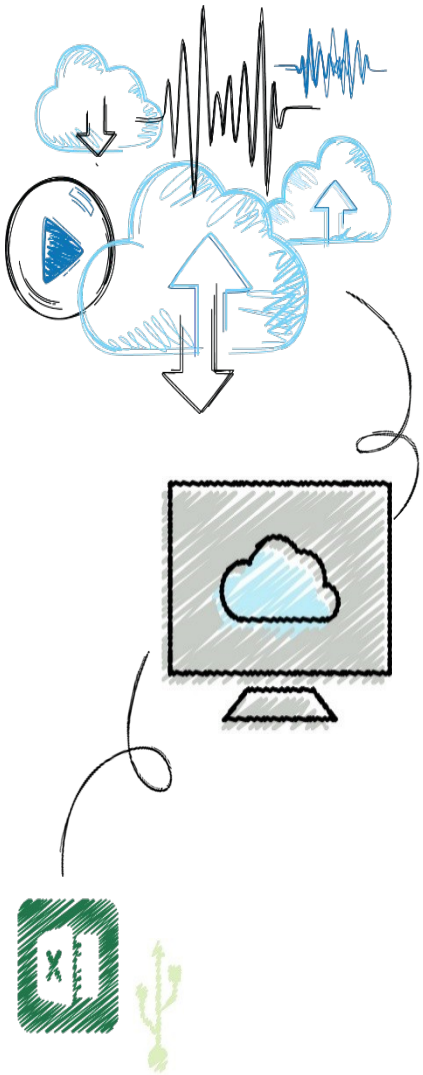
Fundamentals of Python: First Programs Second Edition

Chapter 8

Graphical User Interfaces



Objectives

8.1 Design and code a GUI-based program

8.2 Define a new class using subclassing and inheritance

8.3 Instantiate and lay out different types of window objects, such as labels, entry fields, and command buttons, in a window's frame

8.4 Define methods that handle events associated with window objects

8.5 Organize sets of window objects in nested frames

Introduction

- Most modern computer software employs a **graphical user interface** or **GUI**
- A GUI displays text as well as small images (called icons) that represent objects such as directories, files of different types, command buttons, and drop-down menus
- In addition to entering text at keyboard, the user of a GUI can select an icon with pointing device, such as mouse, and move that icon around on the display

The Behavior of Terminal-Based Programs and GUI-Based Programs

- Two different versions of the same program from a user's point of view:
  - Terminal-based user interface
  - Graphical user interface
- Both programs perform exactly the same function
  - However, their behavior, or look and feel, from a user's perspective are quite different

Terminal-Based Version

- Terminal-based user interface has several effects on its users:
  - User is constrained to reply to a definite sequence of prompts for inputs
    - Once an input is entered, there is no way to change it
  - To obtain results for a different set of input data, user must wait for command menu to be displayed again
    - At that point, the same command and all of the other inputs must be re-entered

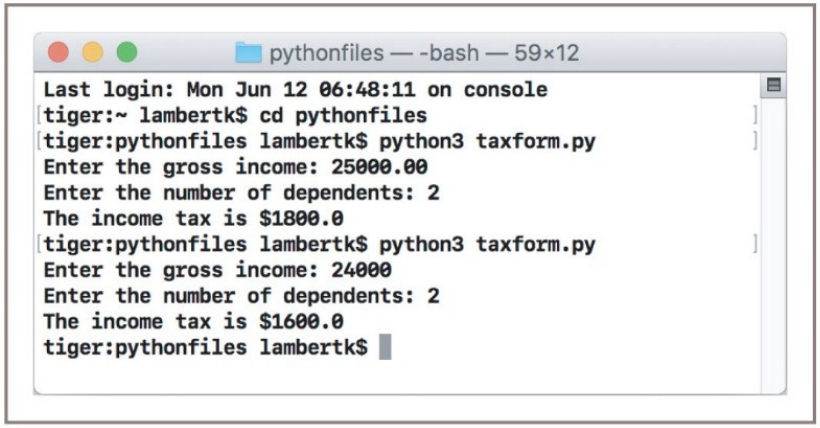


Figure 8-1 A session with the terminal-based tax calculator program

The GUI-Based Version (1 of 2)

- GUI-based version displays a window that contains various components
  - Called **window objects** or **widgets**

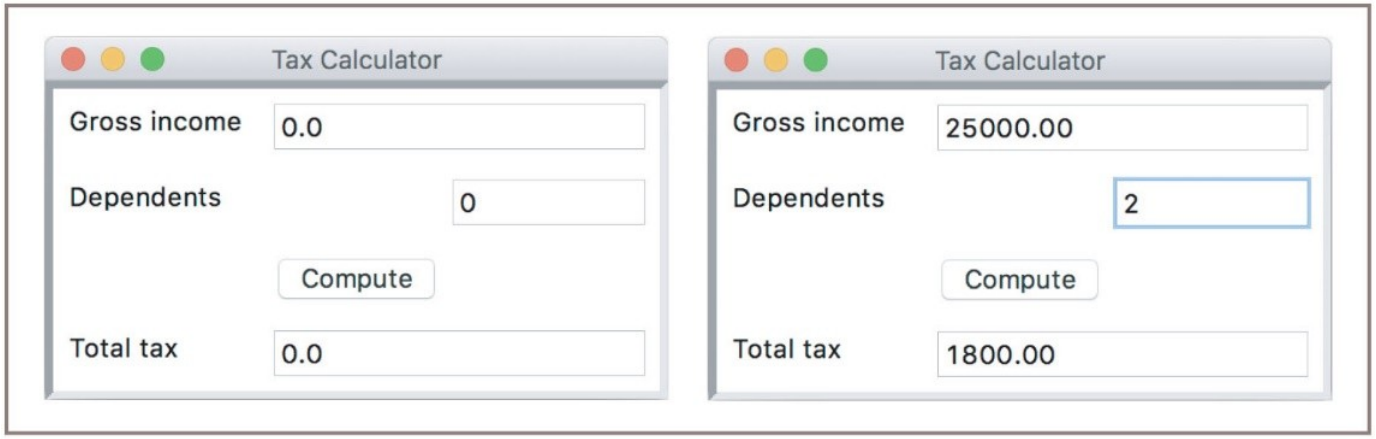


Figure 8-2 A GUI-based tax calculator program

The GUI-Based Version (2 of 2)

