Programming in Python

Module 3

Rijin IK

Assistant Professor
Department of Computer Science and Engineering
Vimal Jyothi Engineering College
Chemperi

April 1, 2024

Outline

Simple Graphics using Turtle

- Turtle is a Python module that provides a drawing board like feature, which enables users to create pictures and shapes.
- The turtle is always at a specific spot, pinpointed by (x, y) coordinates.
- The coordinates use the standard Cartesian system, with (0, 0) at the screen's center, called the home.
- The turtle starts there, facing east at 0 degrees, and turns left to increase the degrees. For example, 90 degrees is north.

Simple Graphics using Turtle

Importing the turtle library:

- Import the turtle library using the following command:

```
import turtle
```

- Alternatively, you can import all objects from the turtle module:

```
from turtle import *
```

- or import the Turtle class specifically:

```
from turtle import Turtle
```

Creating an Object

```
from turtle import Turtle

t = Turtle()
```

Some attributes of a turtle

- In Python, every data value is an object, and these objects belong to classes.
- A class includes methods, which are operations applicable to objects of that class. Since a turtle is an object, its operations are methods too.
- Below are some methods belonging to the Turtle class, where the variable t refers to a specific Turtle object:

Turtle Graphics Attributes

- **Creates a new Turtle object**

```
t = Turtle()  #Creates a new Turtle object and opens its window.
```

- **Heading:**
  - Specifies turtle direction in degrees.
  - Increases counterclockwise.
  - Example:

```
t.setheading(45)  #sets the turtle's direction to 45 degrees.(0
     degrees is east, 90 degrees is north, etc.)
```

- **Color:**
  - Initially black, customizable.
  - Example:

```
t.pencolor("red")  #changes the pen color to red.
```

Turtle Graphics Attributes

- **Width:**
  - Width of the drawn line in pixels.
  - Initial width is 1 pixel.
  - Example:

```
t.width(3)  #sets the line width to 3 pixels.
```

- **Down:**
  - Attribute: true or false.
  - True (pen down): draws a line while moving.
  - False (pen up): moves without drawing.
  - Example:

```
t.down()  #makes the turtle draw . (Lowers turtle's pen to the
    drawing surface.)
```

```
t.up() #makes it move without drawing.(Raises turtle's pen from
     the drawing surface.)
```

Turtle Graphics Methods

- `t.home()`
  Moves `t` to the center of the window and points it east.
- `t.left(degrees)`
  Rotates `t` to the left by the specified degrees.
- `t.right(degrees)`
  Rotates `t` to 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)`
  Changes the pen color of `t` to the specified RGB value.

Drawing a Square with Turtle

- To draw a square using a Turtle, we have a helpful function called drawSquare.
- **Parameters:**
  - t: Turtle object (our drawing tool).
  - x, y: Pair of numbers representing the square's upper-left corner.
  - length: A single number indicating the length of each side.

Simple Graphics using Turtle

Steps:

1. Lift the Pen:
   - t.up(): Imagine lifting the pen from the paper so we can move without drawing.
2. Move to Corner:
   - t.goto(x, y): Shift the Turtle to the starting point (upper-left corner) of the square.
3. Set Direction:
   - t.setheading(270): Point the Turtle south, like the bottom of a map (270 degrees).
4. Lower the Pen:
   - t.down(): Place the pen back on the paper; we're ready to draw!

- Draw the Square:
  5. For each side:
     - Move forward by the given length: t.forward(length)
     - Turn left by 90 degrees: t.left(90)

Simple Graphics using Turtle

```
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()              # Lifts the pen
    t.goto(x, y)        # Moves to the specified corner point
    t.setheading(270)   # Points the turtle due south
    t.down()            # Puts the pen down for drawing
    for count in range(4):
        t.forward(length)
        t.left(90)
```

Setting Up Turtle Graphics in IDLE

- Before you start drawing with Python's turtle module in IDLE, let's set up a configuration file named turtle.cfg for a personalized experience.
  1. Create a turtle.cfg file:
     - Open a text editor.
     - Enter the following settings:

```
width = 300
height = 200
using_IDLE = True
colormode = 255
```

     - Save the file as turtle.cfg in your working directory.
  2. Launch IDLE:
     - Open IDLE, Python's Integrated Development and Learning Environment.
  3. Import the Turtle Class:

```
from turtle import Turtle
```

Setting Up Turtle Graphics in IDLE

4 Create a Turtle Object:

```
t = Turtle()
```

This line creates a Turtle object named t and opens a drawing window.

5 Drawing with Turtle:

- Now, let's have some fun with Turtle graphics. We'll draw the letter "T" in black and red.

```
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 the beginning of the 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
```

Setting Up Turtle Graphics in IDLE

6 Close the Drawing Window:

- To close the turtle's window, click its close box.

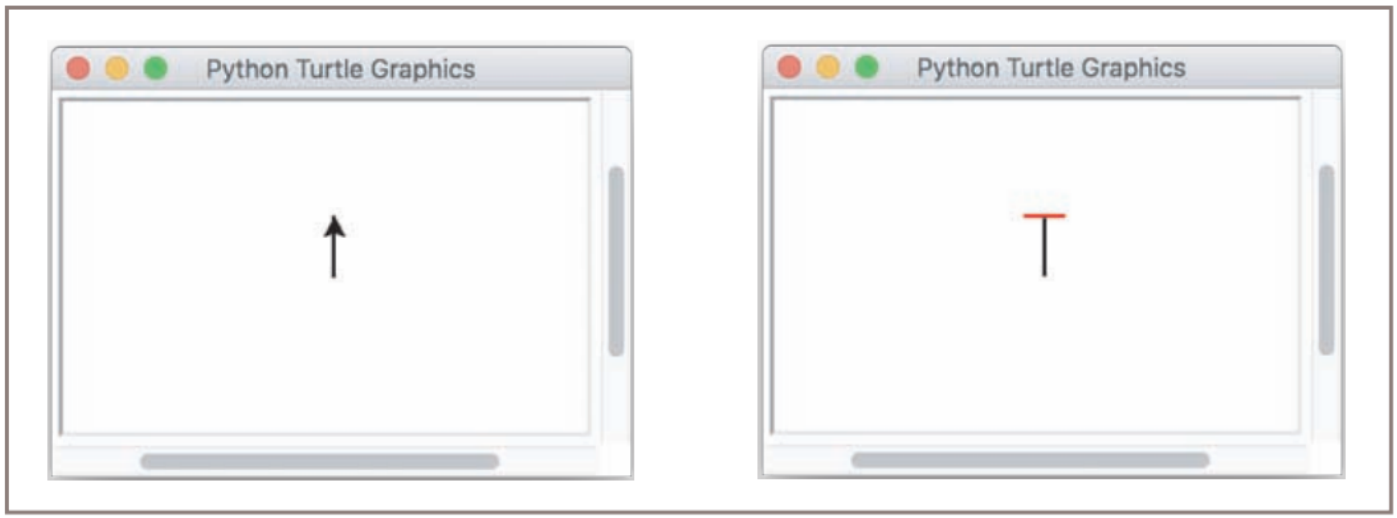

Figure: Drawing vertical and horizontal lines for the letter T

Drawing Two-Dimensional Shapes with Turtle Graphics

- In graphics applications, drawing simple two-dimensional shapes like squares and hexagons is common. Let's explore how to create more flexible functions for drawing these shapes using Python's turtle module.

Draw a Square:

- Here's a function square that draws a square with a given length. The square is drawn based on the turtle's current heading and position.

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

Drawing Two-Dimensional Shapes with Turtle Graphics

Draw a Hexagon:

- Similarly, we can create a function hexagon to draw a hexagon with the given length.

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

Draw Radial Hexagons:

- Now, let's draw a radial pattern of hexagons using the radialHexagons function.

```
def radialHexagons(t, n, length):
    """Draws a radial pattern of n hexagons with the given length.
    """
    for count in range(n):
        hexagon(t, length)
        t.left(360 / n)
```

More General Function - radialPattern:

- To make the functions more versatile, we create a more general function radialPattern.
- It can draw a radial pattern using any regular polygon by passing the corresponding shape function.

```
def radialPattern(t, n, length, shape):
    """Draws a radial pattern of n shapes with the given length."""
    for count in range(n):
        shape(t, length)
        t.left(360 / n)
```

Drawing Two-Dimensional Shapes with Turtle Graphics

```
from turtle import Turtle

t = Turtle()
t.pencolor("blue")
t.hideturtle()

square(t, 50)  # Embed a square in a hexagon
hexagon(t, 50)
t.clear()  # Erase all drawings

radialHexagons(t, 10, 50)  # Draw a radial pattern of hexagons
t.clear()  # Erase all drawings

radialPattern(t, 10, 50, square)  # Draw a radial pattern of squares
t.clear()  # Erase all drawings

radialPattern(t, 10, 50, hexagon)  # Draw a radial pattern of hexagons
```

Drawing Two-Dimensional Shapes - STAR

```
import turtle

# Create a turtle named star
star = turtle.Turtle()

# Adjust the initial orientation
star.right(75)

# Move forward to start drawing
star.forward(100)

# Draw the star
for i in range(4):
    star.right(144)
    star.forward(100)

# Finish the drawing
turtle.done()
```

