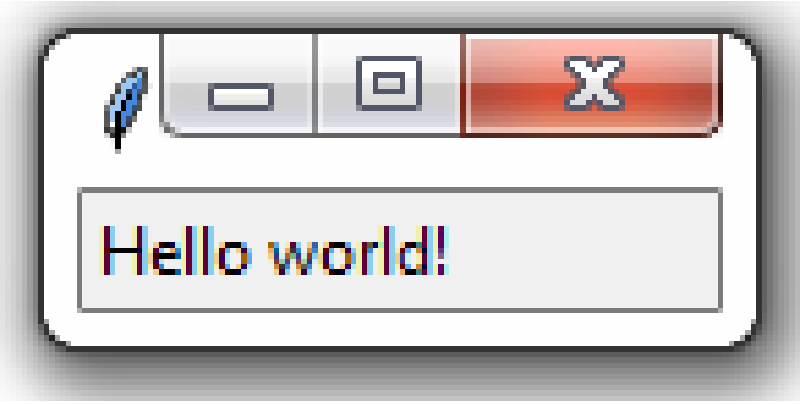
First GUI

- **A grid layout allows programmer to place components in the cells of an invisible grid**

```
from tkinter import *

class LabelDemo(Frame):

    def __init__(self):
        """Sets up the window and widgets."""
        Frame.__init__(self)
        self.master.title("Label Demo")
        self.grid()
        self._label = Label(self, text = "Hello world!")
        self._label.grid()

def main():
    """Instantiate and pop up the window."""
    LabelDemo().mainloop()

main()
```

First GUI

- **The GUI is launched in the main method**
  - Instantiates LabelDemo and calls mainloop
- **mainloop method pops up window and waits for user events**
  - At this point, the main method quits (GUI is running a hidden, event-driven loop in a separate process)



Display Text with `Label` Widgets

- **`Label` widget: displays a single line of text in a window**
  - Made by creating an instance of `tkinter` module's `Label` class
  - Format:

```
tkinter.Label(self.main_window, \
              text = "my text")
```

    - First argument references the root widget, second argument shows text that should appear in label

MyGui.py

Display Text with `Label` Widgets

- **`pack` method: determines where a widget should be positioned and makes it visible when the main window is displayed**
  - Called for each widget in a window
  - Receives an argument to specify positioning
    - Positioning depends on the order in which widgets were added to the main window
    - Valid arguments: `side="top", side="left", side="right"`

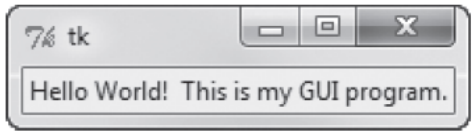
MyGui2.py

Display Text with `Label` Widgets

Figure 13-5 Window displayed by Program 13-3

Figure 13-6 Window displayed by Program 13-4

Figure 13-7 Window displayed by Program 13-5



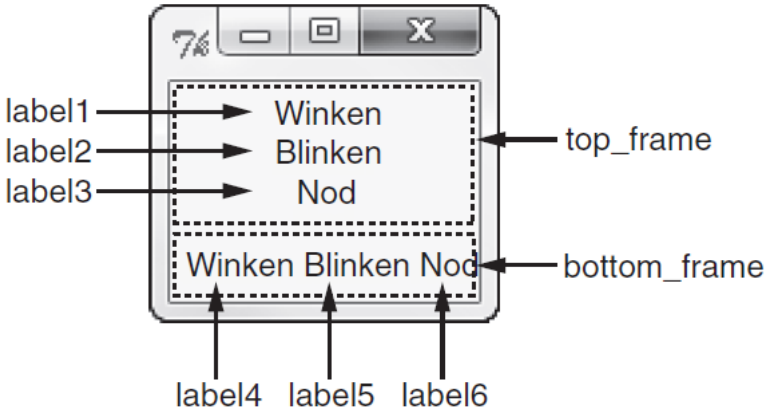
Organizing Widgets with Frames

- **`Frame` widget: container that holds other widgets**
  - Useful for organizing and arranging groups of widgets in a window
  - The contained widgets are added to the frame widget which contains them
    - Example:

```
tkinter.Label(self.top_frame, \
              text = 'hi')
```

Organizing Widgets with Frames (cont'd.)

