KTU STUDY MATERIALS

# PROGRAMMING IN PYTHON

# CST 362

# Module 3

# Module 3: Graphics

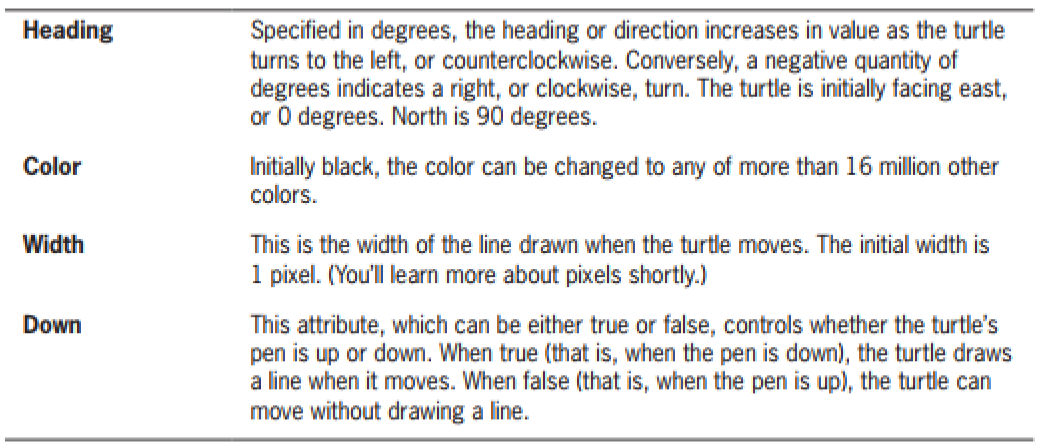
## Simple Graphics

Graphics is the discipline that underlies the representation and display of geometric shapes in two- and three-dimensional space, as well as image processing. Python comes with a large array of resources that support graphics operations. However, these operations are complex and not for the faint of heart. To help you ease into the world of graphics, this section provides an introduction to a gentler set of graphics operations known as turtle graphics. A Turtle graphics toolkit provides a simple and enjoyable way to draw pictures in a window and gives you an opportunity to run several methods with an object.

## Turtle Graphics

The sheet of paper is a window on a display screen, and the turtle is an icon, such as an arrowhead. At any given moment in time, the turtle is located at a specific position in the window. This position is specified with (x, y) coordinates. The coordinate system for Turtle graphics is the standard Cartesian system, with the origin (0, 0) at the center of a window. The turtle's initial position is the origin which is also called the **home**. An equally important attribute of a turtle is its heading, or the direction in which it currently faces. The turtle's initial heading is 0 degrees, or due east on its map. The degrees of the heading increase as it turns to the left, so 90 degrees is due north.

In addition to its position and heading, a turtle also has several other attributes :

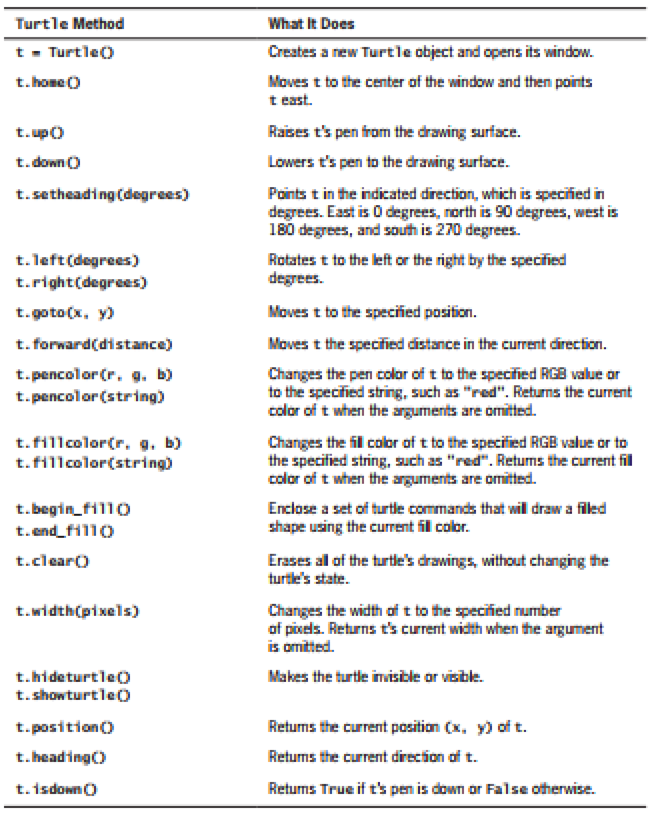


| | |
|---|---|
| **Heading** | Specified in degrees, the heading or direction increases in value as the turtle turns to the left, or counterclockwise. Conversely, a negative quantity of degrees indicates a right, or clockwise, turn. The turtle is initially facing east, or 0 degrees. North is 90 degrees. |
| **Color** | Initially black, the color can be changed to any of more than 16 million other colors. |
| **Width** | This is the width of the line drawn when the turtle moves. The initial width is 1 pixel. (You'll learn more about pixels shortly.) |
| **Down** | This attribute, which can be either true or false, controls whether the turtle's pen is up or down. When true (that is, when the pen is down), the turtle draws a line when it moves. When false (that is, when the pen is up), the turtle can move without drawing a line. |

these attributes make up a turtle's state. The concept of state is a very important one in object-based programming. Generally, an object's state is the set of values of its attributes at any given point in time. The turtle's state determines how the turtle will behave when any operations are applied to it. For example, a turtle will draw when it is moved if its pen is currently down, but it will simply move without drawing when its pen is currently up. Operations also change a turtle's state. For instance, moving a turtle changes its position, but not its direction, pen width, or pen color.