Drawing Two-Dimensional Shapes - Parallelogram

```
import turtle

# Create a turtle named parallelogram
parallelogram = turtle.Turtle()

# Function to draw a parallelogram
def draw_parallelogram(turtle, side_length, angle):
    for _ in range(2):
        turtle.forward(side_length)
        turtle.left(angle)
        turtle.forward(side_length)
        turtle.left(180 - angle)

# Set the side length and angle for the parallelogram
side_length = 100
angle = 60  # Change this angle to modify the shape of the parallelogram
# Draw the parallelogram
draw_parallelogram(parallelogram, side_length, angle)
# Keep the window open until closed by the user
turtle.done()
```

Drawing Two-Dimensional Shapes -Circle

```
import turtle

# Initialize the turtle
t = turtle.Turtle()

# Draw a circle
t.circle(100)  # You can adjust the radius as needed

# Keep the window open until closed by the user
turtle.done()
```

Drawing Two-Dimensional Shapes -filled red circle

```
import turtle
# Create a turtle named pen
pen = turtle.Turtle()

# Set the fill color to red
pen.fillcolor("red")

# Begin filling the shape
pen.begin_fill()

# Draw a circle with a radius of 100 units
pen.circle(100)

# End the filling process
pen.end_fill()

# Hide the turtle
pen.hideturtle()
# Keep the window open until closed by the user
turtle.done()
```

Colors and the RGB System

- Pixels and Display:
  - A computer screen is composed of pixels (picture elements).
  - Pixel size affects the smoothness of lines, determined by display size and resolution.
  - Resolution example: 1680 pixels by 1050 pixels on a 20-inch monitor.
- Changing Pixel Colors:
  - Each pixel represents a color.
  - The Turtle graphics library allows changing colors using methods like pencolor.
  - Basic colors are available (e.g., black, red, yellow), but for a broader range, the RGB system is used.

Colors and the RGB System

RGB Color System:

- Components of RGB:
  - RGB stands for red, green, and blue, the primary color components.
  - Human retina sensitivity to these components.
  - Values range from 0 to 255 for each component, forming a unique color value
- Example Colors and RGB Values:
  - 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)
- Total RGB Color Values:
  - The RGB system allows 16,777,216 distinct color values (256 * 256 * 256).

Colors and the RGB System

Memory Representation:

- Bits and Color Representation:
  - N bits of memory can represent N$\hat{2}$ distinct data values.
  - In early displays, 1 bit represented black or white.
  - With 8 shades of gray, 3 bits were needed, and for 256 colors, 8 bits were required.
  - Each RGB color component requires 8 bits, totaling 24 bits for a distinct color value.

Colors and the RGB System

Introduction to Turtle Fill:

- The Turtle graphics library in Python provides methods for both drawing and filling shapes.
- Two essential methods for handling fill operations are fillcolor(), begin_fill(), and end_fill().

Basic Turtle Fill Operation:

- The process of filling a shape in Turtle involves the following steps:
  1. Define a Filling Color:
     - Use turtle.fillcolor() to set the color for filling.
  2. Begin the Filling Operation:
     - Initiate the fill operation with turtle.begin_fill().
  3. Perform Drawing:
     - Execute the turtle drawing commands within the fill operation.
  4. End the Filling Operation:
     - Conclude the fill operation with turtle.end_fill().

Colors and the RGB System

Example of Turtle Fill:

```
import turtle

# Set filling color
turtle.fillcolor("yellow")

# Begin fill operation
turtle.begin_fill()

# Draw a square
for _ in range(4):
    turtle.forward(100)
    turtle.left(90)

# End fill operation
turtle.end_fill()

# Close the turtle graphics window
turtle.done()
```

Example: Fill Square(orange)

```
import turtle

def square(t, length):
    t.fillcolor("orange")
    t.begin_fill()
    for _ in range(4):
        t.forward(length)
        t.left(90)
    t.end_fill()

# Set up the Turtle
t = turtle.Turtle()

# Call the square function with the desired length
square(t, 30)

# Close the turtle graphics window
turtle.done()
```

Example: Fill Square(red)

```
import turtle

def square(t, length):
    t.fillcolor("#ff0000")
    t.begin_fill()
    for _ in range(4):
    t.forward(length)
    t.left(90)
    t.end_fill()

# Set up the Turtle
t = turtle.Turtle()

# Call the square function with the desired length
square(t, 30)

# Close the turtle graphics window
turtle.done()
```

Example: Filling Radial Patterns with Random Colors

```
"""File: randompatterns.py
Draws a radial pattern of squares in a random fill color at each corner
    of the 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()
    r=(random.randint(0,255),random.randint(0,255),random.randint(0,255)
    )
    t.fillcolor("#%02x%02x%02x" % r)
    radialPattern(t, count, length, shape)
    t.end_fill()
```

Colors and the RGB System

```
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
```

Colors and the RGB System

```
    # 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)
    # Draw hexagons in lower-right corner
    drawPattern(t, width - inset, inset - height, count,length, hexagon)
if __name__ == "__main__":
    main()
```

Colors and the RGB System

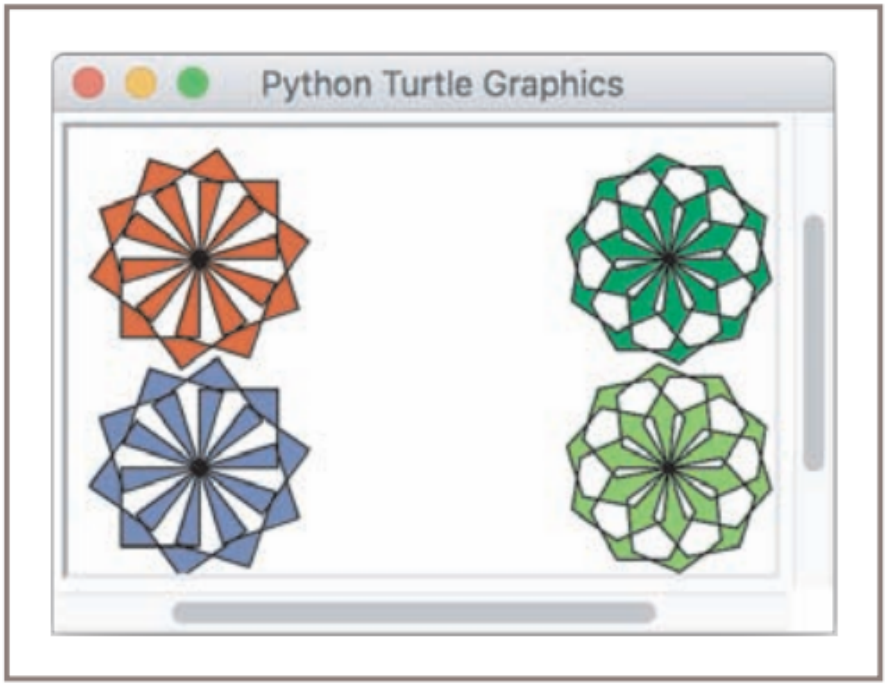

Figure: Radial patterns with random fill colors

Image Processing

- This big field involves the rules and methods for:
  - Taking pictures using things like flatbed scanners and digital cameras
  - Saving and showing pictures in good file formats
  - Making the special steps in programs like Adobe Photoshop to change pictures.

Analog and Digital Information

- Putting photos into a computer is tricky because computers use specific digital values, like numbers and bits, while real-world information, like images and sounds, is smooth and continuous.
- Think of a digital clock showing numbers compared to an analog clock with hands moving smoothly. Devices early on, like vinyl records, used analog wave patterns.
- To get real-world scenes into a computer, we need to convert the smooth information into specific values through a process called sampling.

Sampling and Digitizing Images

- Sampling devices measure discrete color values at distinct points on a two-dimensional grid. These values are pixels.
- In theory, the more pixels that are sampled, the more continuous and realistic the resulting image will appear.

Image File Formats

- Two of the most popular image file formats are JPEG (Joint Photographic Experts Group) and GIF (Graphic Interchange Format).

Image-Manipulation Operations

- Image-manipulation programs perform operations that either change the information within the pixels or rearrange the pixels in the image.
- These operations include
  - 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

The Properties of Images

- When an image is shown in a program like a web browser, the software organizes the bits into a colored pixel area on the screen.
- The pixel coordinates go from (0, 0) at the top-left to (width – 1, height – 1) at the bottom-right, where width and height are the image's size.
- This screen coordinate system is different from the standard one used in Turtle graphics.
- In this system, the RGB color system is used to represent colors, with an image having width, height, and color values accessible using (x, y) coordinates.
- A color is represented by a tuple (r, g, b), where 'r,' 'g,' and 'b' are the integer values for red, green, and blue components.

The images Module

- The "images" module in Python is a tool for image processing.
- It allows users to load, view, manipulate, and save images.
- The module includes a class called Image, representing images as a grid of RGB values.
- To use this module, you can place the "images.py" file and sample images in your current working directory.
- The Image class provides various methods for handling images

Image Methods - Part 1

Method	Description
`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.

Image Methods - Part 2

Method	Description
`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.

Image Methods - Part 3

Method	Description
`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.

Working with the "images" Module in Python

- The "images" module is a Python tool for image processing.
- It specifically accepts image files in GIF format.

Loading and Displaying an Image:

- Import the Image class from the images module.
- Instantiate the class using a GIF image file (e.g., smokey.gif).
- Display the image using the draw() method.

Working with the "images" Module in Python

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

Figure: An image display window

Working with the "images" Module in Python

Handling Exceptions:

- Python raises exceptions if the file is not found or not in GIF format.
- Users must close the display window to return control to the program.