- GUI-based version has the following effects on users:
  - User is not constrained to enter inputs in a particular order
    - Before pressing the Compute button, user can edit any of the data
  - Running different data sets does not require re-entering all of the data
- GUI seems to be a definite improvement on the terminal-based user interface

Event-Driven Programming

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

Coding Simple 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
  - **breezypythongui** a custom, open-source module

A Simple "Hello World" Program

- A new window class extends the EasyFrame class
- The EasyFrame class provides the basic functionality for any window

```
"""
File: labeldemo.py
"""
from breezypythongui import EasyFrame
class LabelDemo(EasyFrame):
    """Displays a greeting in a window."""
    def __init__(self):
        """Sets up the window and the label."""
        EasyFrame.__init__(self)
        self.addLabel(text = "Hello world!", row = 0, column = 0)
def main():
    """Instantiates and pops up the window."""
    LabelDemo().mainloop()
if __name__ == "__main__":
    main()
```

Figure 8-3 Displaying a label with text in a window

A Template for All GUI Programs

- The structure of a GUI program is always the same, so there is a template:

```
from breezypythongui import EasyFrame
Other imports
class ApplicationName(EasyFrame):
        The __init__ method definition
        Definitions of event handling methods
def main():
        ApplicationName().mainloop()
if __name__ == "__main__":
        main()
```

The Syntax of Class and Method Definitions

- Each definition has a one-line header that begins with a keyword (class or def)
  - Followed by a body of code indented one level in the text
- A class header contains the name of the class followed by a parenthesized list of one or more parent classes
- The body, nested one tab under the header, consists of one or more method definitions
- A method header looks like a function header
  - But a method always has at least one parameter named **self**

Subclassing and Inheritance as Abstraction Mechanisms

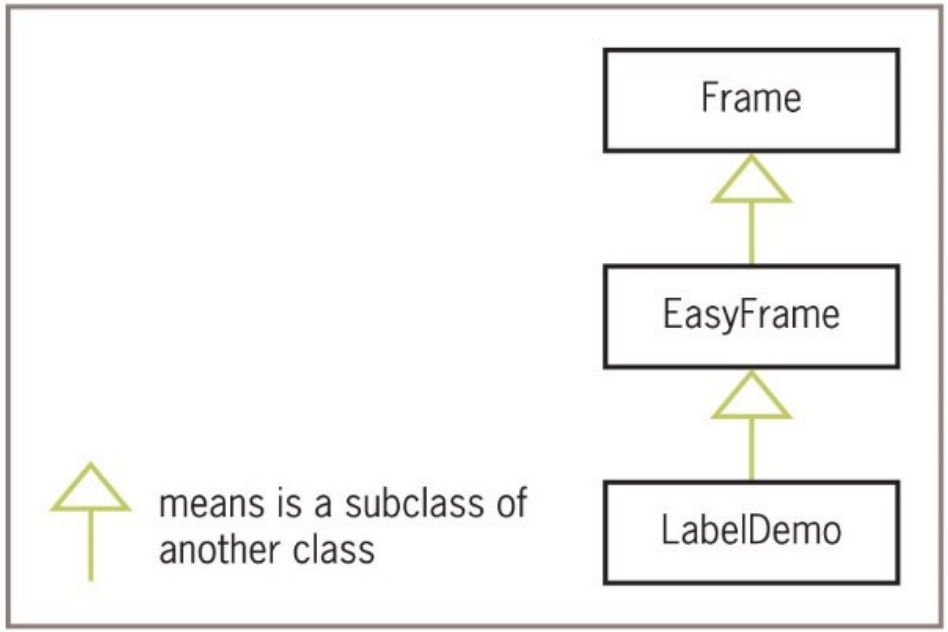


Figure 8-4 A class diagram for the label demo program

Windows and Window Components

- This section explores the details of windows and window components
- You will also learn how to:
  - Choose appropriate classes of GUI objects
  - Access and modify their attributes
  - Organize them to cooperate to perform the task at hand

CENGAGE

Windows and Their Attributes (1 of 2)

- Most important attributes:
  - Title (an empty string by default)
  - Width and height in pixels
  - Resizability (true by default)
  - Background color (white by default)
- Example of overriding dimensions and title:

```
EasyFrame.__init__(self, width = 300, height = 200,
                   title = "Label Demo")
```

- See Table 8-1 for other methods to change a window's attributes

Windows and Their Attributes (2 of 2)

EasyFrame Method	What It Does
setBackground(color)	Sets the window's background color to **color**
setResizable(aBoolean)	Makes the window resizable (**True**) or not (**False**)
setSize(width, height)	Sets the window's width and height in pixels
setTitle(title)	Sets the window's title to **title**

Window Layout (1 of 4)