## Turtle Operations

Every data value in Python is an object. The types of objects are called classes. Included in a class are the methods (or operations) that apply to objects of that class. Because a turtle is an object, its operations are also defined as methods. Table lists some of the methods belonging to the Turtle class. In this table, the variable **t** refers to a particular Turtle object.



| Turtle Method | What It Does |
|---|---|
| `t = Turtle()` | Creates a new `Turtle` object and opens its window. |
| `t.home()` | Moves `t` to the center of the window and then points `t` east. |
| `t.up()` | Raises `t`'s pen from the drawing surface. |
| `t.down()` | Lowers `t`'s pen to the drawing surface. |
| `t.setheading(degrees)` | Points `t` in the indicated direction, which is specified in degrees. East is 0 degrees, north is 90 degrees, west is 180 degrees, and south is 270 degrees. |
| `t.left(degrees)` <br> `t.right(degrees)` | Rotates `t` to the left or the right by the specified degrees. |
| `t.goto(x, y)` | Moves `t` to the specified position. |
| `t.forward(distance)` | Moves `t` the specified distance in the current direction. |
| `t.pencolor(r, g, b)` <br> `t.pencolor(string)` | Changes the pen color of `t` to the specified RGB value or to the specified string, such as `"red"`. Returns the current color of `t` when the arguments are omitted. |
| `t.fillcolor(r, g, b)` <br> `t.fillcolor(string)` | Changes the fill color of `t` to the specified RGB value or to the specified string, such as `"red"`. Returns the current fill color of `t` when the arguments are omitted. |
| `t.begin_fill()` <br> `t.end_fill()` | Enclose a set of turtle commands that will draw a filled shape using the current fill color. |
| `t.clear()` | Erases all of the turtle's drawings, without changing the turtle's state. |
| `t.width(pixels)` | Changes the width of `t` to the specified number of pixels. Returns `t`'s current width when the argument is omitted. |
| `t.hideturtle()` <br> `t.showturtle()` | Makes the turtle invisible or visible. |
| `t.position()` | Returns the current position `(x, y)` of `t`. |
| `t.heading()` | Returns the current direction of `t`. |
| `t.isdown()` | Returns `True` if `t`'s pen is down or `False` otherwise. |

To illustrate the use of some methods with a Turtle object, let's define a function named **drawSquare**. This function expects a Turtle object, a pair of integers that indicate the coordinates of the square's upper-left corner, and an integer that designates the length of a side. The function begins by lifting the turtle up and moving it to the square's corner point. It then points the turtle due south—270 degrees—and places the turtle's pen down on the drawing surface. Finally, it moves the turtle the given length and turns it left by 90 degrees, four times. Here is the code for the drawSquare function:

```
def drawSquare(t, x, y, length):
"""Draws a square with the given turtle t, an upper-left
corner point (x, y), and a side's length."""
        t.up()
        t.goto(x, y)
        t.setheading(270)
        t.down()
        for count in range(4):
                t.forward(length)
                t.left(90)
```

The Turtle class is defined in the turtle module (note carefully the spelling of both names). The following code imports the Turtle class for use in a session:

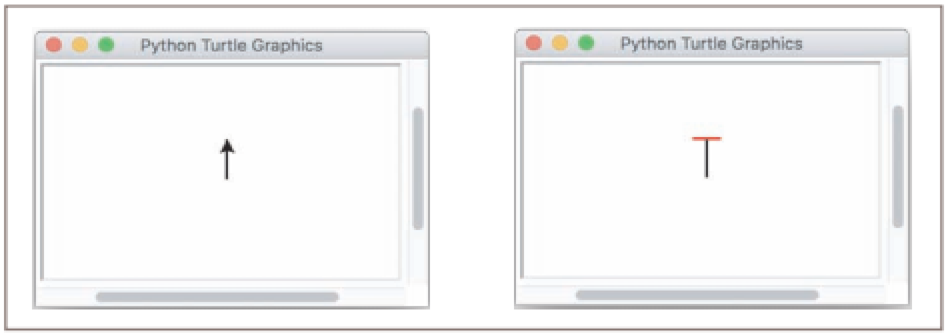
```
>>> from turtle import Turtle
```

The next code segment creates and returns a Turtle object and opens a drawing window.

```
>>> t = Turtle()
```

The session with the code follows. shows screenshots of the window after each line segment is drawn.

```
>>> t.width(2)              # For bolder lines
>>> t.left(90)              # Turn to face north
>>> t.forward(30)           # Draw a vertical line in black
>>> t.left(90)              # Turn to face west
>>> t.up()                  # Prepare to move without drawing
>>> t.forward(10)           # Move to beginning of horizontal line
>>> t.setheading(0)         # Turn to face east
>>> t.pencolor("red")
>>> t.down()                # Prepare to draw
>>> t.forward(20)           # Draw a horizontal line in red
>>> t.hideturtle()          # Make the turtle invisible
```

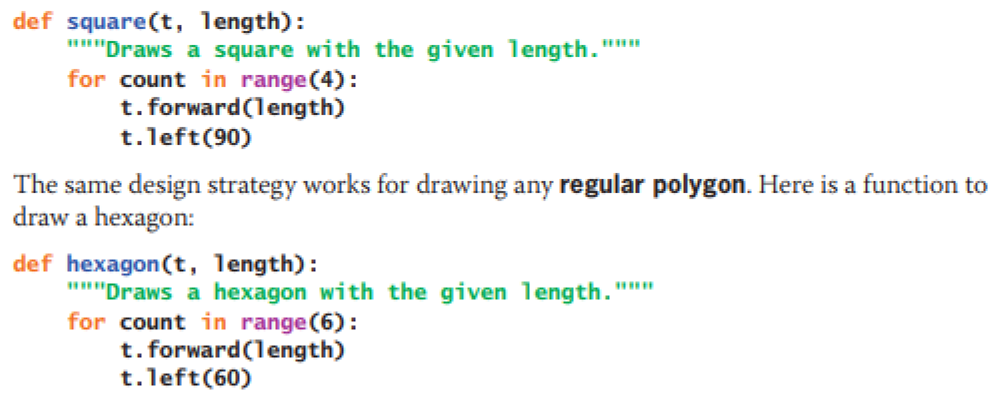


**Drawing Two-Dimensional Shapes**

Many graphics applications use vector graphics, which includes the drawing of simple two-dimensional shapes, such as rectangles, triangles, pentagons, and circles. Earlier we defined a drawSquare function that draws a square with a given corner point and length, and we could do the same for other types of shapes as well. However, our design of the drawSquare function has two limitations:

1. The caller must provide the shape's location, such as a corner point, as an argument, even though the turtle itself could already provide this location
2. The shape is always oriented in the same way, even though the turtle itself could provide the orientation.

