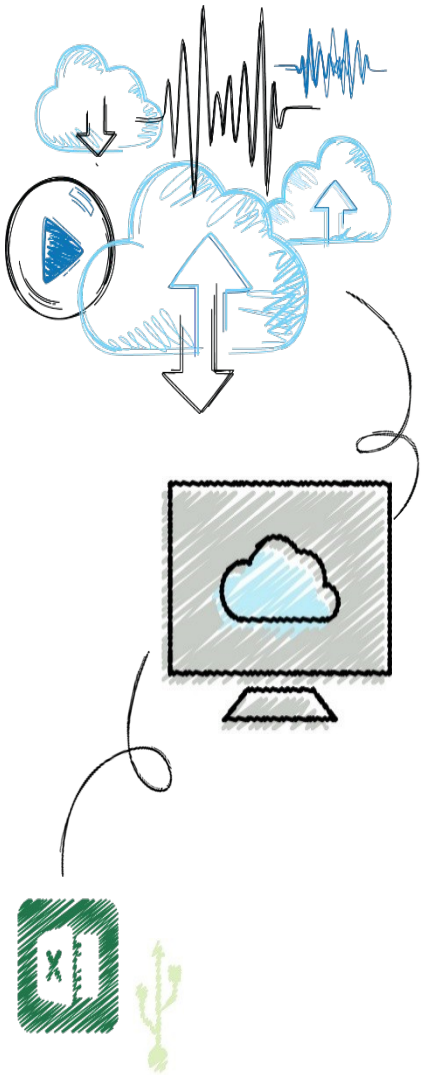
Fundamentals of Python: First Programs Second Edition

Chapter 7

Simple Graphics and Image Processing

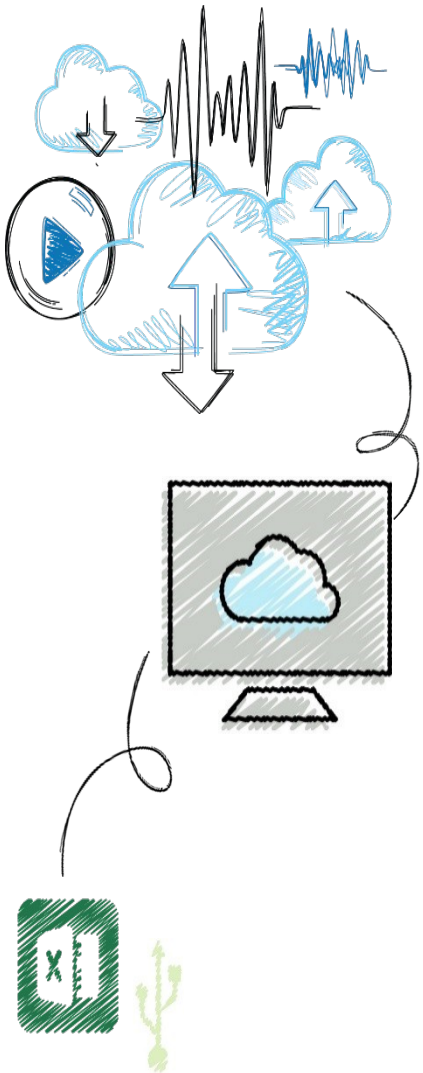


Objectives (1 of 2)

7.1 Use the concepts of object-based programming—classes, objects, and methods—to solve a problem

7.2 Develop algorithms that use simple graphics operations to draw two-dimensional shapes

7.3 Use the RGB system to create colors in graphics applications and modify pixels in images



Objectives (2 of 2)

7.4 Develop recursive algorithms to draw recursive shapes

7.5 Write a nested loop to process a two-dimensional grid

7.6 Develop algorithms to perform simple transformations of images, such as conversion of color to grayscale

Simple Graphics

- **Graphics:** Discipline that underlies the representation and display of geometric shapes in two- and three-dimensional space
- A **Turtle graphics** toolkit provides a simple and enjoyable way to draw pictures in a window
  - **turtle** is a non-standard, open-source Python module

Overview of Turtle Graphics (1 of 2)

- Turtle graphics originally developed as part of the children's programming language Logo
  - Created by Seymour Papert and his colleagues at MIT in the late 1960s
- Analogy: Turtle crawling on a piece of paper, with a pen tied to its tail
  - Sheet of paper is a window on a display screen
  - Position specified with (**x, y**) coordinates
    - Cartesian **coordinate system**, with origin (0, 0) at the center of a window

Overview of Turtle Graphics (2 of 2)

Table 7-1 Some attributes of a turtle

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 degree.
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.