Examining Image Properties:

- getWidth() and getHeight() methods provide image dimensions.
- The string representation of the image includes filename, width, and height.

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

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

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

Working with the "images" Module in Python

Pixel Information:

- Use getPixel(x, y) to retrieve RGB values at specific coordinates.
- Example: image.getPixel(0, 0) returns (194, 221, 114).

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

Creating a Blank Image:

- *Image*(*width*, *height*) creates a new image with transparent pixels.
- Useful for backgrounds or receiving information from other images.

Modifying Pixel Values:

- setPixel(x, y, (r, g, b)) replaces RGB values at a given position.
- Example: Creating blue lines in a 150x150 image.

Working with the "images" Module in Python

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

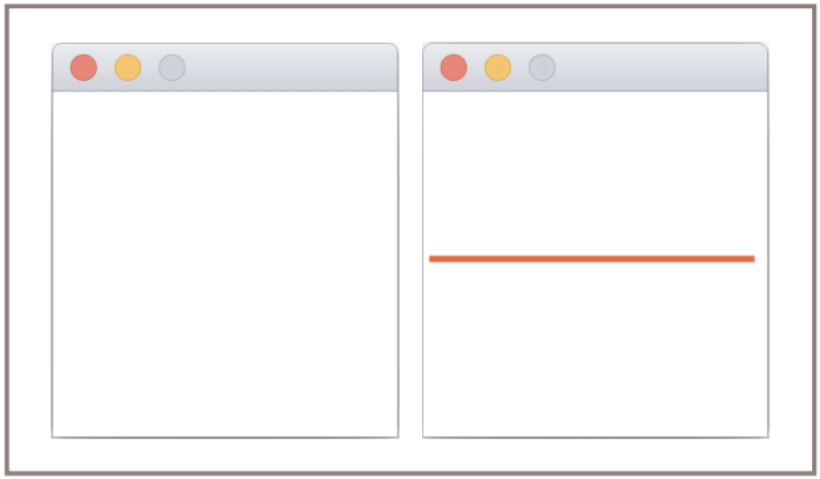

Figure: An image before and after replacing the pixels

Working with the "images" Module in Python

Saving Images:

- save() writes an image back to the current filename.
- save(filename) writes to a new file; automatically adds .gif extension.

```
>>> image.save("horizontal.gif")
```

Loop Pattern for Grid Traversal in Image Processing

- In image processing, algorithms often require traversing a two-dimensional grid of pixels.
- Unlike linear loops, these operations employ a nested loop structure, consisting of an outer loop and an inner loop.
- The outer loop iterates over one coordinate, while the inner loop iterates over the other.

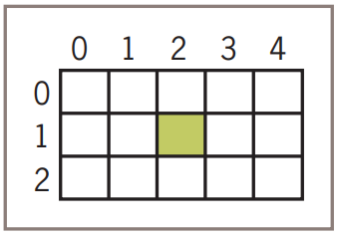

Figure: A grid with 3 rows and 5 columns

- Consider a grid with a height of 3 rows and a width of 5 columns, as shown in Figure . Each pixel in the grid is accessed using a pair of coordinates $(< column >, < row >)$. For example, the pixel at the middle of the grid is at position (2, 1).

Working with the "images" Module in Python

Here's an example of a nested loop structure that traverses the y coordinates of the grid:

```
width = 2
height = 3
for y in range(height):
    for x in range(width):
        print((x, y), end=" ")
    print()
```

This row-major traversal prints pairs of coordinates as the loop moves across rows and columns in the grid.

Working with the "images" Module in Python

The row-major traversal template is as follows:

```
for y in range(height):
    for x in range(width):
        # Do something at position (x, y)
```

In practical use, this pattern is applied to various algorithms. For instance, the following code segment utilizes a nested loop to fill a blank image with red pixels:

```
image = Image(150, 150)
for y in range(image.getHeight()):
    for x in range(image.getWidth()):
        image.setPixel(x, y, (255, 0, 0))
```

This nested loop structure is a fundamental pattern in image processing algorithms, facilitating operations on individual pixels within an image grid.

Understanding Tuples in Image Processing

- To brighten a pixel by increasing its RGB values by 10 in image processing algorithms, you can easily retrieve the pixel's tuple using the getPixel function and manipulate the RGB values by assigning them a new tuple in Python.

```
image = Image("smokey.gif")
(r, g, b) = image.getPixel(0, 0)
```

Now, you can inspect the RGB values:

```
print(r)  # 194
print(g)  # 221
print(b)  # 114
```

Understanding Tuples in Image Processing

- To complete the task, create a new tuple with the updated values and set the pixel to that tuple:

```
image.setPixel(0, 0, (r + 10, g + 10, b + 10))
```

- The pattern (r, g, b) can be used in various contexts, except when defining function parameters.
- In function parameters, components must be extracted within the function's body. For example, the average function computes the average of a triple:

```
def average(triple):
    (a, b, c) = triple
    return (a + b + c) // 3

average((40, 50, 60))  # Output: 50
```

Understanding Tuples in Image Processing

- These basic operations set the stage for exploring simple image-processing algorithms.
- These algorithms can modify the color of each pixel or use pixel information to create a new image.
- Each algorithm is represented as a Python function, taking an image as an argument.
- Some functions return a new image, while others modify the input image for consistency and ease of use.

Converting an Image to Black and White

Converting an Image to Black and White

- The blackAndWhite function in the code turns an image into black and white.
- It goes through each pixel, computes the average RGB value, and changes the pixel to black (0, 0, 0) if the average is closer to 0, or to white (255, 255, 255) if the average is closer to 255.

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)
```

Converting an Image to Black and White

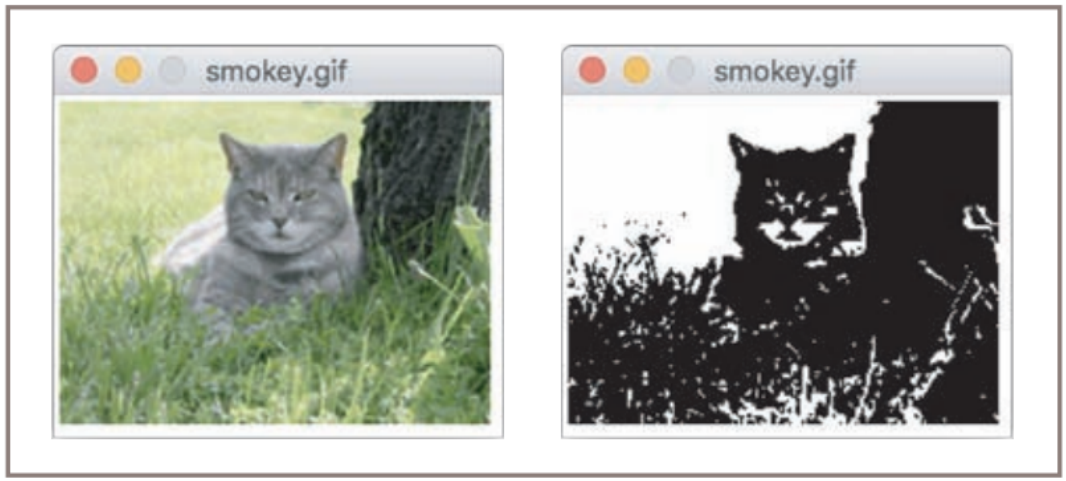

Figure: Converting a color image to black and white

Converting an Image to Grayscale

Converting an Image to Grayscale

- The grayscale function in the provided code converts a color image to grayscale.
- Grayscale images contain shades of gray, and this function calculates the grayscale values for each pixel by taking into account the sensitivity of the human eye to different color components (red, green, and blue).
- The function uses weighted averages based on the relative luminance proportions of these color components.

Converting an Image to Grayscale

```
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)

            # Weighted averages based on relative luminance proportions
            r = int(r * 0.299)
            g = int(g * 0.587)
            b = int(b * 0.114)

            # Calculate the overall luminance and set the pixel to
    grayscale
            lum = r + g + b
            image.setPixel(x, y, (lum, lum, lum))
```

Converting an Image to Grayscale

Figure: Converting a color image to grayscale

Copying an Image

Copying an Image

- To make a new image with the same size as the original and keep its pixel details, we can use the clone method in the Image class.
- This method generates a duplicate image, sharing the dimensions of the original, but having an empty filename.

Copying an Image

```
>>> from images import Image

>>> image = Image("old.gif")

# Display the original image
>>> image.draw()
# Create a copy of the original image using the 'clone' method and store
    it in 'newImage'
>>> newImage = image.clone()  # Create a copy of the image

# Display the copied image
>>> newImage.draw()

# save the copied image
>>> newImage.save("new.gif")
```

Blurring an Image

Blurring an Image

- Sometimes, images have rough edges, a condition known as pixelation.
- Blurring can help soften these areas, but it comes at the expense of losing some clarity.
- This algorithm smoothens an image by setting each pixel's color to the average of its four surrounding pixels.
- The blurring process starts from position (1, 1) and stops at (width - 2, height - 2), avoiding the pixels on the outer edges of the image.

Blurring an Image

```
def blur(image):
    """Builds and returns a new image that is a blurred copy of the
    argument image."""