- Window components are laid out in the window's two-dimensional **grid**
  - Rows and columns are numbered from the position (0,0) in the upper left corner of the window
- Example:

```
class LayoutDemo(EasyFrame):
    """Displays labels in the quadrants."""

    def __init__(self):
     """Sets up the window and the labels."""
      EasyFrame.__init__(self)
      self.addLabel(text = "(0, 0)", row = 0, column = 0)
      self.addLabel(text = "(0, 1)", row = 0, column = 1)
      self.addLabel(text = "(1, 0)", row = 1, column = 0)
      self.addLabel(text = "(1, 1)", row = 1, column = 1)
```

Window Layout (2 of 4)

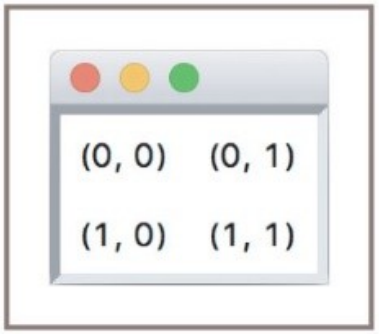


Figure 8-5 Laying out labels in the window's grid

Window Layout (3 of 4)

- Each type of window component has a default alignment
- Programmers can override the default alignment by including the **sticky** attribute as a keyword argument:

```
self.addLabel(text = “(0, 0)”, row = 0, column = 0,
              sticky = “NSEW”)
self.addLabel(text = “(0, 1)", row = 0, column = 1,
              sticky = "NSEW”)
self.addLabel(text = “(1, 0)", row = 1, column = 0,
              sticky = "NSEW”)
self.addLabel(text = “(1, 1)”, row = 1, column = 1,
              sticky = “NSEW”)
```

Window Layout (4 of 4)

- An aspect of window layout involves the spanning of a window component across several grid positions
- The programmer can force a horizontal and/or vertical spanning of grid positions by supplying the **rowspan** and **columnspan** keyword arguments

```
self.addLabel(text = "(0, 0)", row = 0, column = 0,
              sticky = "NSEW")
self.addLabel(text = "(0, 1)", row = 0, column = 1,
              sticky = "NSEW")
self.addLabel(text = "(1, 0 and 1)", row = 1, column = 0,
sticky = "NSEW", columnspan = 2)
```

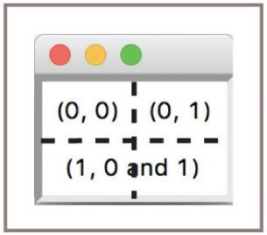


Figure 8-6 Labels with center alignment and a column span of 2

Types of Window Components and Their Attributes

- breezypythongui includes methods for adding each type of window component to a window
- Each method uses the form:

  self.addComponentType(<arguments>)

- When this method is called, breezypythongui
  - Creates an instance of the requested type of window component
  - Initializes the component's attributes with default values or any values provided by the programmer
  - Places the component in its grid position (the row and column are required arguments)
  - Returns a reference to the component

Displaying Images (1 of 3)

- The image label is first added to the window with an empty string
  - Program then creates a **PhotoImage** object from an image file and sets the **image** attribute of the image label to this object
  - The program creates a Font object with a non-standard font and resets the text label's font and foreground attributes to obtain the caption shown in Figure 8-7
- Code:

```
from breezypythongui import EasyFrame
from tkinter import PhotoImage
from tkinter.font import Font
class ImageDemo(EasyFrame):
        """Displays an image and a caption."""
```

Displaying Images (2 of 3)

- Code (continued):

```
def __init__(self):
        """Sets up the window and the widgets."""
        EasyFrame.__init__(self, title = "Image Demo")
        self.setResizable(False);
        imageLabel = self.addLabel(text = " ",
                                   row = 0, column = 0,
                                   sticky = "NSEW")
        textLabel = self.addLabel(text = "Smokey the cat",
                                   row = 1, column = 0,
                                   sticky = "NSEW")

        # Load the image and associate it with the image label.
        self.image = PhotoImage(file = "smokey.gif")
        imageLabel["image"] = self.image

        # Set the font and color of the caption.
        font = Font(family = "Verdana", size = 20, slant = "italic")
        textLabel["font"] = font
        textLabel["foreground"] = "blue"
```

Displaying Images (3 of 3)

Label Attribute	Type of Value
image	A **PhotoImage** object (imported from **tkinter.font**)
text	A string
background	A color
foreground	A color (color of text)
font	A **Font** object (imported from **tkinter.font**)

Command Buttons and Responding to Events (1 of 2)

- A command button is added to a window just like a label
  - By specifying its text and position in the grid
- A button is centered in its grid position by default
- The method **addButton** accomplishes all this and returns an object of type **tkinter.Buttton**
- Figure 8-8 shows these two states of the window, followed by the code for the initial version of the program

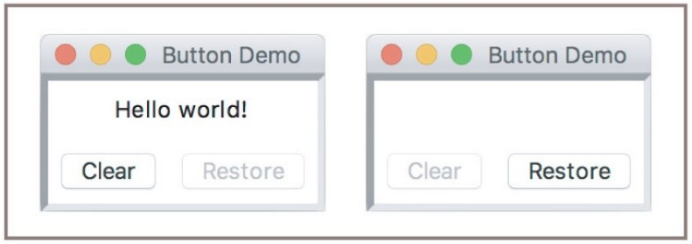


Figure 8-8 Using command buttons

Command Buttons and Responding to Events (2 of 2)

```
class ButtonDemo(EasyFrame):
    """Illustrates command buttons and user events."""

    def __init__(self):
      """Sets up the window, label, and buttons."""
      EasyFrame.__init__(self)

      # A single label in the first row.
      self.label = self.addLabel(text = "Hello world!",
                                 row = 0, column = 0,
                                 columnspan = 2,
                                 sticky = "NSEW")
     # Two command buttons in the second row.
     self.clearBtn = self.addButton(text = "Clear",
                                 row = 1, column = 0)
     self.restoreBtn = self.addButton(text = "Restore",
                                 row = 1, column = 1,
                                 state = "disabled")
```