A more general method of drawing a square would receive just its length and the turtle as arguments, and begin drawing from the turtle's current heading and position. Here is the code for the new function to draw squares, named **square**:



```
def square(t, length):
    """Draws a square with the given length."""
    for count in range(4):
        t.forward(length)
        t.left(90)
```

The same design strategy works for drawing any **regular polygon**. Here is a function to draw a hexagon:

```
def hexagon(t, length):
    """Draws a hexagon with the given length."""
    for count in range(6):
        t.forward(length)
        t.left(60)
```

**Colors and the RGB System**

The rectangular display area on a computer screen is made up of colored dots called picture elements or pixels. The smaller the pixel, the smoother the lines drawn with them will be. The size of a pixel is determined by the size and resolution of the display. For example, one common screen resolution is 1680 pixels by 1050 pixels, which, on a 20-inch monitor, produces a rectangular display area that is 17 inches by 10.5 inches. Setting the resolution to smaller values increases the size of the pixels, making the lines on the screen appear more ragged. Each pixel represents a color. While the turtle's default color is black, you can easily change it to one of several other basic colors, such as red, yellow, or orange, by running the pencolor method with the corresponding string as an argument. To provide the full range of several million colors available on today's computers, we need a more powerful representation scheme. Among the various schemes for representing colors, the rGB system is a common one. The letters stand for the color components of red, green, and blue, to which the human retina is sensitive. These components are mixed together to form a unique color value. Naturally, the computer represents these values as integers, and the display hardware translates this information to the colors you see. Each color component can range from 0 through 255. The value 255 represents the maximum saturation of a given color component, whereas the value 0 represents the total absence of that component. Table lists some example colors and their RGB values. You might be wondering how many total RGB color values are at your disposal. That number would be equal to all the possible combinations of three values, each of which has 256 possible values, or 256 * 256 * 256, or 16,777,216 distinct color values. Although the human eye cannot discriminate between adjacent color values in this set, the RGB system is called a **true color system**.

| Color | RGB Value |
|---|---|
| Black | (0, 0, 0) |
| Red | (255, 0, 0) |
| Green | (0, 255, 0) |
| Blue | (0, 0, 255) |
| Yellow | (255, 255, 0) |
| Gray | (127, 127, 127) |
| White | (255, 255, 255) |

**Example: Filling Radial Patterns with Random Colors**

The Turtle class includes the pencolor and fillcolor methods for changing the turtle's drawing and fill colors, respectively. These methods can accept integers for the three RGB components as arguments. The next script draws radial patterns of squares and hexagons with random fill colors at the corners of the turtle's window.

```
from turtle import Turtle
from polygons import *
import random

def drawPattern(t, x, y, count, length, shape):
    """Draws a radial pattern with a random
    fill color at the given position."""
    t.begin_fill()
    t.up()
    t.goto(x, y)
    t.setheading(0)
    t.down()
    t.fillcolor(random.randint(0, 255),
                random.randint(0, 255),
                random.randint(0, 255))
    radialPattern(t, count, length, shape)
    t.end_fill()

def main():
    t = Turtle()
    t.speed(0)
    # Number of shapes in radial pattern
    count = 10
    # Relative distances to corners of window from center
    width = t.screen.window_width() // 2
    height = t.screen.window_height() // 2
    # Length of the square
    length = 30
    # Inset distance from window boundary for squares
    inset = length * 2
    # Draw squares in upper-left corner
    drawPattern(t, -width + inset, height - inset, count,
                length, square)
    # Draw squares in lower-left corner
    drawPattern(t, -width + inset, inset - height, count,
                length, square)
    # Length of the hexagon
    length = 20
    # Inset distance from window boundary for hexagons
    inset = length * 3
    # Draw hexagons in upper-right corner
    drawPattern(t, width - inset, height - inset, count,
                length, hexagon)

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

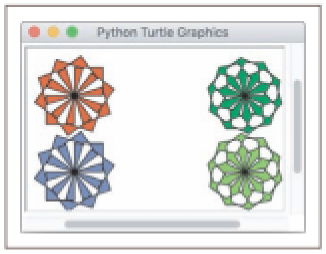


Figure 7-5 Radial patterns with random fill colors

## Image processing

The most recent form of this type of technology is digital image processing. This enormous field includes the principles and techniques for the following:

- The capture of images with devices such as flatbed scanners and digital cameras
- The representation and storage of images in efficient file formats
- Constructing the algorithms in image-manipulation programs such as Adobe Photoshop

Representing photographic images in a computer poses an interesting problem. As you have seen, computers must use digital information which consists of discrete values, such as individual integers, characters of text, or bits in a bit string. However, the information contained in images, sound, and much of the rest of the physical world is analog. Analog information contains a continuous range of values. An analog clock displays the seconds as tick marks on a circle. The clock's second hand passes by these marks as it sweeps around the clock's face. This sweep reveals the analog nature of time: between any two tick marks on the analog clock, there is a continuous range of positions or moments of time through which the second hand passes. . The ticks representing seconds on the analog clock's face thus represent an attempt to sample moments of time as discrete values, whereas time itself is continuous, or analog.

**Image-Manipulation Operations**

Image-manipulation programs either transform the information in the pixels or alter the arrangement of the pixels in the image. These programs also provide fairly low-level operations for transferring images to and from file storage. Among other things, these programs can do the following:

- Rotate an image
- Convert an image from color to grayscale
- Apply color filtering to an image
- Highlight a particular area in an image
- Blur all or part of an image
- Sharpen all or part of an image
- Control the brightness of an image
- Perform edge detection on an image
- Enlarge or reduce an image's size
- Apply color inversion to an image
- Morph an image into another image

You'll learn how to write Python code that can perform some of these manipulation tasks later in this chapter, and you will have a chance to practice others in the programming Projects.

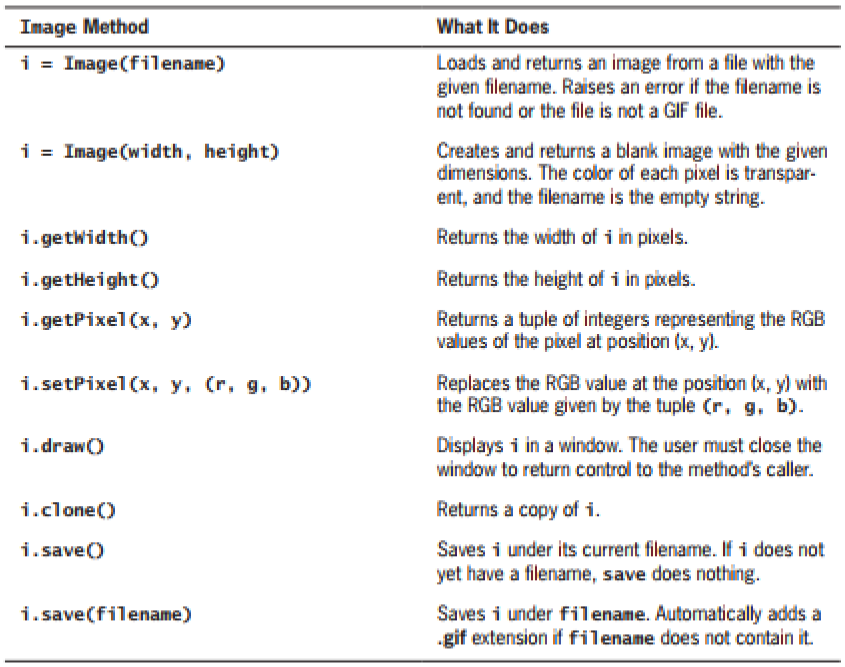
**The Properties of Images**

When an image is loaded into a program such as a Web browser, the software maps the bits from the image file into a rectangular area of colored pixels for display. The coordinates of the pixels in this two-dimensional grid range from (0, 0) at the upper-left corner of an image to (width – 1, height – 1) at the lower-right corner, where width and height are the image's dimensions in pixels. Thus, the **screen coordinate system** for the display of an image is somewhat different from the standard Cartesian coordinate system that we used with Turtle graphics, where the origin (0, 0) is at the center of the rectangular grid. Them RGB color system introduced earlier in this chapter is a common way of representing the colors in images. For our purposes, then, an image consists of a width, a height, and a set of color values accessible by means of (x, y) coordinates. A color value consists of the tuple (r, g, b), where the variables refer to the integer values of its red, green, and blue components, respectively.

**The images Module**

To facilitate our discussion of image-processing algorithms, we now present a small module of high-level Python resources for image processing. This package of resources, which is named images, allows the programmer to load an image from a file, view the image in a window, examine and manipulate an image's RGB values, and save the image to a file. The images module is a non-standard, open-source Python tool.The images module includes a class named Image. The Image class represents an image as a two-dimensional grid of RGB values.

The images module includes a class named Image. The Image class represents an image as a two-dimensional grid of RGB values.



| Image Method | What It Does |
|---|---|
| `i = Image(filename)` | Loads and returns an image from a file with the given filename. Raises an error if the filename is not found or the file is not a GIF file. |
| `i = Image(width, height)` | Creates and returns a blank image with the given dimensions. The color of each pixel is transparent, and the filename is the empty string. |
| `i.getWidth()` | Returns the width of `i` in pixels. |
| `i.getHeight()` | Returns the height of `i` in pixels. |
| `i.getPixel(x, y)` | Returns a tuple of integers representing the RGB values of the pixel at position (x, y). |
| `i.setPixel(x, y, (r, g, b))` | Replaces the RGB value at the position (x, y) with the RGB value given by the tuple `(r, g, b)`. |
| `i.draw()` | Displays `i` in a window. The user must close the window to return control to the method's caller. |
| `i.clone()` | Returns a copy of `i`. |
| `i.save()` | Saves `i` under its current filename. If `i` does not yet have a filename, `save` does nothing. |
| `i.save(filename)` | Saves `i` under `filename`. Automatically adds a .gif extension if `filename` does not contain it. |

This version of the images module accepts only image files in GIF format. For the purposes of this exercise, we also assume that a GIF image of my cat, Smokey, has been saved in a file named smokey.gif in the current working directory. The following session with the interpreter does three things:

1. Imports the Image class from the images module
2. Instantiates this class using the file named smokey.gif
3. Draws the image

The resulting image display window is shown in Figure 7-9.