    # Auxiliary function to sum two RGB tuples
    def tripleSum(triple1, triple2):
        (r1, g1, b1) = triple1
        (r2, g2, b2) = triple2
        return (r1 + r2, g1 + g2, b1 + b2)

    # Create a new image as a copy of the original
    new = image.clone()

    # Traverse the image grid, excluding outer edges
    for y in range(1, image.getHeight() - 1):
        for x in range(1, image.getWidth() - 1):
            oldP = image.getPixel(x, y)
```

Blurring an Image

```
            # Get neighboring pixels
            left = image.getPixel(x - 1, y)
            right = image.getPixel(x + 1, y)
            top = image.getPixel(x, y - 1)
            bottom = image.getPixel(x, y + 1)

            # Calculate sum of RGB values
            sums = reduce(tripleSum, [oldP, left, right, top, bottom])

            # Calculate averages by dividing sums by 5
            averages = tuple(map(lambda x: x // 5, sums))

            # Set the pixel in the new image with the calculated
    averages
            new.setPixel(x, y, averages)

    return new
```

Edge detection

- Edge detection does the opposite of adding colors to an image; it takes a color image and reveals only the outlines of objects.
- The detectEdges function needs an image and an integer as inputs.
- It produces a new black-and-white image that clearly displays the edges from the original image.
- The integer parameter lets users experiment with different luminance variations to control the edge detection.

Edge detection

- Edge detection does the opposite of adding colors to an image; it takes a color image and reveals only the outlines of objects.
- The detectEdges function needs an image and an integer as inputs.
- It produces a new black-and-white image that clearly displays the edges from the original image.
- The integer parameter lets users experiment with different luminance variations to control the edge detection.

Edge detection

```
def detectEdges(image, amount):
    """Returns a new image with highlighted edges in black and white."""

    # Helper function to calculate average luminance
    def average(triple):
        (r,g,b) = triple
        return (r+g+b) // 3

    black_pixel = (0, 0, 0)
    white_pixel = (255, 255, 255)

    # Create a new image as a copy of the original
    new_image = image.clone()

    # Iterate through the image pixels
    for y in range(image.getHeight() - 1):
        for x in range(1, image.getWidth()):
            old_pixel = image.getPixel(x, y)
            left_pixel = image.getPixel(x - 1, y)
            bottom_pixel = image.getPixel(x, y + 1)
```

Edge detection

```
            old_lum = average(old_pixel)
            left_lum = average(left_pixel)
            bottom_lum = average(bottom_pixel)

            # Check for significant differences in luminance
            if abs(old_lum - left_lum) > amount or abs(old_lum -
    bottom_lum) > amount:
                new_image.setPixel(x, y, black_pixel)
            else:
                new_image.setPixel(x, y, white_pixel)

return new_image
```

Edge detection

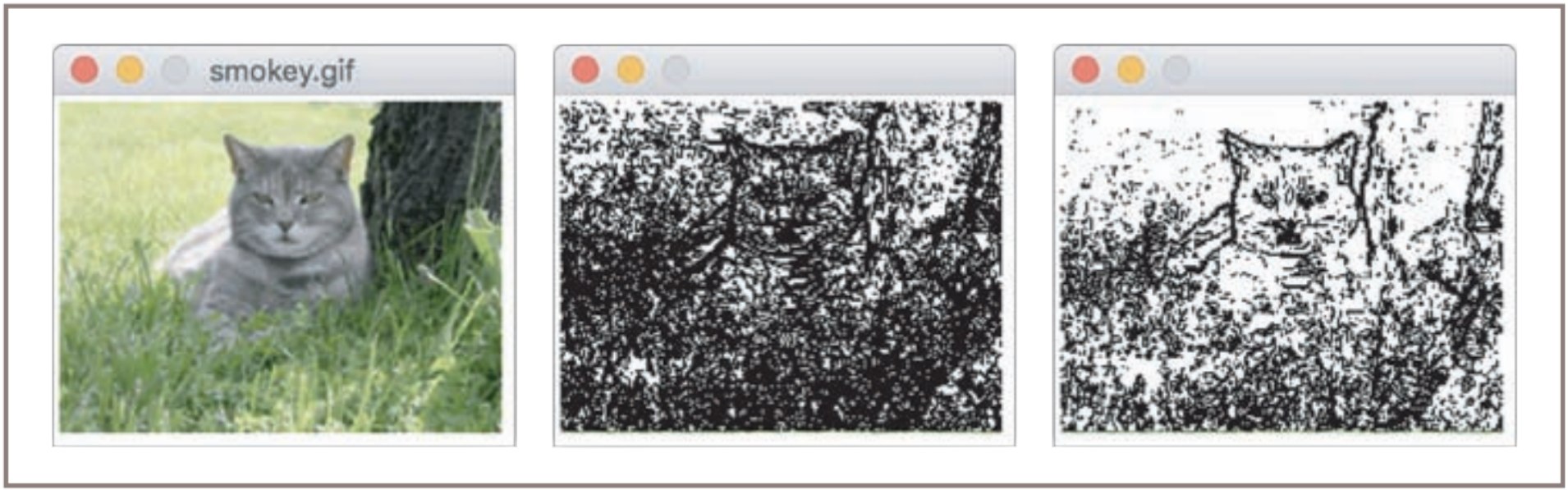

Figure: Edge detection: the original image, a luminance threshold of 10, and a luminance threshold of 20

Reducing the Image Size

- Reducing the size of an image on a display, like a computer monitor, depends on its pixel width and height as well as the display's resolution, measured in dots per inch (DPI).
- The typical approach maintains the image's aspect ratio (width to height ratio) during reduction. To shrink an image, a new one is created with dimensions that are a constant fraction of the original's width and height.
- The function for this task takes the original image and a positive integer shrinkage factor as inputs.
- A shrinkage factor of 2, for instance, means the image is reduced to half its original size.
- The algorithm calculates the dimensions of the new image based on the shrinkage factor and then creates it accordingly.

Reducing the Image Size

```
def shrink(image, factor):
    """Returns a new smaller image by a specified factor."""

    # Get the original image dimensions
    width = image.getWidth()
    height = image.getHeight()

    # Create a new image with reduced dimensions
    new = Image(width // factor, height // factor)

    oldY = 0
    newY = 0

    # Iterate through rows
    while oldY < height - factor:
        oldX = 0
        newX = 0
```

Reducing the Image Size

```
            # Iterate through columns
            while oldX < width - factor:
                old_pixel = image.getPixel(oldX, oldY)
                new.setPixel(newX, newY, old_pixel)

                oldX += factor
                newX += 1

            oldY += factor
            newY += 1

    return new
```

Reducing the Image Size

- The function creates a smaller image based on a specified factor.
- It gets the original image dimensions.
- A new image with reduced dimensions is created.
- The function iterates through rows and columns, copying pixels from the original to the new image.
- The result is a smaller image with dimensions reduced by the specified factor.

Python Imaging Library (PIL)

- Python Imaging Library (PIL), now known as Pillow, is a Python library used for opening, manipulating, and saving many different image file formats.
- It provides capabilities for image processing tasks such as resizing, cropping, rotating, enhancing, and filtering images.
- It supports a wide range of image file formats including JPEG, PNG, GIF, BMP, and TIFF, making it a powerful tool for working with images in Python applications.

Python Imaging Library (PIL)

```
from PIL import Image

# Open the image file
img = Image.open("rose.png")

# Display the opened image
img.show()
```

- This code directly opens the "rose.png" image file using Image.open() method from Pillow.
- it displays the image using the show() method.

Python Imaging Library (PIL)

To create a duplicate copy of an image using Pillow (PIL), you can use the copy() method.