Turtle Operations (1 of 3)

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
t.left(degrees) **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** to the specified distance in the current direction
t.pencolor(r, g, b) **t.pencolor(string)**	Changes the pen color of **t** to the specified RGB value or to the specified string
t.fillcolor(r, g, b) **t.fillcolor(string)**	Changes the fill color of **t** to the specified RBG value or to the specified string

Turtle Operations (2 of 3)

Turtle Method	What It Does
t.begin_fill() **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
t.hideturtle() **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

Turtle Operations (3 of 3)

- **Interface:** set of methods of a given class
  - Used to interact with an object
  - Use docstring mechanism to view an interface
    - **help(<class name>)**
    - **help(<class name>.<method name>)**

```
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)
```

Setting Up a turtle.cfg File and Running IDLE

- Turtle graphics configuration file (turtle.cfg)
  - Contains initial settings of several attributes of **Turtle**, **Screen**, and **Canvas** objects
  - Python creates default settings for these attributes
- The attributes in the file used for most of out examples are:

  width = 300

  height = 200

  using_IDLE = True

  colormode = 255

- To create a file with these settings:
  - Open a text editor, enter the settings, and save the file as **turtle.cfg**

Object Instantiation and the Turtle Module (1 of 3)

- Before you apply any methods to an object, you must create the object (i.e., an **instance** of)
- **Instantiation:** Process of creating an object
- Use a **constructor** to instantiate an object:

  <variable name> = <class name>(<any arguments>)
- To instantiate the **Turtle** class:

```
>>> from turtle import Turtle
>>> t = Turtle()
```

Object Instantiation and the Turtle Module (2 of 3)

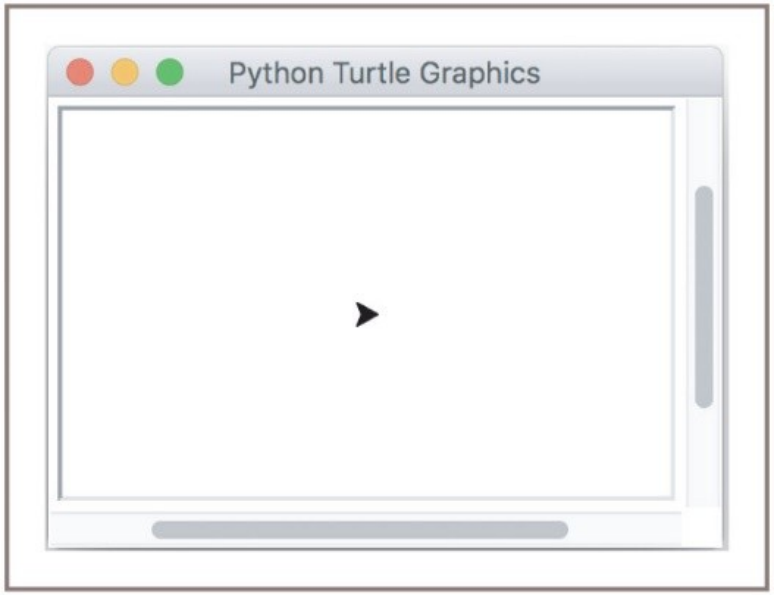


Figure 7-1 Drawing window for a turtle

- To close a turtle's window, click its close box
- Attempting to manipulate a turtle whose window has been closed raises an error

Object Instantiation and the Turtle Module (3 of 3)

```
>>> 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
```

Figure 7-2 Drawing vertical and horizontal lines for the letter T

Drawing Two-Dimensional Shapes (1 of 3)