Figure 13-9 Arrangement of widgets



`Button` Widgets and Info Dialog Boxes

- **<u>`Button` widget</u>: widget that the user can click to cause an action to take place**
  - When creating a button can specify:
    - Text to appear on the face of the button
    - A callback function
- **<u>Callback function</u>: function or method that executes when the user clicks the button**
  - Also known as an event handler

`Button` Widgets and Info Dialog Boxes

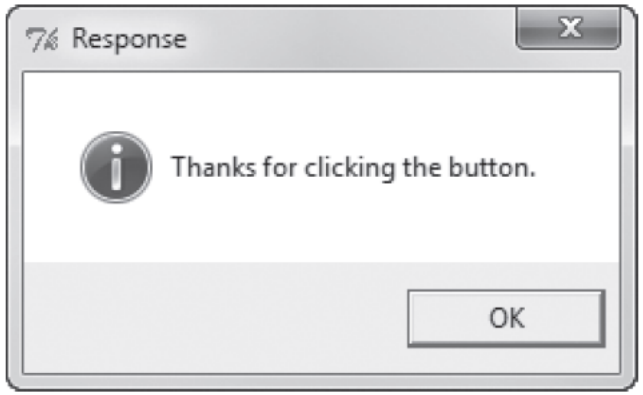
- **<u>Info dialog box</u>: a dialog box that shows information to the user**
  - Format for creating an info dialog box:
    - Import `tkinter.messagebox` module
    - `tkinter.messagebox.showinfo(`*`title`*`, \`
      `message``)`
      - *`title`* is displayed in dialog box's title bar
      - *`message`* is an informational string displayed in the main part of the dialog box

ButtonDemo2.py

`Button` Widgets and Info Dialog Boxes (cont'd.)

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

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



TestAverages.py

Getting Input with the `Entry` Widget

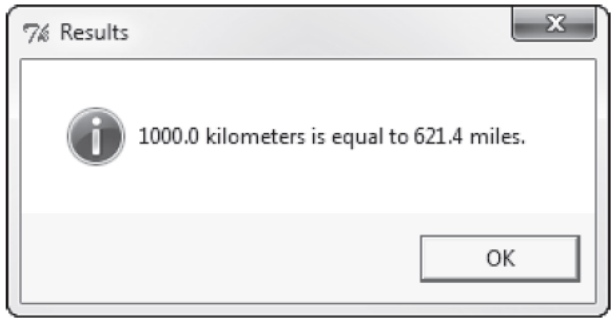
- **`Entry` widget: rectangular area that the user can type text into**
  - Used to gather input in a GUI program
  - Typically followed by a button for submitting the data
    - The button's callback function retrieves the data from the `Entry` widgets and processes it
  - `Entry` widget's `get` method: used to retrieve the data from an `Entry` widget
    - Returns a string

Getting Input with the `Entry` Widget

Figure 13-15 The info dialog box

1 The user enters 1000 into the Entry widget and clicks the Convert button.

2 This info dialog box is displayed.



KilometersToMiles.py

Using Labels as Output Fields

- **Can use `Label` widgets to dynamically display output**
  - Used to replace info dialog box
  - Create empty `Label` widget in main window, and write code that displays desired data in the label when a button is clicked

Using Labels as Output Fields

- **`StringVar` class: `tkinter` module class that can be used along with `Label` widget to display data**
  - Create `StringVar` object and then create `Label` widget and associate it with the `StringVar` object
  - Subsequently, any value stored in the `StringVar` object with automatically be displayed in the `Label` widget

Using Labels as Output Fields

Figure 13-16 The window initially displayed

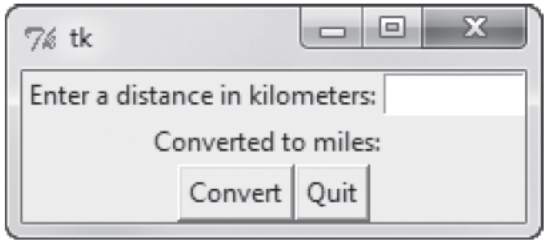
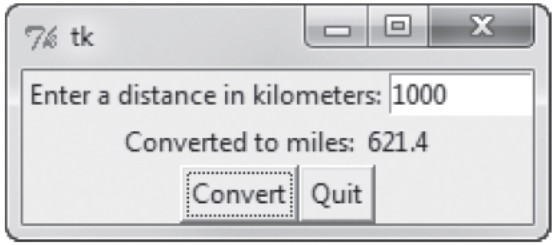