Input and Output with Entry Fields

- Entry field
  - A box in which the user can position the mouse cursor and enter a number or a single line of text
- This section explores the use of entry fields to allow a GUI program to take input text or numbers from a user
  - And display text or numbers as input

Text Fields (1 of 3)

- Text field
  - Appropriate for entering or displaying a single-line string of characters
- Programmers use the method **addTextField** to add a text field to a window
  - The method returns an object of type **TextField**, which is subclass of **tkinter.Entry**
- Required arguments to **addTextField** are:
  - **Text**, **row**, and **column**
  - Optional arguments are **rowspan**, **columnspan**, **sticky**, **width**, and **state**

Text Fields (2 of 3)

- Code:

```
class TextFieldDemo(EasyFrame):
        """ Converts an input string to uppercase and displays
        the result. """
        def __init__(self):
         """ Sets up the window and widgets."""
           EasyFrame.__init__(self, title = "Text Field Demo")
           # Label and field for the input
           self.addLabel(text = "Input", row = 0, column = 0)
           self.inputField = self.addTextField(text = " ",
                                               row = 0,
                                               column = 1)
           # Label and field for the output
           self.addLabel(text = "Output", row = 1, column = 0)
           self.outputField = self.addTextField(text = " ",
                                               row = 1,
                                               column = 1,
                                               state = "readonly")
```

Text Fields (3 of 3)

- Code (continued):

```
# The command button
        self.addButton(text = "Convert", row = 2, column = 0,
                columnspan = 2, command = self.convert)

    # The event handling method for the button
    def convert(self):
        """ Inputs the string, converts it to uppercase,
        and outputs the result."""
        text = self.inputField.getText()
        result = text.upper()
        self.outputField.setText(result)
```

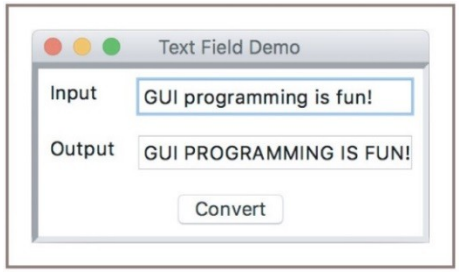


Figure 8-9 Using text fields for input and output

Integer and Float Fields for Numeric Data (1 of 4)

- **breezypythongui** includes two types of data fields for the input and output of integers and floating-point numbers:
  - **IntegerField** and **FloatField**
- Similar in usage to the method **addTextField**
  - However, instead of an initial **text** attribute, the programmer supplies a **value** attribute
- The method **addFloatField** allows an optional precision argument
- The methods **getNumber** and **setNumber** are used for the input and output of numbers with integer and float fields

Integer and Float Fields for Numeric Data (2 of 4)

- Code:

```
class NumberFieldDemo(EasyFrame):
    """ Computes and displays the square root of an
    input number."""

    def __init__(self):
        """Sets up the window and widgets."""
        EasyFrame.__init__(self, title = "Number Field Demo")
        # Label and field for the input
        self.addLabel(text = "An integer",
                      row = 0, column = 0)
        self.inputField = self.addIntegerField(value = 0,
                                               row = 0,
                                               column = 1,
                                               width = 10)
        # Label and field for the output
        self.addLabel(text = "Square root",
                      row = 1, column = 0)
```

Integer and Float Fields for Numeric Data (3 of 4)

- Code (continued):

```
    self.outputField = self.addFloatField(value = 0.0,
                                           row = 1,
                                           column = 1,
                                           width = 8,
                                           precision = 2,
                                           state = "readonly")
    # The command button
    self.addButton(text = "Compute", row = 2, column = 0,
                   columnspan = 2,
                   command = self.computeSqrt)

# The event handling method for the button
def computeSqrt(self):
    """Inputs the integer, computes the square root,
    and outputs the result."""
    number = self.inputField.getNumber()
    result = math.sqrt(number)
    self.outputField.setNumber(result)
```

Integer and Float Fields for Numeric Data (4 of 4)

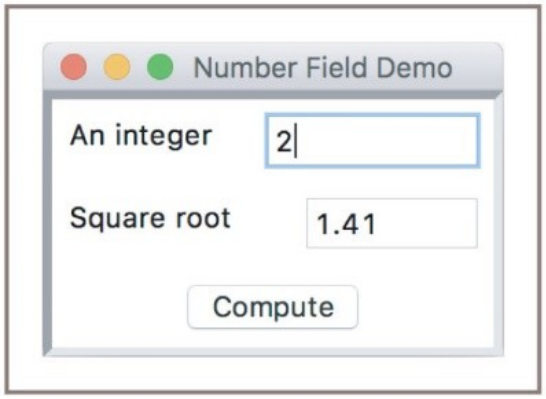


Figure 8-10 Using an integer field and a float field for input and output

Using Pop-Up Message Boxes (1 of 2)

- When errors arise in a GUI-based program
  - Program often responds by popping up a dialog window with an error message
- Code:

```
# The event handling method for the button
def computeSqrt(self):
    """Inputs the integer, computes the square root,
    and outputs the result. Handles input errors
    by displaying a message box."""
    try:
        number = self.inputField.getNumber()
        result = math.sqrt(number)
        self.outputField.setNumber(result)
    except ValueError:
        self.messageBox(title = "ERROR",
                        message = "Input must be an integer >= 0")
```