```
>>> from images import Image
>>> image = Image("smokey.gif")
>>> image.draw()
```

Python raises an exception if it cannot locate the file in the current directory, or if the file is not a GIF file. Note also that the user must close the window to return control to the caller of the method draw. If you are working in the shell, the shell prompt will reappear when you do this. The image can then be redrawn, after other operations are performed, by calling draw again.

Once an image has been created, you can examine its width and height, as follows:

```
>>> image.getWidth()
198
>>> image.getHeight()
149
```

Alternatively, you can print the image's string representation:

```
>>> print(image)
Filename: smokey.gif
Width: 198
Height: 149
```

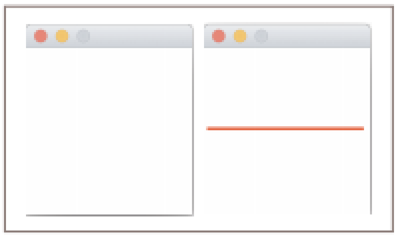
The method getPixel returns a tuple of the RGB values at the given coordinates. The following session shows the information for the pixel at position (0, 0), which is at the image's upper-left corner.

```
>>> image.getPixel(0, 0)
(194, 221, 114)
```

Instead of loading an existing image from a file, the programmer can create a new, blank image. The programmer specifies the image's width and height; the resulting image consists of transparent pixels. Such images are useful for creating backgrounds for drawing simple shapes, or for creating new images that receive information from existing images.

The programmer can use the method setPixel to replace an RGB value at a given position in an image. The next session creates a new 150-by-150 image. The pixels along the three horizontal lines at the middle of the image are then replaced with new blue pixels.