```
from PIL import Image

# Open the original image
original_image = Image.open("original_image.jpg")

# Create a duplicate copy of the original image
duplicate_image = original_image.copy()

# Save the duplicate image
duplicate_image.save("duplicate_image.jpg")

# Optionally, you can also show the duplicate image
duplicate_image.show()
```

Graphical User Interfaces

- Graphical User Interfaces (GUIs) are commonly used in computer software, displaying text and icons for user interaction.
- Users can click icons using a mouse to perform actions.
- GUIs allow direct manipulation of information with a pointing device.

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

- In contrast, terminal-based programs prompt users for inputs in a specific order.
- GUIs are event-driven, waiting for user actions like button clicks, while terminal programs maintain constant control, guiding users through successive inputs.

The Terminal-Based Version

- For instance, a terminal-based tax program prompts users for income and dependents, calculates tax, and displays the result, ending the program.
- Users must follow a fixed sequence of prompts, unable to go back or change inputs.
- To get results for different data, the program must be run again with all inputs re-entered.

Graphical User Interfaces

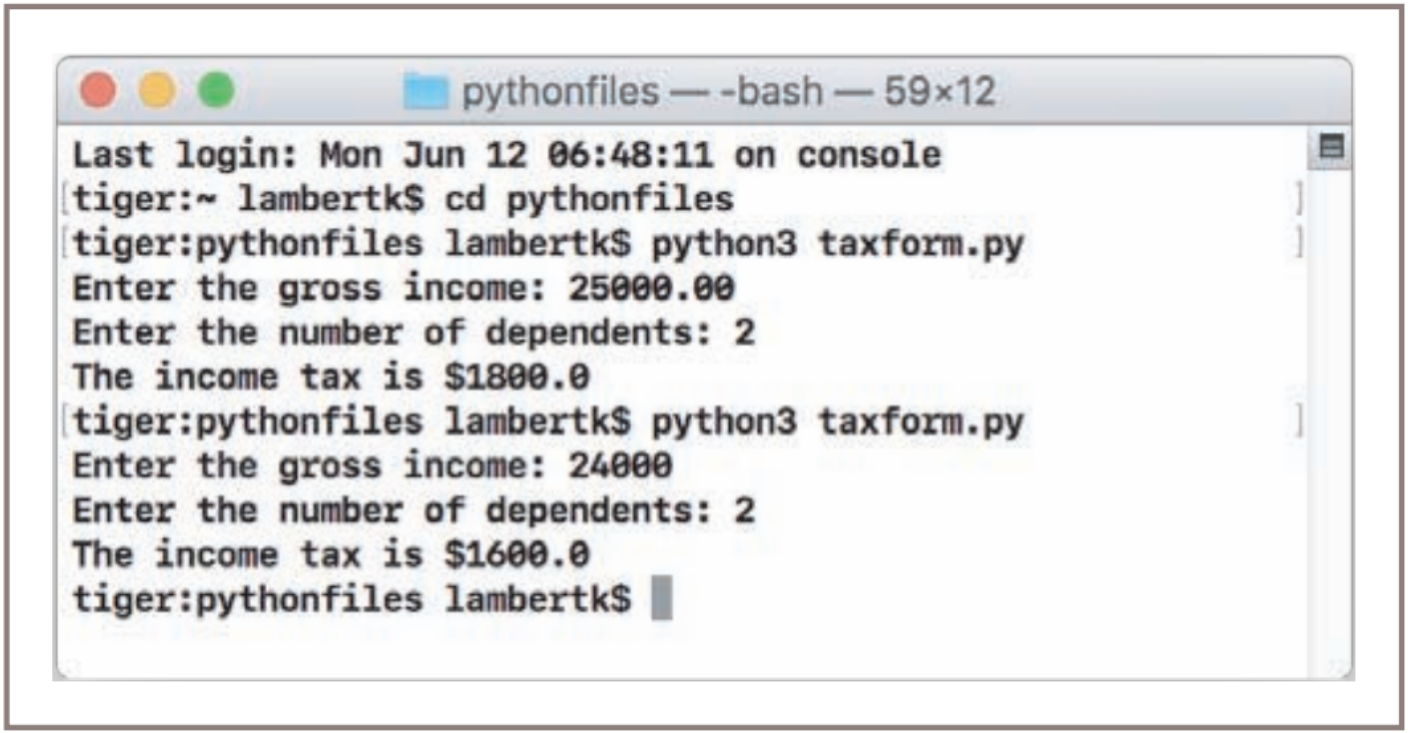

Figure: A session with the terminal-based tax calculator program

The GUI-Based Version

- The GUI version of the program has a window with different parts like a title bar, buttons, labels, and entry fields.
- Users can easily interact with the program by clicking buttons and entering data.
- Unlike the terminal version, users can edit inputs before calculating and don't need to re-enter all data for different results.
- This makes the GUI a clear improvement, especially as the program's complexity increases.

Graphical User Interfaces

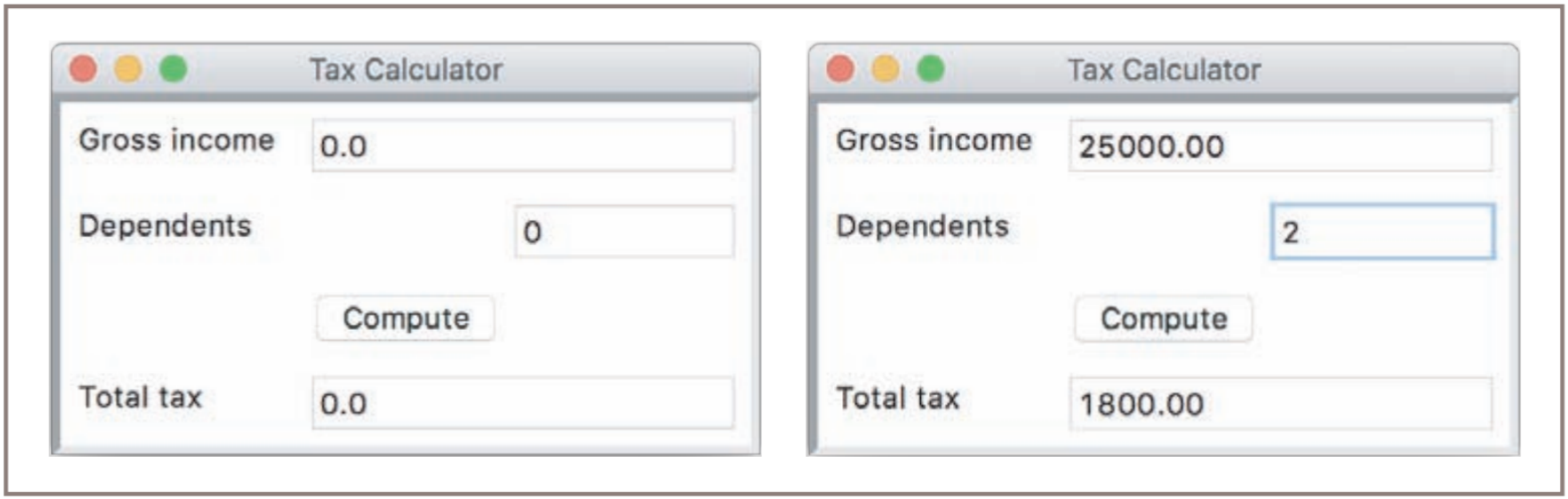

Figure: A GUI-based tax calculator program

Event-Driven Programming

- In a GUI program, a window opens and waits for users to use the mouse.
- When users click, like with a mouse click, the program responds by taking in inputs, processing them, and showing results. This type of software is called event-driven, where actions from users trigger responses.
- In the analysis phase, you figure out the types and arrangement of window components, often using existing classes or creating new ones.

Event-Driven Programming

- Taking the tax calculator example, the GUI has labeled entry fields and a "Compute" button. Clicking the button triggers a method that calculates and displays results. Once interactions are planned, coding begins.
- 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

- For coding simple GUI-based programs in Python, we uses an open-source module called breezypythongui instead of the standard tkinter module, which might be challenging for beginners.
- Breezypythongui simplifies the process, occasionally drawing on some basic features of tkinter.

Coding simple GUI-based programs

1. Import the EasyFrame class:
   - Import the necessary class from the module.

```
from breezypythongui import EasyFrame
```

   - This class helps create GUI windows and is a simpler way to work with Tkinter's Frame class.
2. Define the LabelDemo class:
   - Create a new class called LabelDemo that inherits from EasyFrame.

```
class LabelDemo(EasyFrame):
```

   - This class will describe the layout and functionality of the window.

Coding simple GUI-based programs

3 Define the _init_ method in LabelDemo:
   - Set up the initialization method that automatically runs when the window is created.

```
def __init__(self):
    EasyFrame.__init__(self)
    self.addLabel(text="Hello world!", row=0, column=0)
```

   - Call the superclass's init method (EasyFrame._init_) and then add a label to the window at position (0, 0).

4 Define the main function and check if run as a program:
   - Create a main function that creates an instance of LabelDemo and displays the window.

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

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

   - If the script is run directly (not imported), it creates an instance of LabelDemo and runs its mainloop method, which displays the window.

Coding simple GUI-based programs

```
"""
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()
```

Coding simple GUI-based programs

- The mainloop() function in Tkinter or similar GUI libraries runs a continuous loop that listens for user actions like mouse clicks or keyboard inputs, ensuring the interface stays interactive and responsive while updating according to user interactions.
- we use if __name__ == "__main__": to make sure that certain code only runs when we run the script directly, not when we import it into another script. It helps keep our code organized and prevents unintended execution of certain parts of the script.

Coding simple GUI-based programs

Figure: Displaying a label with text in a window

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()
```

A Template for All GUI Programs

- A GUI application window is represented as a class extending EasyFrame, and its __init__ method initializes the window with attributes and GUI components.
- The event handling methods respond to user events, and the main function creates an instance of the window class, running the mainloop method to display the window. Closing the window exits the program.
- In IDLE, you can rerun the program by entering main() at the shell prompt.

The Syntax of Class and Method Definitions

- In Python, when you create a class or define a method:
- Class Definition Syntax:

```
class ClassName:
    # Class body with method definitions and other code
```

  - The class name is conventionally capitalized.
  - The class body, indented under the header, contains method definitions and other class-related code.

Method Definition Syntax:

- The method header is similar to a function header.

```
def methodName(self, other_parameters):
    # Method body with code
```

  - The method always has at least one parameter named self in the first position.
  - The self parameter represents the instance of the object on which the method is called. When calling a method, Python automatically assigns the object to the self parameter.

The Syntax of Class and Method Definitions

- In this example, obj.myMethod(5) automatically assigns obj to the self parameter inside myMethod.
- The method then uses self to perform operations on the object.

```
class MyClass:
    def myMethod(self, x):
        # Method body accessing the instance with self
        result = x + 1
        return result

# Creating an instance of the class
obj = MyClass()

# Calling the method on the object
result_value = obj.myMethod(5)
```

Subclassing and Inheritance as Abstraction Mechanisms

- When creating a new type of object, like a window, we can use existing classes to save effort.
- In our program, the LabelDemo class is designed for specific tasks.
- It inherits capabilities from the EasyFrame class, which itself is a kind of window.
- This approach, known as subclassing and inheritance, allows us to reuse and customize existing classes, making it easier to create new classes with specific functionalities.

