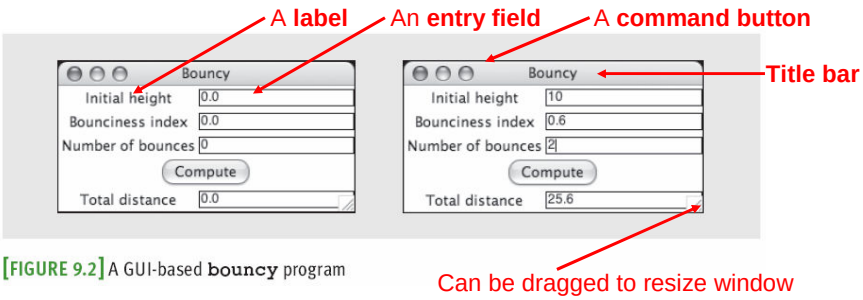
Introduction

- 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
- *https://docs.python.org/3.6/library/tkinter.html*

The GUI-Based Version

- Uses a window that contains various components
  - Called **window objects** or **widgets**

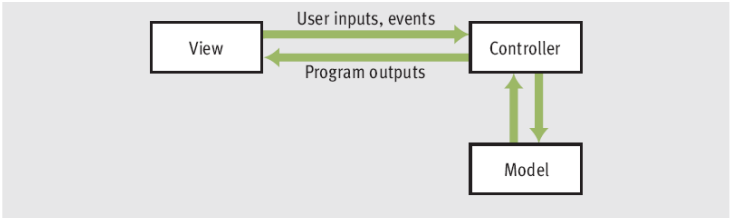


[FIGURE 9.2] A GUI-based `bouncy` program

- Solves problems of terminal-based version

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**



[FIGURE 9.3] The model/view/controller pattern

Event-Driven Programming (continued)

- 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

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
  - **`tkinter.messagebox`** includes functions for several standard pop-up dialog boxes

Windows and Labels

- 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()
```


Windows and Labels (continued)

- 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)

[FIGURE 9.4] Displaying a text label in a window

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 (continued)

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

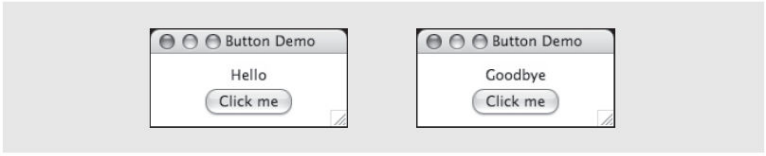
[FIGURE 9.5] Displaying a captioned image

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"
```

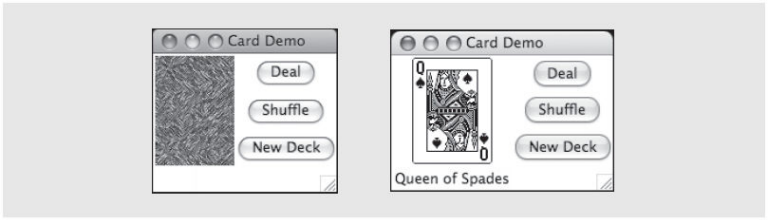

Command Buttons and Responding to Events (continued)

- 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



[FIGURE 9.6] When the user presses the Click me button, the message changes from "Hello" to "Goodbye"

Viewing the Images of Playing Cards



[FIGURE 9.7] A GUI for viewing playing cards

```
BACK_NAME = 'DECK/b.gif'

def __init__(self, rank, suit):
    """Creates a card with the given rank, suit, and
    image filename."""
    self.rank = rank
    self.suit = suit
    self.fileName = 'DECK/' + str(rank) + suit[0] + '.gif'
```


Entry Fields for the Input and Output of Text

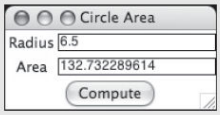
- A **form filler** consists of labeled **entry fields**, which allow the user to enter and edit a single line of text
- A field can also contain text output by a program
- **`tkinter's Entry`** displays an entry field
- Three types of data **container objects** can be used with **`Entry`** fields:

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

[TABLE 9.1] Data container classes for different data types

Entry Fields for the Input and Output of Text (continued)

```
    def _area(self):
        """Event handler for the button."""
        radius = self._radiusVar.get()
        area = radius ** 2 * math.pi
        self._areaVar.set(area)

def main():
    CircleArea().mainloop()
```



[FIGURE 9.8] The `circlearea` program recast as a GUI program

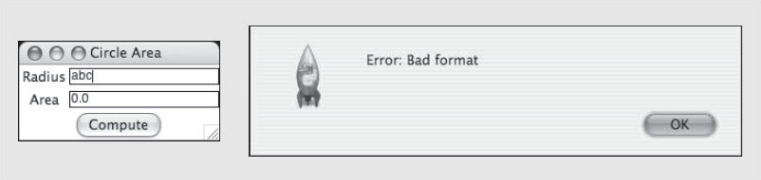
Using Pop-up Dialog Boxes

`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 (continued)

```
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)
```



[FIGURE 9.9] A pop-up dialog box with an error message

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 and drop-down menus
- Color, size, and style of text and of some GUI components can be adjusted
- GUI-based programs can be configured to respond to various keyboard events and mouse events

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`.

[TABLE 9.3] Font attributes

Text Attributes (continued)

- Example:

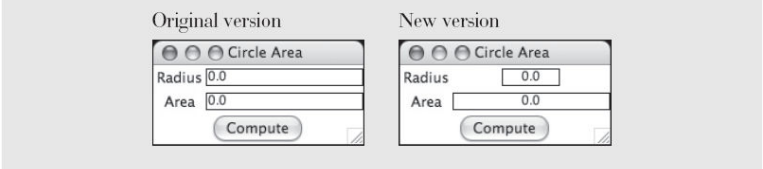
```
font = tkinter.font.Font(family = "Verdana",
                         size = 20, slant = "italic")
self._label = Label(self, font = font, text = "Hello world!")
```

[FIGURE 9.11] Setting a type font

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)
```



[FIGURE 9.12] Setting the size and justification of entry fields

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

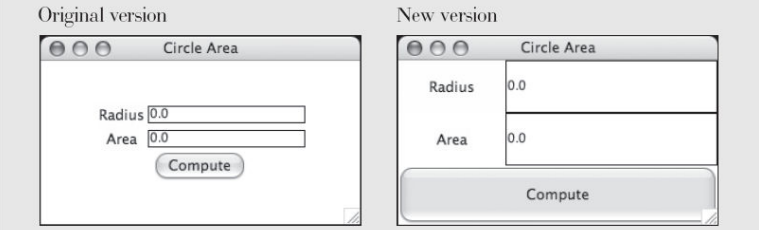
Grid Attributes

- 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 Attributes (continued)

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

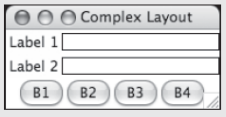
```
for row in range(3):
    self.rowconfigure(row, weight = 1)
for column in range(2):
    self.columnconfigure(column, weight = 1)
```



[FIGURE 9.15] The `circlearea` GUI with widget expansion

Using Nested Frames to Organize Components

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



[FIGURE 9.16] A complex grid layout

- 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 (continued)

- 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)
```


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.

[TABLE 9.5] Some **Listbox** methods

Scrolling List Boxes (continued)

```
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.

[TABLE 9.6] Mouse events

Mouse Events (continued)

- 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.

[TABLE 9.7] Some key events

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

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

Summary

- 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`** 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

Summary (continued)

- 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`**)


EAST CENTRAL UNIVERSITY
1909