Figure 13-17 The window showing 1000 kilometers converted to miles



Displaying Images

- **Steps to create a label with an image:**
  - __init__ creates an instance of PhotoImage from a GIF file on disk
  - The label's image attribute is set to this object

```
from tkinter import *

class ImageDemo(Frame):

    def __init__(self):
        """Sets up the window and widgets."""
        Frame.__init__(self)
        self.master.title("Image Demo")
        self.grid()
        self._image = PhotoImage(file = "smokey.gif")
        self._imageLabel = Label(self, image = self._image)
        self._imageLabel.grid()
        self._textLabel = Label(self, text = "Smokey the cat")
        self._textLabel.grid()
```

Displaying Images

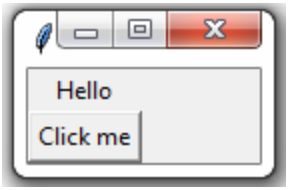
- The image label is placed in the grid before the text label
- The resulting labels are centered in a column in the window

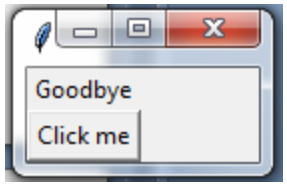
Command Buttons and Responding to Events

- **A button can display either text or an image**
- **To activate a button and enable it to respond to clicks, set command to an event-handling method**
  - In this case, _switch examines the text attribute of the label and sets it to the appropriate value
  - Attributes are stored in a dictionary

Command Buttons and Responding to Events

```
from tkinter import *

class ButtonDemo(Frame):

    def __init__(self):
        """Sets up the window and widgets."""
        Frame.__init__(self)
        self.master.title("Button Demo")
        self.grid()
        self._label = Label(self, text = "Hello")
        self._label.grid()
        self._button = Button(self,
                              text = "Click me",
                              command = self._switch)
        self._button.grid()

    def _switch(self):
        """Event handler for the button."""
        if self._label["text"] == "Hello":
            self._label["text"] = "Goodbye"
        else:
            self._label["text"] = "Hello"
```





Entry Fields for the Input and Output of Text

- A form filler consists of labeled entry fields, which allow the user to enter and edit a single line of text
- A field can also contain text output by a program
- tkinter's Entry displays an entry field
- Three types of data container objects can be used with Entry fields:

TYPE OF DATA	TYPE OF DATA CONTAINER
float	DoubleVar
int	IntVar
str (string)	StringVar

Creating a Quit Button

- **<u>Quit button</u>: closes the program when the user clicks it**
- **To create a quit button in Python:**
  - Create a `Button` widget
  - Set the root widget's `destroy` method as the callback function
    - When the user clicks the button the `destroy` method is called and the program ends

QuitButtonDemo.py

Radio Buttons and Check Buttons

- **<u>Radio button</u>: small circle that appears filled when it is selected and appears empty when it is deselected**
  - Useful when you want the user to select one choice from several possible options
- **<u>`Radiobutton` widgets</u>: created using `tkinter` module's `Radiobutton` class**
  - `Radiobutton` widgets are mutually exclusive
    - Only one radio button in a container may be selected at any given time

Radio Buttons and Check Buttons

- **`IntVar` class: a `tkinter` module class that can be used along with `Radiobutton` widgets**
  - Steps for use:
    - Associate group of `Radiobutton` widgets with the same `IntVar` object
    - Assign unique integer to each `Radiobutton`
    - When a `Radiobutton` widgets is selected, its unique integer is stored in the `IntVar` object
  - Can be used to select a default radio button