```
>>> image = Image(150, 150)
>>> image.draw()
>>> blue = (0, 0, 255)
>>> y = image.getHeight() // 2
>>> for x in range(image.getWidth()):
        image.setPixel(x, y - 1, blue)
        image.setPixel(x, y, blue)
        image.setPixel(x, y + 1, blue)
>>> image.draw()
```



**Converting an Image to Black and White**

```
def blackAndWhite(image):
    """Converts the argument image to black and white."""
    blackPixel = (0, 0, 0)
    whitePixel = (255, 255, 255)
    for y in range(image.getHeight()):
        for x in range(image.getWidth()):
        (r, g, b) = image.getPixel(x, y)
        average = (r + g + b) // 3
        if average < 128:
            image.setPixel(x, y, blackPixel)
        else:
            image.setPixel(x, y, whitePixel)
```

The function can be tested in a short script, as follows:

```
from images import Image
# Code for blackAndWhite's function definition goes here
def main(filename = "smokey.gif"):
    image = Image(filename)
    print("Close the image window to continue.")
    image.draw()
    blackAndWhite(image)
    print("Close the image window to quit.")
    image.draw()
if __name__ == "__main__":
main()
```

## Converting an Image to Grayscale

A scheme that combines the three components needs to take these differences in luminance into account. A more accurate method would weight green more than red and red more than blue. Therefore, to obtain the new RGB values, instead of adding up the color values and dividing by 3, you should multiply each one by a weight factor and add the results. Psychologists have determined that the relative luminance proportions of green, red, and blue are .587, .299, and .114, respectively.

```
def grayscale(image):
"""Converts the argument image to grayscale."""
        for y in range(image.getHeight()):
                for x in range(image.getWidth()):
                        (r, g, b) = image.getPixel(x, y)
                        r = int(r * 0.299)
                        g = int(g * 0.587)
                        b = int(b * 0.114)
                        lum = r + g + b
                        image.setPixel(x, y, (lum, lum, lum))
```

**(Refer text book for the programs Blurring an Image Edge Detection Reducing the Image Size)**

## Graphical User Interfaces

Most interactive computer software employs a graphical user interface or GUI (or its close relative, the touchscreen interface). A GUI displays text as well as small images (called icons) that represent objects such as folders, files of different types, command buttons, and drop-down menus. In addition to entering text at the keyboard, the user of a GUI can select some of these icons with a pointing device, such as a mouse, and move them around on the display. Commands can be activated by pressing the enter key or control keys, by pressing a command button, by selecting a drop-down menu item, or by double-clicking on some icons with the mouse. Put more simply, a GUI displays all information, including text, graphically to its users and allows them to manipulate this information directly with a pointing device.

### The Behavior of terminal-Based programs and GUI-Based programs

A **GUI program** is **event driven**, meaning that it is inactive until the user clicks a button or selects a menu option. In contrast, a **terminal-based program** maintains constant control over the interactions with the user. Put differently, a terminal-based program prompts users to enter successive inputs, whereas a GUI program puts users in change, allowing them to enter inputs in any order and waiting for them to press a command button or select a menu option.

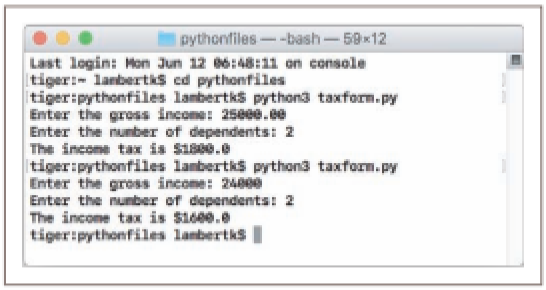
**The Terminal-Based Version**

The terminal-based version of the program prompts the user for his gross income and number of dependents. After he enters his inputs, the program responds by computing and displaying his income tax. The program then terminates execution.

This terminal-based user interface has several obvious effects on its users:

- The user is constrained to reply to a definite sequence of prompts for inputs. Once an input is entered, there is no way to back up and change it.

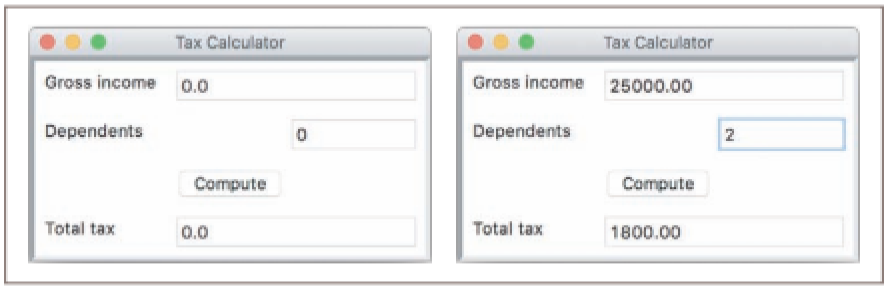
• To obtain results for a different set of input data, the user must run the program again. At that point, all of the inputs must be re-entered.



```
Last login: Mon Jun 12 06:48:11 on console
tiger:~ lambertk$ cd pythonfiles
tiger:pythonfiles lambertk$ python3 taxform.py
Enter the gross income: 25000.00
Enter the number of dependents: 2
The income tax is $1800.0
tiger:pythonfiles lambertk$ python3 taxform.py
Enter the gross income: 24000
Enter the number of dependents: 2
The income tax is $1600.0
tiger:pythonfiles lambertk$
```

### The GUI-Based Version

The GUI-based version of the program displays a window that contains various components, also called widgets. Some of these components look like text, while others provide  visual cues as to their use.. The snapshot on the left shows the interface at program start-up, whereas the snapshot on the right shows the interface after the user has entered inputs and clicked the Compute button.



• A title bar at the top of the window. This bar contains the title of the program, "Tax Calculator." It also contains three colored disks. Each disk is a command button. The user can use the mouse to click the left disk to quit the program, the middle disk to minimize the window, or the right disk to zoom the window. The user can also move the window around the screen by holding the left mouse button on the title bar and dragging the mouse.
• A set of labels along the left side of the window. These are text elements that describe the inputs and outputs. For example, "Gross income" is one label.
• A set of entry fields along the right side of the window. These are boxes within which the program can output text or receive it as input from the user. The first two entry fields will be used for inputs, while the last field will be used for the output. At program start-up, the fields contain default values.
• A single command button labeled Compute. When the user uses the mouse to press this button, the program responds by using the data in the two input fields to compute the income tax. This result is then displayed in the output field.

- The user can also alter the size of the window by holding the mouse on its lower-right corner and dragging in any direction.

Although this review of features might seem tedious to anyone who regularly uses GUIbased programs, a careful inventory is necessary for the programmer who builds them.
Also, a close study of these features reveals the following effects on users:

- The user is not constrained to enter inputs in a particular order. Before she presses the Compute button, she can edit any of the data in the two input fields.
- Running different data sets does not require re-entering all of the data. The user can edit just one value and press the Compute button to observe different results.

## Event-Driven Programming

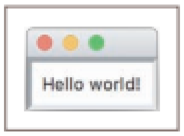
a GUI-based program opens a window and waits for the user to manipulate window components with the mouse. These user-generated events, such as mouse clicks, trigger operations in the program to respond by pulling in inputs, processing them, and displaying results. This type of software system is event-driven, and the type of programming used to create it is called event-driven programming.
In the analysis step, the types of window components and their arrangement in the window are determined. Because GUI-based programs are almost always object based, this becomes a matter of choosing among GUI component classes available in the programming language or inventing new ones if needed. The example of the tax calculator program to see how it might be structured as an event-driven program. The GUI in this program consists of the window and its components, including the labeled entry fields and the Compute button. The action triggered when this button is clicked is a method call. This method fetches the input values from the input fields and performs the computation. The result is then sent to the output field to be displayed.
Once the interactions among these resources have been determined, their coding can begin. This phase consists of several steps:

1. Define a new class to represent the main application window.
2. Instantiate the classes of window components needed for this application, such as labels, fields, and command buttons.
3. Position these components in the window.
4. Register a method with each window component in which an event relevant to the application might occur.
5. Define these methods to handle the events.
6. Define a main function that instantiates the window class and runs the appropriate method to launch the GUI.

**Coding Simple GUI-Based programs**

Python's standard tkinter module includes classes for windows and numerous types of window components, but its use can be challenging for beginners. Therefore, this book uses a custom, open-source module called breezypythongui, while occasionally relying upon some of the simpler resources of tkinter.

**A Simple "Hello World" Program**