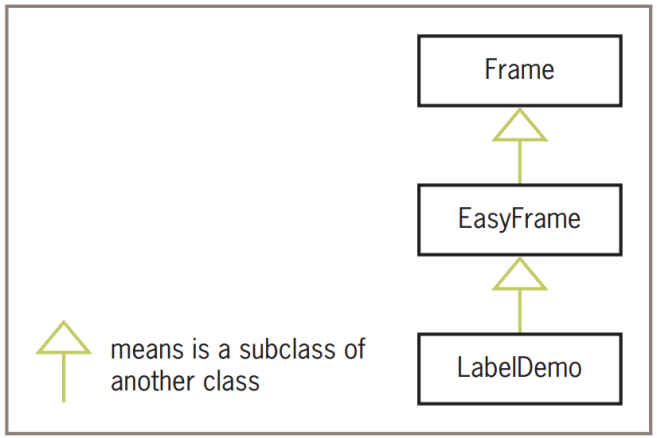

Figure: A class diagram for the label demo program

Windows and Window Components

- This section discusses the details of windows and their attributes in GUI programming. Windows possess key attributes such as title, width, height, resizability, and background color.
- The default values include an empty string for the title, white background, and resizable windows.
- In the label demo program, the window's attributes, aside from the title, maintain their default values.
- To customize the title, width, and height, optional arguments can be supplied in the EasyFrame method.
- For instance:

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

Windows and Window Components

- Alternatively, attributes can be modified by accessing the window's attribute dictionary.
- Each window or component maintains a dictionary of attributes and values, which can be accessed or modified using standard subscript notation.
- In the label demo's __init__ method, the background color can be set to yellow:

```
self["background"] = "yellow"
```

- t's important to note that self in this context refers to the window itself.
- Another method to change window attributes is by using predefined methods provided by the EasyFrame class, including four methods listed in Table.

Windows and Window Components

Method	Description
`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`.

- Using

  `self.setResizable(False)`

  in your code prevents the window from being resized by the user.

Window Layout

- Window components in GUI applications are arranged in a two-dimensional grid.
- Rows and columns of the grid are numbered from (0, 0) in the upper left corner of the window.
- Example Program (layoutdemo.py):

```
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)
```

  - Labels are added to the window with specific row and column positions.
  - The window is shrink-wrapped around the labels initially.

Window Layout

Default Alignment:

- Each window component has a default alignment within its grid position.
- Labels default to northwest alignment.
- Custom Alignment:

```
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 **sticky** attribute is used to customize alignment.
  - Values: "N," "S," "E," "W," or combinations.

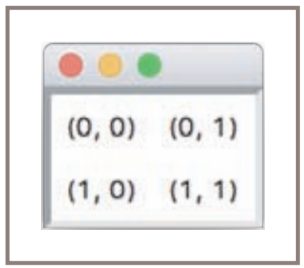

Figure: Laying out labels in the window's grid

Spanning 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 and 1)", row=1, column=0, sticky="NSEW",
    columnspan=2)
```

- Horizontal and/or vertical spanning using columnspan and rowspan.
- Spanning works with centered alignment.

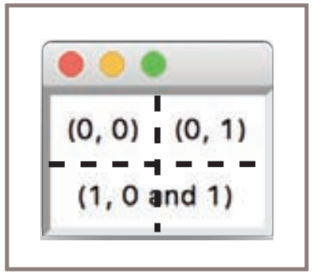

Figure: Labels with center alignment and a column span of 2

Types of Window Components and Their Attributes

- In GUI programs, we use various types of window components, often called widgets.
- These include labels, entry fields, text areas, buttons, drop-down menus, sliders, list boxes, canvases, and more.
- The **breezypythongui** module provides methods for adding each type of component to a window. When you use these methods, the module:
  - Creates the specific window component requested.
  - Sets the component's attributes with default or provided values.
  - Positions the component in the window's grid (using required row and column arguments).
  - Gives you a reference to the created component.
- The supported components in breezypythongui are either standard tkinter types like Label, Button, and Scale, or subclasses like FloatField, TextArea, and EasyCanvas.

Frame 2: Basic Window Components

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.
Button	A clickable command area.
EasyCheckbutton(Checkbutton)	A labeled checkbox.

Frame 3: Additional Window Components (Part 1)

Type of Window Component	**Purpose**
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.

Frame 4: Additional Window Components (Part 2)

Type of Window Component	**Purpose**
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.

Types of Window Components and Their Attributes

```
# Import necessary modules
from breezypythongui import EasyFrame

# Define a class that extends EasyFrame
class MyGUI(EasyFrame):
    def __init__(self):
        EasyFrame.__init__(self, title="My GUI")

        # Add a label to the window
        self.addLabel(text="Hello, BreezyPythonGUI!", row=0, column=0)

        # Add a button to the window
        self.addButton(text="Click me", row=1, column=0, command=self.
    buttonClicked)
```

Types of Window Components and Their Attributes

```
    def buttonClicked(self):
        # Define the action when the button is clicked
        self.messageBox(title="Button Clicked", message="You clicked the
     button!")

# Instantiate and run the GUI
if __name__ == "__main__":
    MyGUI().mainloop()
```

Displaying Images in a GUI (ImageDemo Program)

- The goal is to illustrate the use of attribute options for a label component within a GUI program named "ImageDemo.py."
- The program window consists of two labels: one for displaying an image and the other for showing a caption.

Components of the Program:

- Window Setup:
  - The program extends the EasyFrame class from the breezypythongui module.
  - The window is titled "Image Demo."
  - It is set to be non-resizable.

Label Components:

- Two labels are added to the window:
  - imageLabel for displaying the image.
  - textLabel for showing the caption ("Smokey the cat").

Displaying Images in a GUI (ImageDemo Program)

Image Display:

- imageLabel initially has an empty text string.
- The program creates a PhotoImage object from an image file ("smokey.gif").
- The image attribute of imageLabel is set to this PhotoImage object.
- Note: The variable holding the reference to the image must be an instance variable (prefixed by self).

Caption Formatting:

- A Font object is created with a non-standard font (family: "Verdana," size: 20, slant: "italic").
- The font and color attributes of textLabel are modified to achieve the desired caption appearance (blue color).

Window Dimensions:

- Two labels are added to the window:
  - The window is "shrink-wrapped" around the two labels.
  - Its dimensions are fixed based on the content.

Displaying Images in a GUI (ImageDemo Program)

```
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)

        # Add an empty label for the image
        imageLabel = self.addLabel(text="", row=0, column=0, sticky="
    NSEW")

        # Add a label for the caption
        textLabel = self.addLabel(text="Smokey the cat", row=1, column
    =0, sticky="NSEW")
```

Displaying Images in a GUI (ImageDemo Program)

```
# 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"
```

Figure: Displaying a captioned image

Displaying Images in a GUI (ImageDemo Program)

Additional Information:

- The image file must be in GIF format for the program to function correctly.
- Refer to Table for a summary of the tkinter.Label attributes used in this program.

Attribute	**Type of Value**
image	A PhotoImage object (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 (from tkinter.font).

Command Buttons and Responding to Events

- To create a button in a window, use the addButton method and specify its text and position in the grid.
- A button is initially centered. It can display an image instead of text and has a state, set to "normal" for enabled (default) or "disabled.
- " GUI designers often set up the window first to check its appearance before adding code to respond to user clicks. The buttons in this application clear or restore a label.
- Clicking "Clear" erases the label, disables the Clear button, and enables the Restore button. Clicking "Restore" redisplay the label, disables Restore, and enables Clear.

Command Buttons and Responding to Events

Figure: Using command 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")
```

Command Buttons and Responding to Events

```
        # 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 make a program react when a button is clicked, the programmer needs to specify the button's command attribute.
- This can be done either by providing a keyword when adding the button to the window or by assigning a method to the button's attribute dictionary later on.
- The command attribute's value should be a method without any arguments, defined within the program's window class. By default, this attribute is set to a method that does nothing.

Command Buttons and Responding to Events

- In the finished program, there are two methods, often called event handlers, for the two buttons.
- These methods reset the label to the right text and then enable or disable the buttons accordingly.

```
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)
```

Command Buttons and Responding to Events

```
        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"
```

Input and Output with Entry Fields

- A text field is appropriate for entering or displayinga single-line string of characters.