Using Callback Functions with `Radiobuttons`

- **You can specify a callback function with `Radiobutton` widgets**
  - Provide an argument `command=self.my_method` when creating the `Radiobutton` widget
  - The command will execute immediately when the radio button is selected
  - Replaces the need for a user to click OK or submit before determining which `Radiobutton` is selected

RadioButtonDemo.py

Check Buttons

- **<u>Check button</u>: small box with a label appearing next to it; check mark indicates when it is selected**
  - User is allowed to select any or all of the check buttons that are displayed in a group
    - Not mutually exclusive
- **<u>`Checkbutton` widgets</u>: created using `tkinter` module's `Checkbutton` class**
  - Associate different `IntVar` object with each `Checkbutton` widget

CheckButtonDemo.py

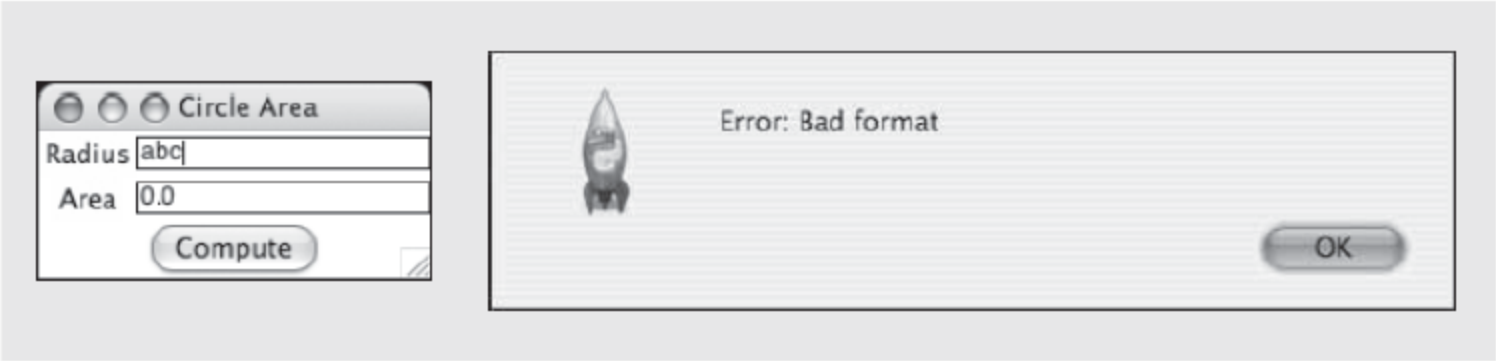
Using Pop-up Dialog Boxes

tkinter.messagebox FUNCTION	WHAT IT DOES
`askokcancel(title = None, message = None, parent = None)`	Asks an OK/Cancel question, returns **True** if **OK** is selected, **False** otherwise.
`askyesno(title = None, message = None, parent = None)`	Asks a Yes/No question, returns **True** if **Yes** is selected, **False** otherwise.
`showerror(title = None, message = None, parent = None)`	Shows an error message.
`showinfo(title = None, message = None, parent = None)`	Shows information.
`showwarning(title = None, message = None, parent = None)`	Shows a warning message.

[TABLE 9.2] Some **tkinter.messagebox** functions

Using Pop-up Dialog Boxes

```
def _area(self):
    """Event handler for the button."""
    try:
        radius = self._radiusVar.get()
        area = radius ** 2 * math.pi
        self._areaVar.set(area)
    except ValueError:
        tkinter.messagebox.showerror(message = "Error: Bad format",
                                     parent = self)
```



Colors

- **tkinter module supports the RGB**
  - Values expressed in hex notation (e.g., #ff0000)
  - Some commonly used colors have been defined as string values (e.g., "white", "black", "red")
- **For most components, you can set two color attributes:**
  - A foreground color (fg) and a background color (bg)

```
self._exampleLabel = Label(self, text = "Example",
                           fg = "red", bg = "#cccccc")
```

```
Frame.__init__(self, bg = "blue")
```