```
"""
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()
```

We will speak more generally about class definitions shortly. For now, note that this program performs the following steps:

1. Import the EasyFrame class from the breezypythongui module. This class is a subclass of tkinter's Frame class, which represents a top-level window. In many GUI programs, this is the only import that you will need.
2. Define the LabelDemo class as a subclass of EasyFrame. The LabelDemo class describes the window's layout and functionality for this application.
3. Define an __init__ method in the LabelDemo class. This method is automatically run when the window is created. The __init__ method runs a method with the same name on the EasyFrame class and then sets up any window components to display in the window. In this case, the addLabel method is run on the window itself. The addLabel method creates a window component, a label object with the text "Hello world!," and adds it to the window at the grid position (0, 0).

4. The last five lines of code define a main function and check to see if the Python code file is being run as a program. If this is true, the main function is called to create an instance of the LabelDemo class. The mainloop method is then run on this object. At this point, the window pops up for viewing. Note that mainloop, as the name implies, enters a loop. The Python Virtual Machine runs this loop behind the scenes. Its purpose is to wait for user events, as mentioned earlier. The loop terminates when the user clicks the window's close box.

## A Template for All GUI Programs

```
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()
```

## Windows and Window Components

A window has several attributes. The most important ones are its

- **title (an empty string by default)**
- **width and height in pixels**
- **resizability (true by default)**
- **background color (white by default)**

With the exception of the window's title, the attributes of our label demo program's window have the default values. The background color is white and the window is resizable. The window's initial dimensions are automatically established by shrink-wrapping the window around the label contained in it.

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

The final way to change a window's attributes is to run a method included in the EasyFrame class. This class includes the four methods

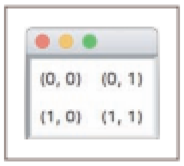
| EasyFrame Method | What It Does |
|---|---|
| `setBackground(color)` | Sets the window's background color to `color`. |
| `setResizable(aBoolean)` | Makes the window resizable (`True`) or not (`False`). |
| `setSize(width, height)` | Sets the window's width and height in pixels. |
| `setTitle(title)`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

**Window Layout**

Window components are laid out in the window's two-dimensional grid. The grid's rows and columns are numbered from the position (0, 0) in the upper left corner of the window. A window component's row and column position in the grid is specified when the component is added to the window.

```
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)
```

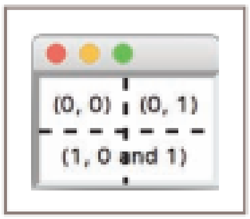


Each type of window component has a default alignment within its grid position. Because labels frequently appear to the left of data entry fields, their default alignment is northwest. The programmer can override the default alignment by including the sticky attribute as a keyword argument when the label is added to the window. The values of sticky are the strings "N," "S," "E," and "W," or any combination thereof. The next code segment centers the four labels in their grid positions:

```
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")
```

The programmer can force a horizontal and/ or vertical spanning of grid positions by supplying the rowspan and columnspan keyword arguments when adding a component (like merging cells in a table or spreadsheet). The spanning does not take effect unless the alignment of the component is centered along that dimension, however. The next code segment adds the three labels. The window's grid cells are outlined in the figure.