- Many graphics applications use **vector graphics**, or the drawing of simple two-dimensional shapes, such as rectangles, triangles, and circles
- Example code:

```
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 (2 of 3)

- The code for a function to draw a pattern named radialHexagons, expects a turtle, the number of hexagons, and the length of a side as arguments:

```
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)
```

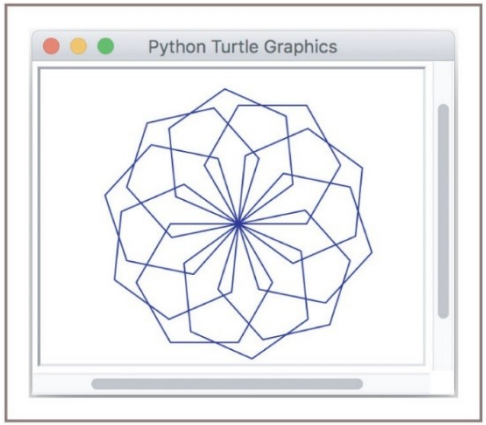


Figure 7-3 A radial pattern with 10 hexagons

Drawing Two-Dimensional Shapes (3 of 3)

- The code for drawing a radial pattern using any regular polygon followed by a session using it with squares and hexagons:

```
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)

>>> t = Turtle()
>>> radialPattern(t, n = 10, length = 50, shape = square)
>>> t.clear()
>>> radialPattern(t, n = 10, length = 50, shape = hexagon)
```

Examining an Object's Attributes

- **Mutator methods** - methods that change the internal state of a Turtle object
- **Accessor methods** – methods that return the values of a Turtle object's attributes without altering its state
- Code that shows accessor methods in action:

```
>>> from turtle import Turtle
>>> t = Turtle()
>>> t.position()
(0.0, 0.0)
>>> t.heading()
0.0
>>> t.isdown()
True
```

Manipulating a Turtle's Screen

- Access a turtle's **Screen** object using the notation **t.screen**
  - Then call a Screen method with this object
- Methods **window_width()** and **window_height()** can be used to locate the boundaries of a turtle's window
- Code that resets the screen's background color:

```
>>> from turtle import Turtle
>>> t = Turtle()
>>> t.screen.bgcolor("orange")
>>> x = t.screen.window_width() // 2
>>> y = t.screen.window_height() // 2
>>> print((-x, y), (x, -y))
```

Taking a Random Walk (1 of 2)