Text Attributes

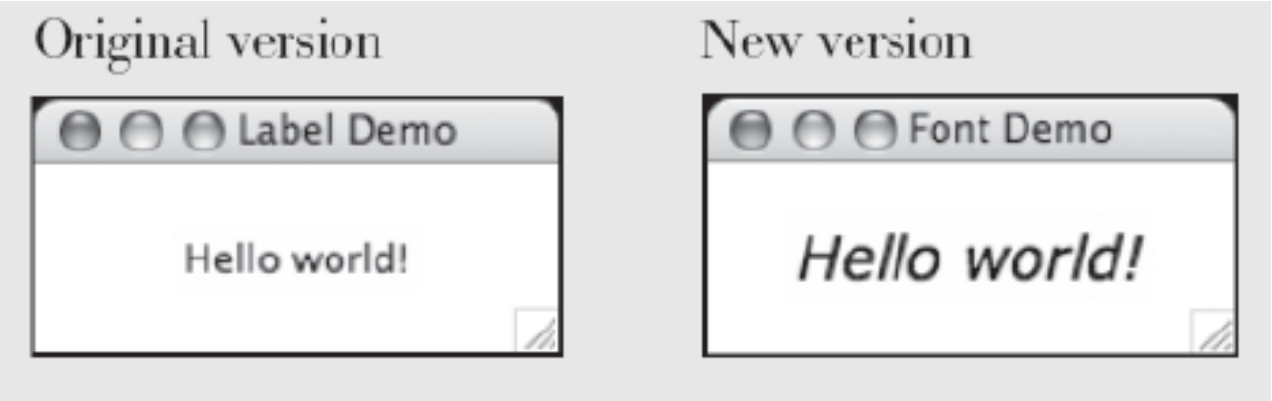
- The text displayed in a label, entry field, or button can also have a type font

tkinter.font ATTRIBUTE	VALUES
family	A string, as included in the tuple returned by tkinter.font.families().
size	An integer specifying the point size.
weight	"bold" or "normal".
slant	"italic" or "roman".
underline	1 or 0.

Text Attributes

- Example:

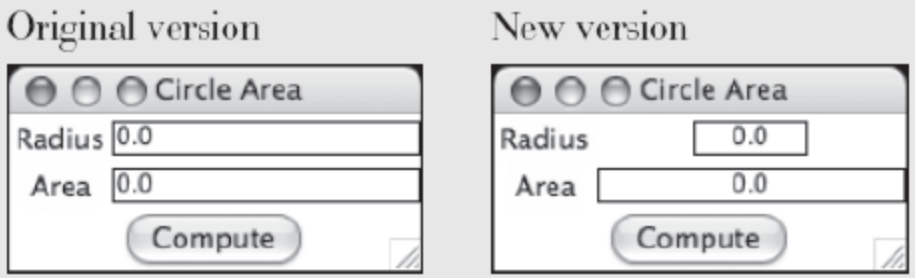
```
font = tkinter.font.Font(family = "Verdana",
                         size = 20, slant = "italic")
self._label = Label(self, font = font, text = "Hello world!")
```



Sizing and Justifying an Entry

- It's common to restrict the data in a given entry field to a fixed length; for example:
- – A nine-digit number for a Social Security number

```
self._radiusEntry = Entry(self, justify = "center", width = 7,
                          textvariable = self._radiusVar)
self._areaEntry = Entry(self, justify = "center",
                        textvariable = self._areaVar)
```



Sizing the Main Window

- To set the window's title:

```
self.master.title(<a string>)
```

- Two other methods, `geometry` and `resizable`, can be run with the root window to affect its sizing

```
self.master.geometry("200x100")
self.master.resizable(0, 0)
```

- Generally, it is easiest for both the programmer and the user to manage a window that is *not* resizable
  - Some flexibility might occasionally be warranted