```
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)
```



**Types of Window Components and Their Attributes**

GUI programs use several types of **window components, or widgets** as they are commonly called. These include labels, entry fields, text areas, command buttons, drop-down menus, sliding scales, scrolling list boxes, canvases, and many others. The **breezypythongui** module includes methods for adding each type of window component to a window. Each such method uses the form

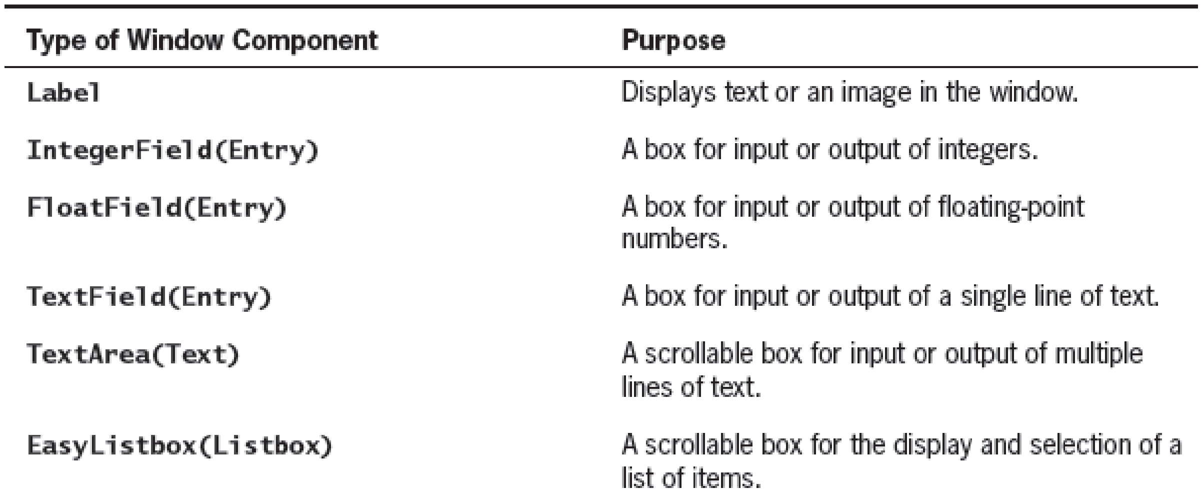
**self.addComponentType(<arguments>)**

When this method is called, **breeypythongui**

- 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

The window components supported by **breezypythongui** are either of the standard **tkinter** types, such as **Label, Button, and Scale**, or subclasses thereof, such as **FloatField, TextArea, and EasyCanvas.**



| Type of Window Component | Purpose |
|---|---|
| `Label` | Displays text or an image in the window. |
| `IntegerField(Entry)` | A box for input or output of integers. |
| `FloatField(Entry)` | A box for input or output of floating-point numbers. |
| `TextField(Entry)` | A box for input or output of a single line of text. |
| `TextArea(Text)` | A scrollable box for input or output of multiple lines of text. |
| `EasyListbox(Listbox)` | A scrollable box for the display and selection of a list of items. |

| Type of Window Component | Purpose |
|---|---|
| `Button` | A clickable command area. |
| `EasyCheckbutton(Checkbutton)` | A labeled checkbox. |
| `Radiobutton` | A labeled disc that, when selected, deselects related radio buttons. |
| `EasyRadiobuttonGroup(Frame)` | Organizes a set of radio buttons, allowing only one at a time to be selected. |
| `EasyMenuBar(Frame)` | Organizes a set of menus. |
| `EasyMenubutton(Menubutton)` | A menu of drop-down command options. |
| `EasyMenuItem` | An option in a drop-down menu. |
| `Scale` | A labeled slider bar for selecting a value from a range of values. |
| `EasyCanvas(Canvas)` | A rectangular area for drawing shapes or images. |
| `EasyPanel(Frame)` | A rectangular area with its own grid for organizing window components. |
| `EasyDialog(simpleDialog.Dialog)` | A resource for defining special-purpose popup windows. |

**Displaying Images**

Program adds two labels to the window. One label **displays the image** and the other label **displays the caption**. Unlike earlier examples, the program now keeps variable references to both labels for further processing. The image label is first added to the window with an empty text string. The program then creates a **PhotoImage** object from an image file and sets the image attribute of the image label to this object. Note that the variable used to hold the reference to the image must be an instance variable (prefixed by self), rather than a temporary variable. The image file must be in GIF format. Lastly, 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. The window is shrink-wrapped around the two labels and its dimensions are fixed.

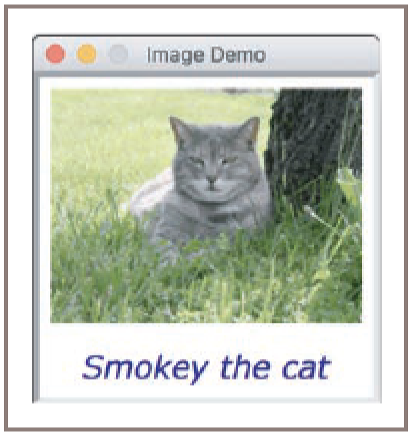
```
from breezypythongui import EasyFrame
from tkinter import PhotoImage
from tkinter.font import Font

class ImageDemo(EasyFrame):
    """Displays an image and a caption."""

    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"
```



**Command Buttons**

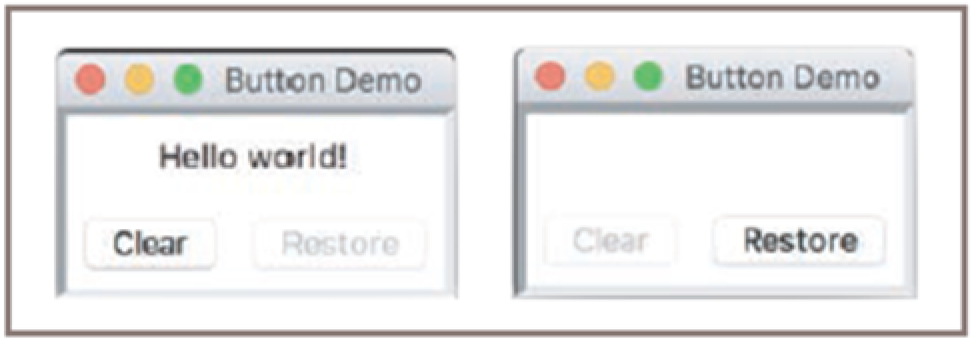
| Label Attribute | Type of Value |
|---|---|
| `image` | A `PhotoImage` object (imported from `tkinter.font`). Must be loaded from a GIF file. |
| `text` | A string. |
| `background` | A color. A label's background is the color of the rectangular area enclosing the text of the label. |
| `foreground` | A color. A label's foreground is the color of its text. |
| `font` | A `Font` object (imported from `tkinter.font`). |

Table 8-3 The `tkinter.Label` attributes

## Command Buttons

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.Button**. Like a label, a button can display an image, usually a small icon, instead of a string. A button also has a state attribute, which can be set to "normal" to enable the button (its default state) or "disabled" to disable it.

GUI programmers often lay out a window and run the application to check its look and feel, before adding the code to respond to user events. The buttons allow the user to clear or restore the label. When the user clicks **Clear**, the label is erased, the **Clear** button is disabled, and the **Restore** button is enabled. When the user clicks **Restore**, the label is redisplayed, the **Restore** button is disabled, and the **Clear** button is enabled.