```
from turtle import Turtle
import random

def randomWalk(t, turns, distance = 20):
    """Turns a random number of degrees and moves a given
    distance for a fixed number of turns."""
    for x in range(turns):
    if x % 2 == 0:
      t.left(random.randint(0, 270))
    else:
      t.right(random.randint(0, 270))
    t.forward(distance)
def main():
    t = Turtle()
    t.shape("turtle")
    randomWalk(t, 40, 30)
if __name__ == "__main__":
    main()
```

Taking a Random Walk (2 of 2)

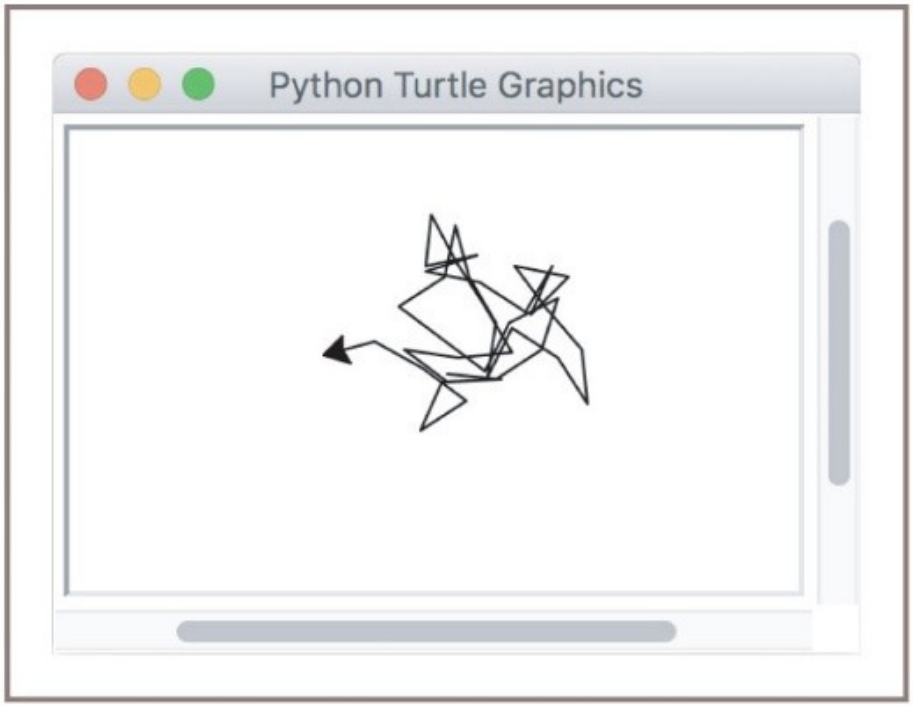


Figure 7-4 A random walk

Colors and the RGB System (1 of 2)

- Display area on a computer screen is made up of colored dots called picture elements or **pixels**
- Each pixel represents a color – the default is black
  - Change the color by running the **pencolor** method
- **RGB** is a common system for representing colors
  - RGB stands for red, green, and blue
  - Each color component can range from 0 – 255
    - 255 → maximum saturation of a color component
    - 0 → total absence of that color component
  - A **true color** system

Colors and the RGB System (2 of 2)

- Each color component requires 8 bits; total number of bits needed to represent a color value is 24
  - Total number of RGB colors is 2^{24} (16,777,216)

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 (1 of 5)

- The **Turtle** class includes a **pencolor** method for changing the turtle's drawing color
  - Expects integers for the three RGB components
- Script that draws radial patterns of squares and hexagons with random fill colors at the corners of turtle's window (output is shown in Figure 7-5):

```
"""
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
```

Example: Filling Radial Patterns with Random Colors (2 of 5)

```
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()
```

Example: Filling Radial Patterns with Random Colors (3 of 5)

```
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)
```

Example: Filling Radial Patterns with Random Colors (4 of 5)