CENGAGE

Using Pop-Up Message Boxes (2 of 2)

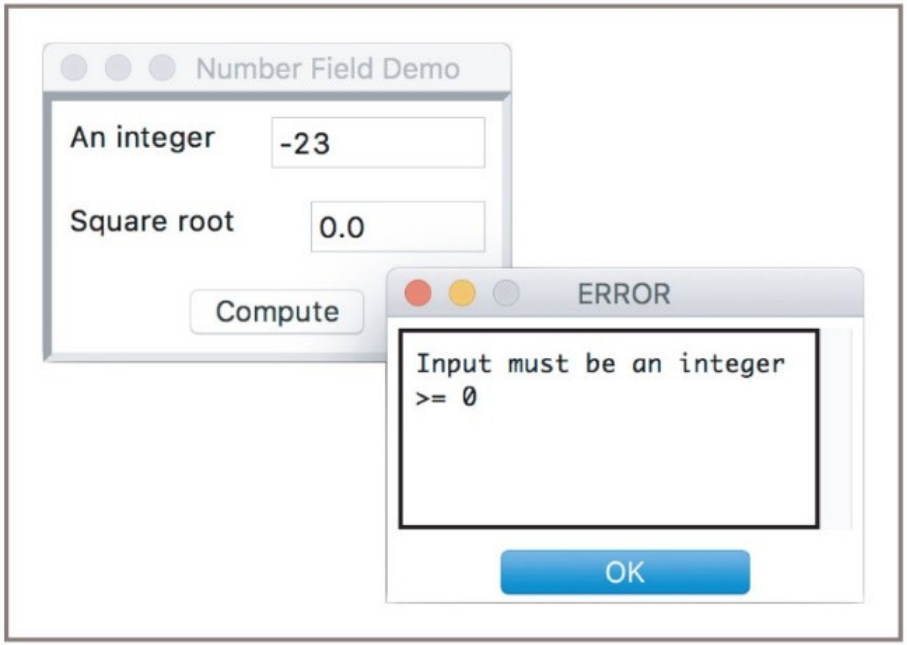


Figure 8-11 Responding to an input error with a message box

Defining and Using Instance Variables (1 of 4)

- **Instance variable**
  - Used to store data belonging to an individual object
- The values of an object's instance variables make up its **state**
- Example: the state of a given window includes its title, background color, and dimensions, among other things
- When you customize an existing class
  - You can add to the state of its objects by including new instance variables
  - Define these new variables (which must begin with the name **self**) within the class's **_init_** method

Defining and Using Instance Variables (2 of 4)

- Code example:

```
class CounterDemo(EasyFrame):
 """ Illustrates the use of a counter with an
   instance variable. """

   def __init__(self):
        """Sets up the window, label, and buttons."""
        EasyFrame.__init__(self, title = "Counter Demo")
        self.setSize(200, 75)

        # Instance variable to track the count.
        self.count = 0

    # A label to display the count in the first row.
        self.label = self.addLabel(text = "0",
                                   row = 0, column = 0,
                                   sticky = "NSEW",
                                   columnspan = 2)
```

Defining and Using Instance Variables (3 of 4)

- Code example (continued):

```
# Two command buttons.
        self.addButton(text = “Next”,
                       row = 1, column = 0,
                      command = self.next)
        self.addButton(text = “Reset”,
                       row = 1, column = 1,
                      command = self.reset)

# Methods to handle user events.
def next(self):
        “”” Increments the count and updates the                    display.”””
        self.count += 1
        self.label[“text”] = str(self.count)
def reset(self):
        “”” Resets the count to 0 and updates the display. ”””
        self.count = 0
        self.label[“text”] = str(self.count)
```

Defining and Using Instance Variables (4 of 4)

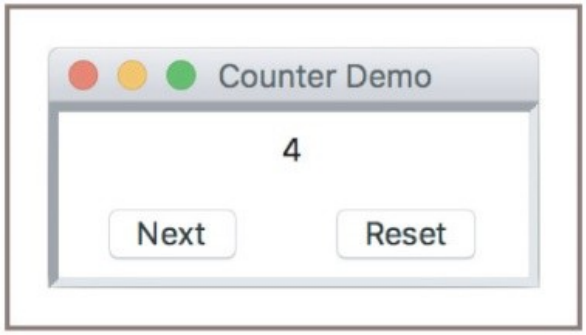


Figure 8-12 The GUI for a counter application

Other Useful GUI Resources

- Layout of GUI components can be specified in more detail
  - Groups of components can be nested in panes
- Paragraphs can be displayed in scrolling text boxes
- Lists of information can be presented for selection in scrolling list boxes as check boxes and radio buttons
- GUI-based programs can be configured to respond to various keyboard events and mouse events

Using Nested Frames to Organize Components (1 of 2)

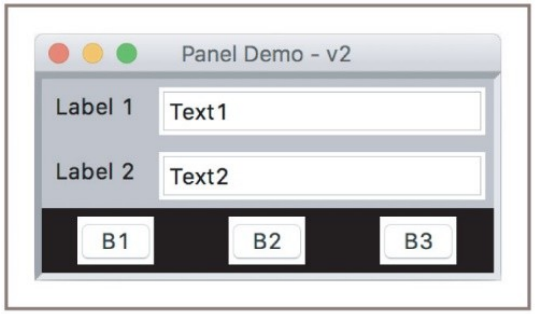


Figure 8-15 Using panels to organize widgets evenly

- Code for laying out the GUI shown in Figure 8-15:

```
class PanelDemo(EasyFrame):

    def __init__(self):

        # Create the main frame
        EasyFrame.__init__(self, "Panel Demo - v2")

        # Create the nested frame for the data panel
        dataPanel = self.addPanel(row = 0, column = 0,
                                  background = "gray")
```