```
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")
```

To allow a program to respond to a button click, the programmer must set the button's command attribute. There are two ways to do this: either by supplying a keyword argument when the button is added to the window or, later, by assignment to the button's attribute dictionary. The value of the command attribute should be a method of no arguments, defined in the program's window class. The default value of this attribute is a method that does nothing.

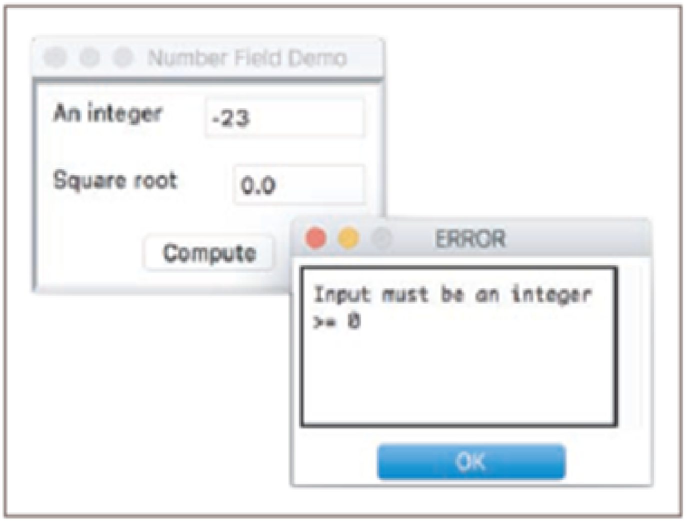
The completed version of the example program supplies two methods, which are commonly called **event handlers**, for the program's two buttons. Each of these methods resets the label to the appropriate string and then enables and disables the relevant buttons.

```
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, with event
                # handler methods supplied.
                self.clearBtn = self.addButton(text = "Clear", row = 1, column = 0, command =
                self.clear)
                self.restoreBtn = self.addButton(text = "Restore", row = 1, column = 1, state =
                "disabled", command = self.restore)
        # Methods to handle user events.
        def clear(self):
        """Resets the label to the empty string and updates the button states."""
                self.label["text"] = ""
                self.clearBtn["state"] = "disabled"
                self.restoreBtn["state"] = "normal"
        def restore(self):
        """Resets the label to 'Hello world!' and updates the button states."""
                self.label["text"] = "Hello world!"
                self.clearBtn["state"] = "normal"
                self.restoreBtn["state"] = "disabled"
```

**Pop-Up Message Boxes**

When errors arise in a GUI-based program, the program often responds by popping up a dialog window with an error message. Such errors are usually the result of invalid input data. The program detects the error, pops up the dialog to inform the user, and, when the user closes the dialog, continues to accept and check input data. In a terminal-based program, this process usually requires an explicit loop structure. In a GUI-based program, Python's implicit event-driven loop continues the process automatically.

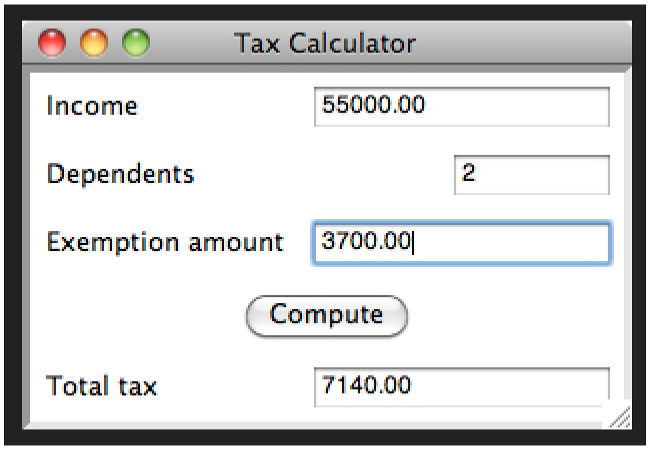


Input must be an integer >= 0

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

```
# 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")
```

**Sample program using breezypythongui**



The user can enter the three inputs in any order, and back out of an entry by editing it, before selecting the button to compute and display the result. Because the window stays alive after the Compute button is pressed, the user can subsequently adjust one input and leave the other previous inputs alone to compute and view a different result. Unlike terminal- and dialog-based applications, which guide the user through a determinate sequence of steps, a GUI-based application puts the user in charge of deciding which actions to perform. To give the kind of control to the user just mentioned, a GUI-based program must do four basic things:

1. Layout and pop up a window with the appropriate data fields, buttons, and other controls.
2. Wait for the user to perform an action, such as pressing a button or selecting text in a field.
3. Detect a user's action (also called an event).
4. Respond appropriately to each type of user action. We call this type of programming event-driven, because the program's behavior is driven by user events. The good news is that the runtime system handles the second and third of the four tasks – waiting for and detecting user events – mentioned above. The other two tasks, laying out the window and responding to user events, are a matter of defining several methods belonging to a window class. The GUI-based tax calculator program shows clearly where these two tasks are performed.

```
from breezypythongui import EasyFrame
class TaxCodeDemo(EasyFrame):
    """Application window for the tax calculator."""

    def __init__(self):
        """Sets up the window and the widgets."""
        EasyFrame.__init__(self, title = "Tax Calculator")
        # Label and field for the income
        self.addLabel(text = "Income", row = 0, column = 0)
        self.incomeField = self.addFloatField(value = 0.0,  row = 0, column = 1)
         # Label and field for the number of dependents
        self.addLabel(text = "Dependents",  row = 1, column = 0)
        self.depField = self.addIntegerField(value = 0, row = 1,  column = 1)
        # Label and field for the exemption amount
        self.addLabel(text = "Exemption amount",  row = 2, column = 0)
        self.exeField = self.addFloatField(value = 0.0,  row = 2, column = 1)
        # The command button
        self.addButton(text = "Compute", row = 3, column = 0, columnspan = 2,
                                                   command = self.computeTax)
         # Label and field for the tax
        self.addLabel(text = "Total tax", row = 4, column = 0)
        self.taxField = self.addFloatField(value = 0.0, row = 4, column = 1, precision = 2)
      # The event handler method for the button
    def computeTax(self):
         """Obtains the data from the input fields and uses  them to compute the tax, which is sent to
the output field."""
        income = self.incomeField.getNumber()
        numDependents = self.depField.getNumber()
        exemptionAmount = self.exeField.getNumber()
        tax = (income - numDependents * exemptionAmount) * .15
        self.taxField.setNumber(tax)
#Instantiate and pop up the window.
TaxCodeDemo().mainloop()
```

Reference for more details ::
https://lambertk.academic.wlu.edu/breezypythongui/tutorial-for-breezypythongui/