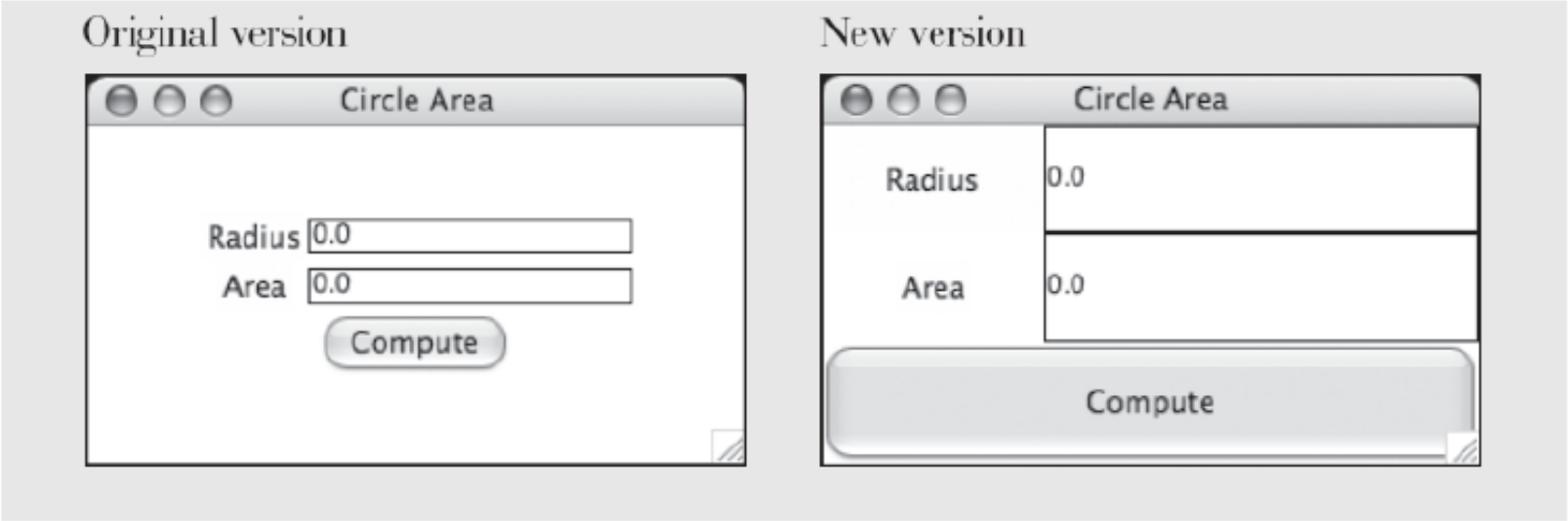
Grid Attributes

- By default, a newly opened window shrink-wraps around its components and is resizable
  - When window is resized, the components stay shrink-wrapped in their grid
    - Grid remains centered within the window
    - Widgets are also centered within their grid cells
- Occasionally,
  - A widget must be aligned to left/right of its grid cell,
  - Grid must expand with surrounding window, and/or
  - Components must expand within their cells

Grid Attribute

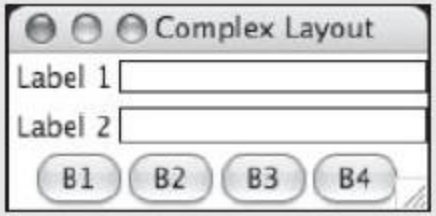
```
self.master.rowconfigure(0, weight = 1)
self.master.columnconfigure(0, weight = 1)
self.grid(sticky = W+E+N+S)
```

```
for row in range(3):
    self.rowconfigure(row, weight = 1)
for column in range(2):
    self.columnconfigure(column, weight = 1)
```



Using Nested Frames to Organize Components

- Suppose a GUI requires a row of command buttons beneath two columns of labels and entry fields:



- It is difficult, but not impossible, to create this complex layout with a single grid
- Alternative: decompose window into two nested frames (panes), each containing its own grid

Using Nested Frames to Organize Components

- The new frame is then added to its parent's grid and becomes the parent of the widgets in its own grid

```
class ComplexLayout(Frame):

    def __init__(self):

        # Create the main frame
        Frame.__init__(self)
        self.master.title("Complex Layout")
        self.grid()

        # Create the nested frame for the data pane
        self._dataPane = Frame(self)
        self._dataPane.grid(row = 0, column = 0)
```

Multi-Line Text Widgets

- Use `Text` widget to display multiple lines of text
  - `wrap` attribute → `CHAR` (default), `WORD`, or `NONE`
  - Widget can expand with its cell; alternative: scroll bars
- Text within a `Text` widget is accessed by index positions (specified as strings):