Using Nested Frames to Organize Components (2 of 2)

- Code (continued):

```
# Create and add widgets to the data panel
dataPanel.addLabel(text = “Label 1”, row = 0, column = 0,
                   background = “gray”)
dataPanel.addTextField(text = “Text1”, row = 0, column = 1)
dataPanel.addLabel(text = “Label 2”, row = 1, column = 0,
                   background = “gray”)
dataPanel.addTextField(text = “Text2”, row = 1, column = 1)

#Create the nested frame for the button panel
buttonPanel = self.addPanel(row = 1, column = 0,
                            background = “black”)
# Create and add buttons to the button panel
buttonPanel.addButton(text = “B1”, row = 0, column = 0)
buttonPanel.addButton(text = “B2”, row = 0, column = 1)
buttonPanel.addButton(text = “B3”, row = 0, column = 2)
```

Multi-Line Text Areas (1 of 4)

- The method **addTextArea** adds a text area to the window
  - Returns an object of type **TextArea**, a subclass of **tkinter.Text**
- This object recognizes three important methods: **getText**, **setText**, and **appendText**

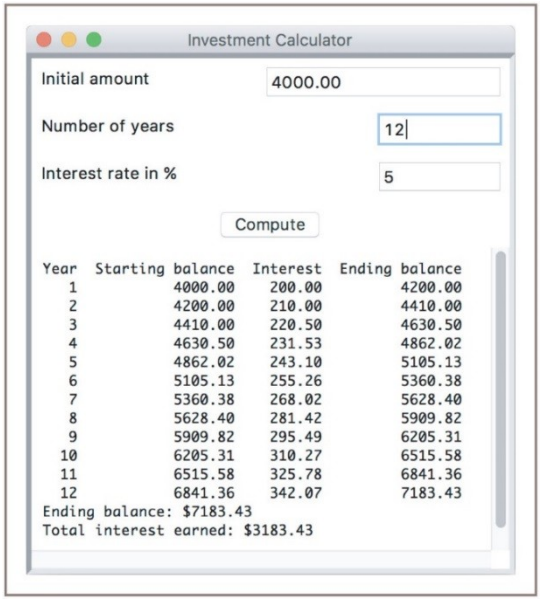


Figure 8-16 Displaying data in a multi-line text area

Multi-Line Text Areas (2 of 4)

```
class TextAreaDemo(EasyFrame):
    """An investment calculator demonstrates the use of a
    multi-line text area. """

    def __init__(self):
        """Sets up the window and widgets. """
        EasyFrame.__init__(self, "Investment Calculator")
        self.addLabel(text = " Initial amount", row = 0,                    column = 0)
        self.addLabel(text = " Number of years", row = 1,                        column = 0)
        self.addLabel(text = " Interest rate in %", row = 2, column = 0)
        self.amount = self.addFloatField(value = 0.0, row = 0,              column = 1)
        self.period = self.addIntegerField(value = 0, row = 1,              column = 1)
        self.rate = self.addIntegerField(value = 0, row = 2,                column = 1)
```

Multi-Line Text Areas (3 of 4)

```
        self.outputArea = self.addTextArea(“ ”, row = 4, column = 0,
                                           columnspan = 2,
                                           width = 50, height = 15)
        self.compute = self.addButton(text = “Compute”, row = 3, column = 0,
                                                  columnspan = 2,command =  self.compute)
# Event handling method.
def compute(self):
        “”” Computes the investment schedule based on the inputs
        and outputs the schedule. ”””
        # Obtain and validate the inputs
        startBalance = self.amount.getNumber()
        rate = self.rate.getNumber() / 100
        years = self.period.getNumber()
        if startBalance == 0 or rate == 0 or years == 0:
                return
```

Multi-Line Text Areas (4 of 4)

```
# Set the header for the table
result = “%4s%18s%10s%16s\n” % (“Year”, "Starting balance",
                                "Interest","Ending balance")
# Compute and append the results for each year
totalInterest = 0.0
for year in range(1, years + 1):
        Interest = startBalance * rate
        endBalance = startBalance + interest
        result += "%4d%18.2f%10.2f%16.2f\n" % (year, startBalance, interest, endBalance)
        startBalance = endBalance
         totalInterest += interest
# Append the totals for the period
result += "Ending balance: $%0.2f\n" % endBalance
result += "Total interest earned: $%0.2f\n" % totalInterest
# Output the result while preserving read-only status
self.outputArea["state"] = "normal"
self.outputArea.setText(result)
self.outputArea["state"] = "disabled"
```

File Dialogs (1 of 2)

- GUI-based programs allow the user to browse the computer's file system with file dialogs
- tkinter.filedialog module includes two functions to support file access in GUI-based programs:
  - askopenfilename and asksaveasfilename
- Syntax:

```
fList = [("Python files", "*.py"), ("Text files", "*.txt")]
filename = tkinter.filedialog.askopenfilename(parent = self,
                                              filetypes = fList)
filename = tkinter.filedialog.asksaveasfilename(parent = self)
```

File Dialogs (2 of 2)