```
        # 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)
        # Draw hexagons in lower–right corner
        drawPattern(t, width – inset, inset – height, count,
                length, hexagon)
if _name_== “_main_”:
   main()
```

Example: Filling Radial Patterns with Random Colors (5 of 5)

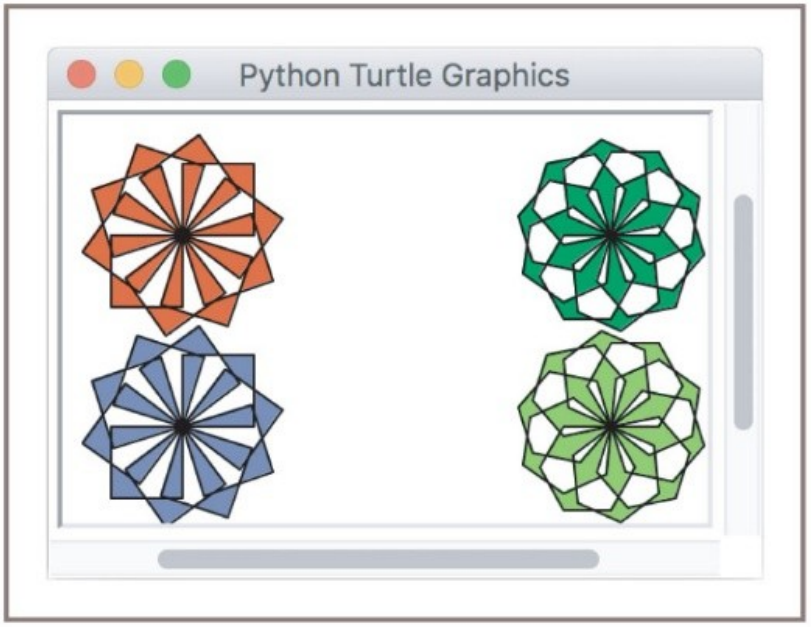


Figure 7-5 Radial patterns with random fill colors

Image Processing

- Digital image processing 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

Analog and Digital Information

- Computers must use digital information which consists of **discrete values**
  - Example: Individual integers, characters of text, or bits
- 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
- Ticks representing seconds on an analog clock's face represent an attempt to **sample** moments of time as discrete values (time itself is analog)

Sampling and Digitizing Images

- A visual scene projects an infinite set of color and intensity values onto a two-dimensional sensing medium
  - If you sample enough of these values, digital information can represent an image more or less indistinguishable (to human eye) from original scene
- Sampling devices measure discrete color values at distinct points on a two-dimensional **grid**
  - These values are pixels
  - As more pixels are sampled, the more realistic the resulting image will appear

Image File Formats

- Once an image has been sampled, it can be stored in one of many file formats
- A **raw image file** saves all of the sampled information
- Data can be compressed to minimize its file size
  - JPEG (Joint Photographic Experts Group)
    - Uses **lossless compression** and a **lossy scheme**
  - GIF (Graphic Interchange Format)
    - Uses a lossy compression and a **color palette** of up to 256 of the most prevalent colors in the image

Image-Manipulation Operations

- Image-manipulation programs either transform the information in the pixels or alter the arrangement of the pixels in the image
- Examples:
  - 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

- The coordinates of pixels in the two-dimensional grid of an image range from (0, 0) at the upper-left corner to (**width**−1, **height**−1) at lower-right corner
  - **width/height** are the image's dimensions in pixels
  - Thus, the **screen coordinate system** for the display of an image is different from the standard Cartesian coordinate system that we used with Turtle graphics
- The RGB color system is a common way of representing the colors in images

The images Module (1 of 5)

- The **images** module is a non-standard, open-source Python tool
  - **Image** class represents an image as a two-dimensional grid of RGB values
- The following session with the interpreter does the following:
  - Imports the Image class from the images module
  - Instantiates this class using the file named smokey.gif
  - Draws the image

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

Figure 7-9 An image display window

The images Module (2 of 5)

- 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 images Module (3 of 5)

- 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):

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

- The programmer can use the method setPixel to replace an RGB value at a given position in an image:

```
>>> 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()
```

CENGAGE

The images Module (4 of 5)

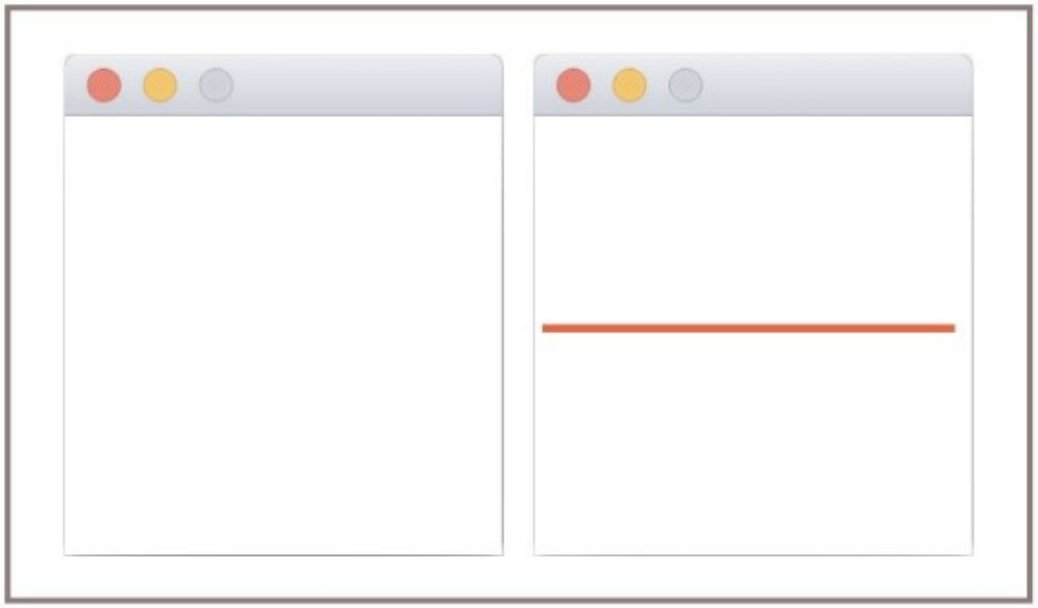


Figure 7-10 An image before and after replacing the pixels

The images Module (5 of 5)

- Use the **save** operation to write an image back to an existing file using the current filename
- The following code saves the new image using the filename horizontal.gif:

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

A Loop Pattern for Traversing a Grid (1 of 2)

- Most of the loops we have used in this book have had a **linear loop structure**
- Many image-processing algorithms use a **nested loop structure** to traverse a two-dimensional grid of pixels

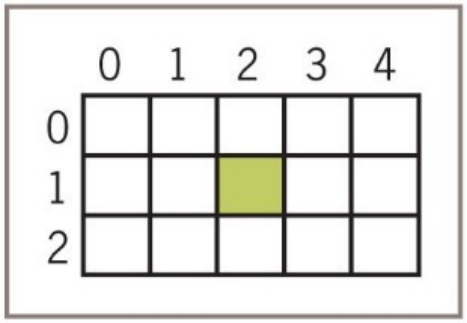


Figure 7-11 A grid with 3 rows and 5 columns

A Loop Pattern for Traversing a Grid (2 of 2)

- The following loop uses a **row-major traversal**