Text Fields:

- Adding Text Field:
  - The addTextField method is used to add a text field to a window.
  - Syntax:

```
addTextField(text, row, column, ...optional_arguments...)
```

  - Returns a TextField object, which is a subclass of tkinter.Entry.
  - Example:

```
self.inputField = self.addTextField(text="", row=0, column=1)
```

Text Fields

Text Field Attributes:

- Default alignment: northeast of its grid cell.
- Default width: 20 characters.
- Set the state attribute to "readonly" to prevent user editing for output fields.
- Example:

```
self.outputField = self.addTextField(text="", row=1, column=1,
    state="readonly")
```

Text Field Methods:

- getText(): Returns the string currently contained in a text field.
- setText(result): Outputs the string argument to a text field.
- Example:

```
text = self.inputField.getText()
result = text.upper()
self.outputField.setText(result)
```

Text Fields

Example Program:

```
class TextFieldDemo(EasyFrame):
    def __init__(self):
        EasyFrame.__init__(self, title="Text Field Demo")
        self.addLabel(text="Input", row=0, column=0)
        self.inputField = self.addTextField(text="", row=0, column=1)
        self.addLabel(text="Output", row=1, column=0)
        self.outputField = self.addTextField(text="", row=1, column=1,
    state="readonly")
        self.addButton(text="Convert", row=2, column=0, columnspan=2,
    command=self.convert)

    def convert(self):
        text = self.inputField.getText()
        result = text.upper()
        self.outputField.setText(result)
```

Text Fields

Figure: Using text fields for input and output

Numeric Fields:

- Adding Integer and Float Fields:
- addIntegerField and addFloatField methods for integers and floating-point numbers, respectively.
- Syntax:

```
addIntegerField(value, row, column, ...optional_arguments...)
```

- Example:

```
self.inputField = self.addIntegerField(value=0, row=0, column=1,
    width=10)
self.outputField = self.addFloatField(value=0.0, row=1, column=1,
    width=8, precision=2, state="readonly")
```

Numeric Fields:

Numeric Field Methods:

- getNumber() and setNumber(result) for input and output of numbers.
- Conversion between numbers and strings is automatic.
- Example:

```
number = self.inputField.getNumber()
result = math.sqrt(number)
self.outputField.setNumber(result)
```

Numeric Fields:

```
class NumberFieldDemo(EasyFrame):
    def __init__(self):
        EasyFrame.__init__(self, title="Number Field Demo")
        self.addLabel(text="An integer", row=0, column=0)
        self.inputField = self.addIntegerField(value=0, row=0, column=1,
     width=10)
        self.addLabel(text="Square root", row=1, column=0)
        self.outputField = self.addFloatField(value=0.0, row=1, column
    =1, width=8, precision=2, state="readonly")
        self.addButton(text="Compute", row=2, column=0, columnspan=2,
    command=self.computeSqrt)

    def computeSqrt(self):
        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")
```

Numeric Fields:

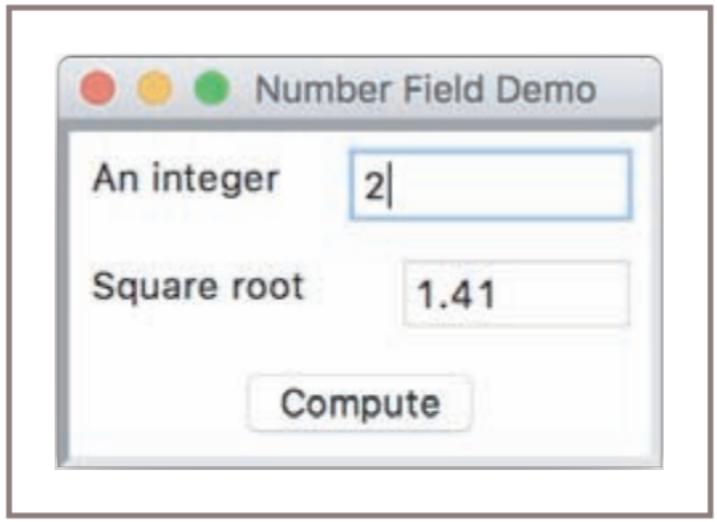

Figure: Using an integer field and a float field for input and output

Numeric Fields:

Handling Errors with Pop-Up Message Boxes:

- Use try-except to handle exceptions and show error messages in a pop-up box.
- Example:

```
def computeSqrt(self):
    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")
```

Numeric Fields:

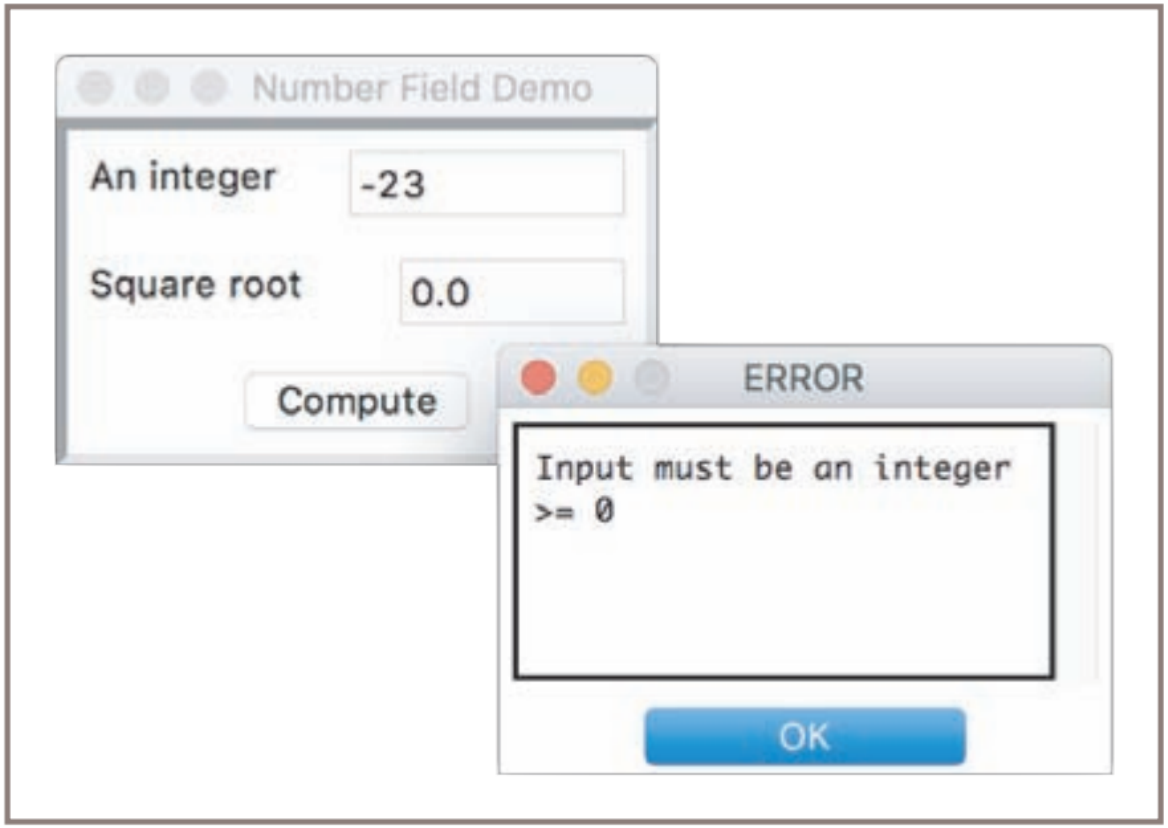

Figure: Responding to an input error with a message box

Instance Variables in Python:

Instance Variables in Python:

- Instance variables are used to store data specific to an individual object.
- They define the state of an object and are crucial in object-oriented programming.

Adding Instance Variables:

- Instance variables are added within the class's __init__ method.
- They must begin with the keyword self to be accessible throughout the class.
- Example:

```
def __init__(self):
    self.count = 0  # Instance variable to track the count
```

Instance Variables in Python:

Example Program - Counter Demo:

```
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)
        # 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)
```

Instance Variables in Python:

```
    # 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)
```

Instance Variables in Python:

- The CounterDemo class inherits from EasyFrame.
- The __init__ method initializes the window and sets up the UI components.
  - self.count is an instance variable that tracks the count.
  - next and reset are methods to handle the Next and Reset buttons, respectively.
- The instance variable self.count is updated and reflected in the label when buttons are clicked.

Instance Variables in Python:

Figure: The GUI for a counter application

Check Buttons in Python:

Check Buttons in Python:

- Check buttons consist of a label and a box that users can select or deselect.
- Often used to represent a group of options where any number of options can be selected.

Adding Check Buttons:

- addCheckbutton method is used to add check buttons to a window. Syntax:

```
addCheckbutton(text, row, column, ...optional_arguments...)
```

- Returns an EasyCheckbutton object.

Check Buttons in Python:

Example Program - Check Button Demo:

```
class CheckbuttonDemo(EasyFrame):
    """Allows the user to place a restaurant order from a set of options
    ."""
    def __init__(self):
        """Sets up the window and widgets."""
        EasyFrame.__init__(self, "Check Button Demo")
        # Add four check buttons
        self.chickCB = self.addCheckbutton(text="Chicken", row=0, column
    =0)
        self.taterCB = self.addCheckbutton(text="French fries", row=0,
    column=1)
        self.beanCB = self.addCheckbutton(text="Green beans", row=1,
    column=0)
        self.sauceCB = self.addCheckbutton(text="Applesauce", row=1,
    column=1)
        # Add the command button
        self.addButton(text="Place order", row=2, column=0, columnspan
    =2, command=self.placeOrder)
```

Check Buttons in Python:

Example Program - Check Button Demo:

```
    # Event handling method.
    def placeOrder(self):
        """Display a message box with the order information."""
        message = ""
        if self.chickCB.isChecked():
            message += "Chicken\n\n"
        if self.taterCB.isChecked():
            message += "French fries\n\n"
        if self.beanCB.isChecked():
            message += "Green beans\n\n"
        if self.sauceCB.isChecked():
            message += "Applesauce\n"
        if message == "":
            message = "No food ordered!"
        self.messageBox(title="Customer Order", message=message)
```

Check Buttons in Python:

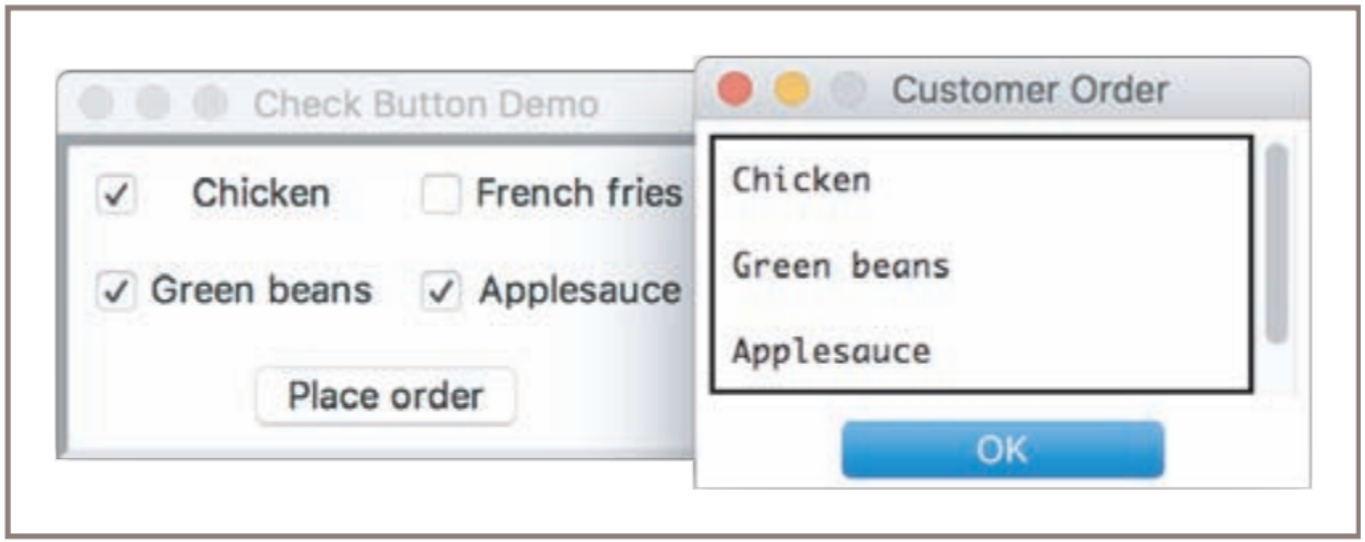

Figure: Using check buttons

Radio Buttons in Python

Radio Buttons in Python:

- Radio buttons are used when users must choose exactly one option from a group of options.
- Consist of labels and control widgets.
- Only one button in the group can be selected at a time.

Adding Radio Buttons:

- addRadiobuttonGroup method adds a radio button group to a window. Syntax:

```
addRadiobuttonGroup(row, column, ...optional_arguments...)
```

- Returns an EasyRadiobuttonGroup object.
- addRadiobutton method adds radio buttons to a radio button group.
- Syntax:

```
addRadiobutton(text, ...optional_arguments...)
```

- Returns a tkinter.Radiobutton object.

Radio Buttons in Python

```
from breezypythongui import EasyFrame

class RadiobuttonDemo(EasyFrame):
    """Allows the user to place a restaurant order from a set of options
    ."""
    def __init__(self):
        """Sets up the window and widgets."""
        EasyFrame.__init__(self, "Radio Button Demo")

        # Add the label, button group, and buttons for meats
        self.addLabel(text = "Meat", row = 0, column = 0)
        self.meatGroup = self.addRadiobuttonGroup(row = 1,column = 0,
    rowspan = 2)
        self.meatGroup.addRadiobutton(text = "Beef")
        self.meatGroup.addRadiobutton(text = "Mutton")
        defaultRB =self.meatGroup.addRadiobutton(text = "Chicken")
        self.meatGroup.setSelectedButton(defaultRB)

        self.addButton(text = "Place order", row = 3, column = 0,
        columnspan = 3, command = self.placeOrder)
```

Radio Buttons in Python

```
    # Event handler method
    def placeOrder(self):
        """Display a message box with the order information."""
        message = self.meatGroup.getSelectedButton()["text"]

        self.messageBox(title = "Customer Order", message = message)

def main():
    RadiobuttonDemo().mainloop()

# Instantiate and pop up the window."""
if __name__ == "__main__":
    main()
```

Radio Buttons in Python

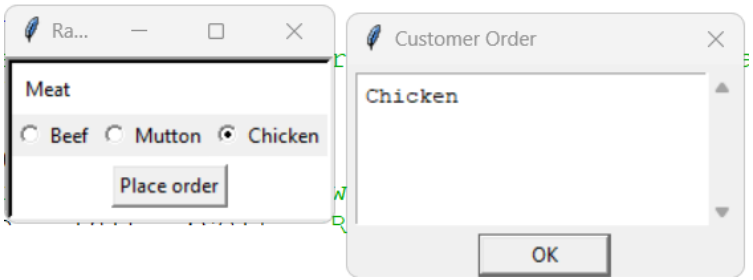

Figure: Using radio buttons

Radio Buttons in Python

Example Program - Radio Button Demo:

```
class RadiobuttonDemo(EasyFrame):
    """Allows the user to place a restaurant order from a set of
options."""
    def __init__(self):
        """Sets up the window and widgets."""
        EasyFrame.__init__(self, "Radio Button Demo")
        # Add the label, button group, and buttons for meats
        self.addLabel(text="Meat", row=0, column=0)
        self.meatGroup = self.addRadiobuttonGroup(row=1, column=0,
rowspan=2)
        defaultRB = self.meatGroup.addRadiobutton(text="Chicken")
        self.meatGroup.setSelectedButton(defaultRB)
        self.meatGroup.addRadiobutton(text="Beef")
```

Radio Buttons in Python

Example Program - Radio Button Demo:

```
        # Add the label, button group, and buttons for potatoes
        self.addLabel(text="Potato", row=0, column=1)
        self.taterGroup = self.addRadiobuttonGroup(row=1, column=1,
rowspan=2)
        defaultRB = self.taterGroup.addRadiobutton(text="French
fries")
        self.taterGroup.setSelectedButton(defaultRB)
        self.taterGroup.addRadiobutton(text="Baked potato")
                    # Add the label, button group, and buttons for
veggies
        self.addLabel(text="Vegetable", row=0, column=2)
        self.vegGroup = self.addRadiobuttonGroup(row=1, column=2,
rowspan=2)
        defaultRB = self.vegGroup.addRadiobutton(text="Applesauce")
        self.vegGroup.setSelectedButton(defaultRB)
        self.vegGroup.addRadiobutton(text="Green beans")
        self.addButton(text="Place order", row=3, column=0,
columnspan=3, command=self.placeOrder)
```

Radio Buttons in Python

```
    # Event handler method.
    def placeOrder(self):
        """Display a message box with the order information."""
        message = ""
        message += self.meatGroup.getSelectedButton()["text"] + "\n\
n"
        message += self.taterGroup.getSelectedButton()["text"] + "\n
\n"
        message += self.vegGroup.getSelectedButton()["text"]
        self.messageBox(title="Customer Order", message=message)
```

Radio Buttons in Python

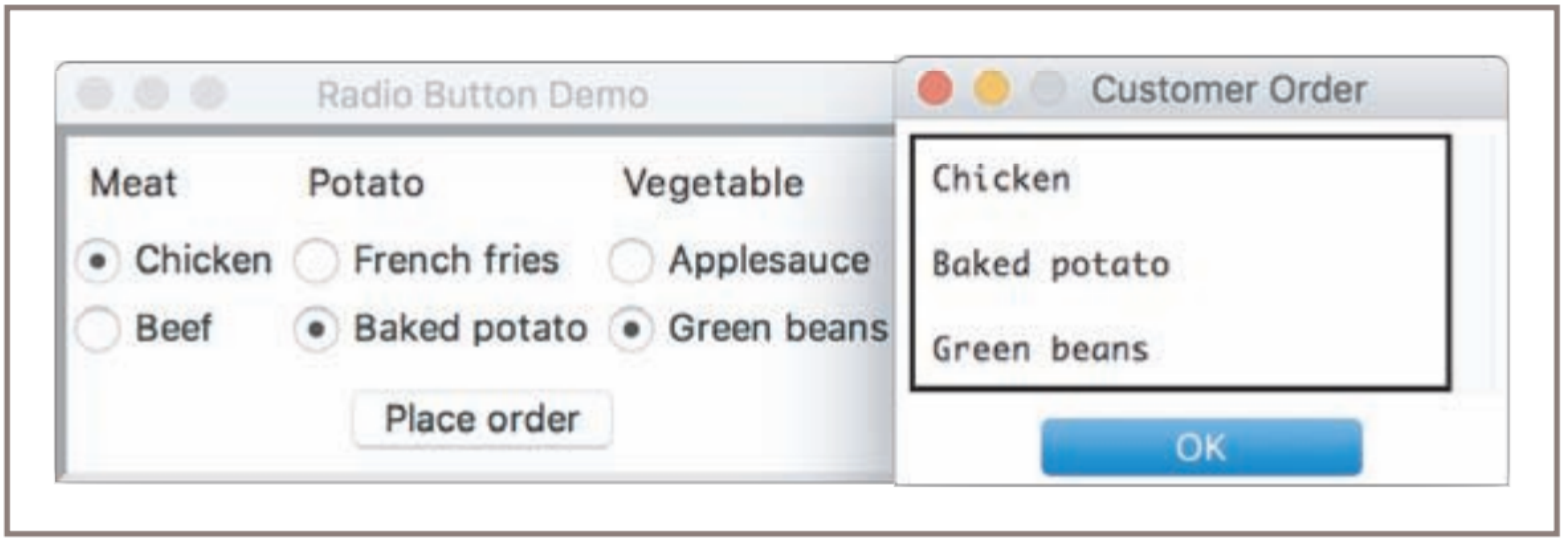

Figure: Using radio buttons