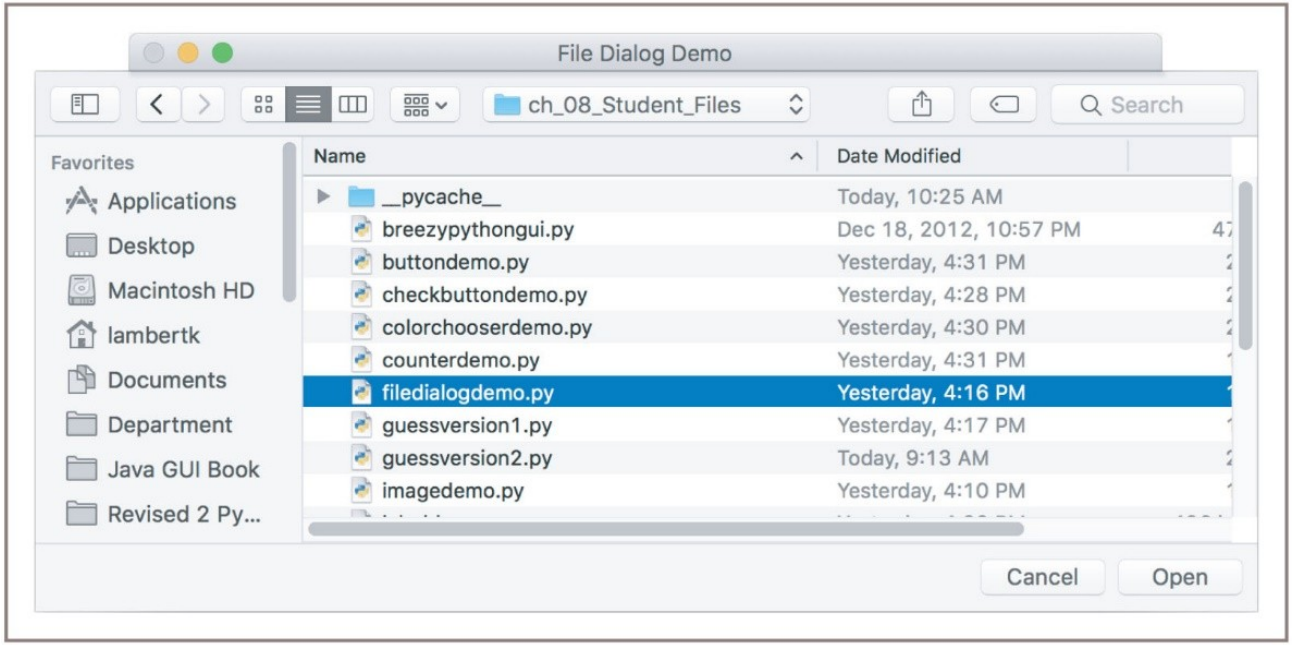


Figure 8-17 A file dialog

Obtaining Input with Prompter Boxes (1 of 2)

```
class PrompterBoxDemo(EasyFrame):
    def __init__(self):
        """ Sets up the window and widgets."""
        EasyFrame.__init__(self, title = "Prompter Box Demo", width = 300, height = 100)
        self.label = self.addLabel(text = " ", row = 0, column = 0, sticky = "NSEW")
        self.addButton(text = "Username", row = 1, column =  0, command = self.getUserName)

    def getUserName(self):
        text = self.prompterBox(title = "Input Dialog", promptString = "Your username:")
        self.label["text"] = "Hi " + name + "!"
```

Obtaining Input with Prompter Boxes (2 of 2)

Figure 8-19 Using a prompter box

Check Buttons

- Check button
  - Consists of a label and a box that a user can select or deselect with the mouse
- The method **addCheckbutton** expects a **text** argument and an optional command argument

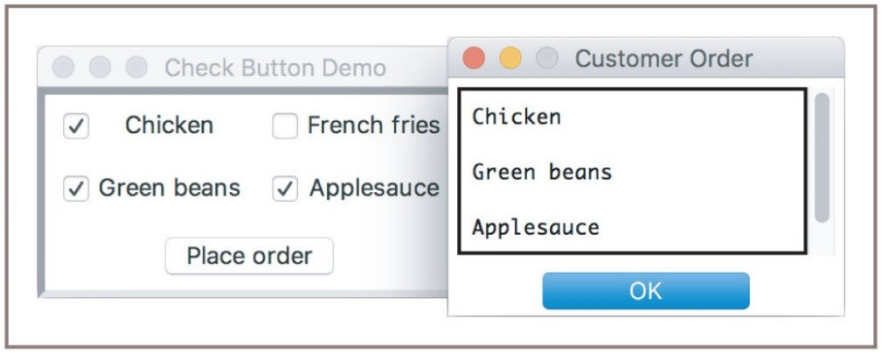


Figure 8-20 Using check buttons

Radio Buttons (1 of 2)

- Radio buttons
  - Used when the user must be restricted to one selection only
  - Consists of a label and a control widget
- The **EasyRadiobuttonGroup** method **getSelectedButton** returns the currently selected radio button in a radio button group
- The method **setSelectionButton** selects a radio button under program control
  - Once a radio button group is created, the programmer can add radio buttons to it with the **EasyRadiobuttonGroup** method **addRadiobutton**
  - This method expects a **text** argument and an optional **command** argument

CENGAGE

Radio Buttons (2 of 2)