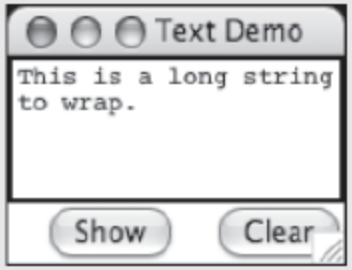
  "*rowNumber*.*characterNumber*"
- `insert` is used to send a string to a `Text` widget:

```
output.insert("1.0", "Pythonƒrules!")
output.insert(END, "Pythonƒrules! ")
```
- `delete` can be used to clear a `Text` widget:

```
output.delete("1.0",  END)
```

Multi-Line Text Widgets

```
def _show(self):
    self._outputArea.insert("1.0", self._text)

def _clear(self):
    self._outputArea.delete("1.0", END)
```



Scrolling List Boxes

Listbox METHOD	WHAT IT DOES
`box.activate(index)`	Selects the string at **index**, counting from 0.
`box.curselection()`	Returns a tuple containing the currently selected index, if there is one, or the empty tuple.
`box.delete(index)`	Removes the string at **index**.
`box.get(index)`	Returns the string at **index**.
`box.insert(index, string)`	Inserts the string at index, shifting the remaining lines down by one position.
`box.see(index)`	Adjust the position of the list box so the string at **index** is visible.
`box.size()`	Returns the number of strings in the list box.
`box.xview()`	Used with a horizontal scroll bar to effect scrolling.
`box.yview()`	Used with a vertical scroll bar to effect scrolling.

Scrolling List Boxes

```
self._theList.insert(END, "Apple")
self._theList.insert(END, "Banana")
self._theList.insert(END, "Cherry")
self._theList.insert(END, "Orange")
self._theList.activate(0)
```

```
self.rowconfigure(0, weight = 1)
self._listPane.rowconfigure(0, weight = 1)
```

```
def _add(self):
    """If an input is present, insert it at the
    end of the items in the list box and scroll to it."""
    item = self._inputVar.get()
    if item != "":
        self._theList.insert(END, item)
        self._theList.see(END)
```

```
def _remove(self):
    """If there are items in the list, remove
    the selected item."""
    if self._theList.size() > 0:
        self._theList.delete(ACTIVE)
```

Mouse Events

TYPE OF MOUSE EVENT	DESCRIPTION
<ButtonPress-*n*>	Mouse button *n* has been pressed while the mouse cursor is over the widget; *n* can be 1 (left button), 2 (middle button), or 3 (right button).
<ButtonRelease-*n*>	Mouse button *n* has been released while the mouse cursor is over the widget; *n* can be 1 (left button), 2 (middle button), or 3 (right button).
<B*n*-Motion>	The mouse is moved with button *n* held down.
<*Prefix*-Button-*n*>	The mouse has been clicked over the widget; *Prefix* can be **Double** or **Triple**.
<Enter>	The mouse cursor has entered the widget.
<Leave>	The mouse cursor has left the widget.

Mouse Event

- Associate a mouse event and an event-handling method with a widget by calling the `bind` method:

```
self._theList.bind("<ButtonRelease-1>", self._get)
```

- Now all you have to do is define the `_get` method
  - Method has a single parameter named `event`

```
def _get(self, event):
    """If the list is not empty, copy the selected
    string to the entry field."""
    if self._theList.size() > 0:
        index = self._theList.curselection()[0]
        self._inputVar.set(self._theList.get(index))
```

Keyboard Events

- GUI-based programs can also respond to various keyboard events:

TYPE OF KEYBOARD EVENT	DESCRIPTION
<KeyPress>	Any key has been pressed.
<KeyRelease>	Any key has been released.
<KeyPress-*key*>	***key*** has been pressed.
<KeyRelease-*key*>	***key*** has been released.

- Example: to bind the key press event to a handler

```
self._radiusEntry.bind("<KeyPress-Return>",
                       lambda event: self._area())
```

Summary

- **This chapter covered:**
  - Graphical user interfaces and their role as event-driven programs
  - The `tkinter` module, including:
    - Creating a GUI window
    - Adding widgets to a GUI window
    - Organizing widgets in frames
    - Receiving input and providing output using widgets
    - Creating buttons, check buttons, and radio buttons