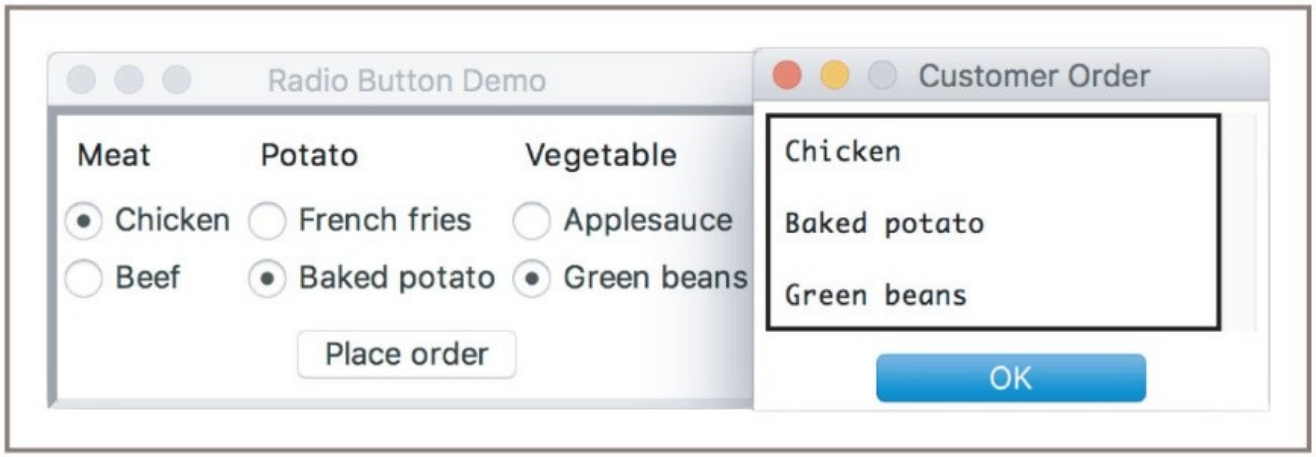


Figure 8-21 Using radio buttons

Keyboard Events

- You can associate a keyboard event and an event-handling method with a widget by calling the bind method
  - This method expects a string containing a key event as its first argument and the method to be triggered as its second argument
- The string for the return key event is “<Return>”
- The event-handling method should have a single parameter named event
- You bind the keyboard return event to a handler for the inputField widget:

```
self.inputField.bind("<Return>",
                     lambda event: self.computeSqrt())
```

Working with Colors (1 of 2)

- Python represents an RGB value as a string containing a six-digit hexadecimal number
  - Of the form “0xRRGGBB”
- The tkinter module also accepts the simpler representation “#RRGGBB” for hexadecimal values (called a hex string)

Working with Colors (2 of 2)

Ordinary Value	RGB Triple	Hex String
“black”	(0,0,0)	“#000000”
“red”	(255,0,0)	“#ff0000”
“green”	(0,255,0)	“#00ff00”
“blue”	(0,0,255)	“#0000ff”
“gray”	(127,127,127)	“#7f7f7f”
“white”	(255,255,255)	“#ffffff”

CENGAGE

Using a Color Chooser

- Most graphics software packages allow the user to pick a color with a standard color chooser

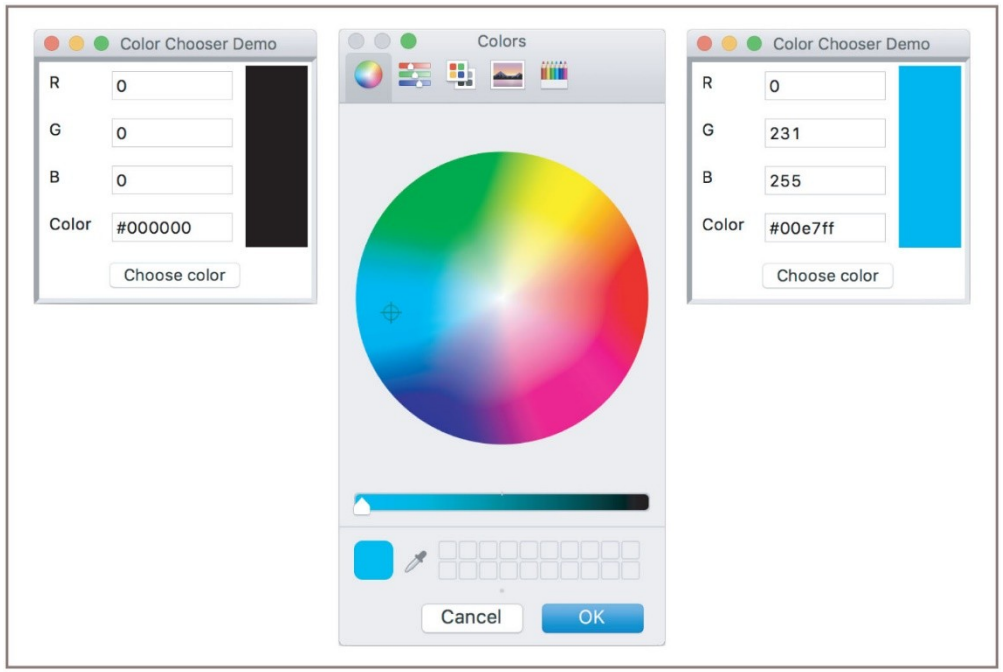


Figure 8-22 Using a color chooser

Chapter Summary (1 of 2)

- A GUI-based program responds to user events by running methods to perform various tasks
  - The model/view/controller pattern assigns the roles and responsibilities to three different sets of classes
- **tkinter** and **breezypythongui** module includes classes, functions, and constants used in GUI programming
- A GUI-based program is structured as a main window class (extends the **Frame** class)
- Examples of window components: labels, entry fields, command buttons, text areas, and list boxes

Chapter Summary (2 of 2)

- Pop-up dialog boxes display messages and ask yes/no question (**tkinter.messagebox** module)
- Objects can be arranged using grids and panes
- Each component has attributes for the foreground color and background color
- Text has a type font attribute
- The **command** attribute of a button can be set to a method that handles a button click
- Mouse and keyboard events can be associated with handler methods for window objects (**bind**)

CENGAGE