```
>>> width = 2
>>> height = 3
>>> for y in range(height):
for x in range(width):
    print((x, y), end = " ")
    print()
(0, 0) (1, 0)
(0, 1) (1, 1)
(0, 2) (1, 2)
```

- We use this template to develop many of the algorithms that follow:

```
for y in range(height):
for x in range(width):
<do something at position (x, y)>
```

A Word on Tuples

- A pixel's RGB values are stored in a tuple:

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


>>> r
194
>>> g
221
>>> b
114

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

Converting an Image to Black and White (1 of 2)

- For each pixel, compute average of R/G/B values
- Then, reset pixel's color values to 0 (black) if the average is closer to 0, or to 255 (white) if the average is closer to 255

```
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 (2 of 2)

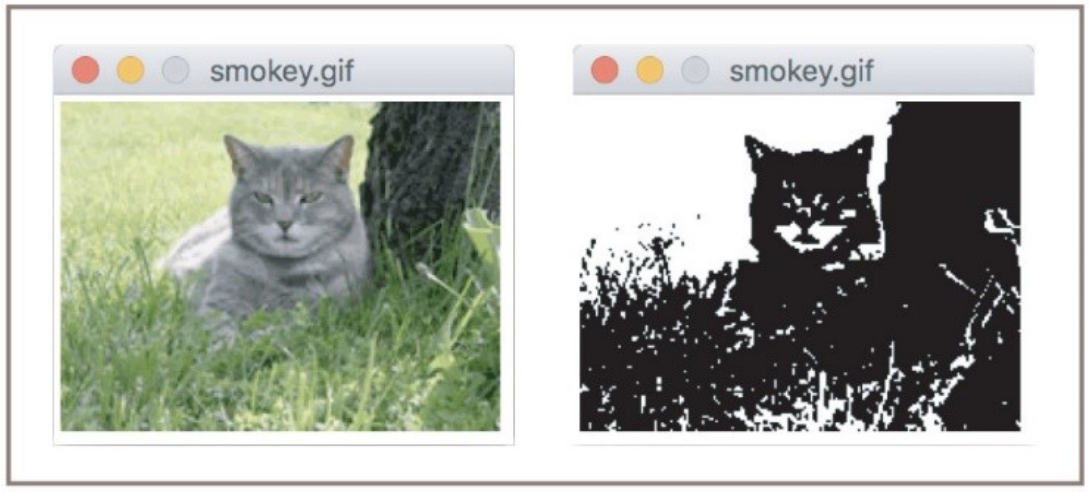


Figure 7-12 Converting a color image to black and white

Converting an Image to Grayscale (1 of 3)

- Black and white photographs contain various shades of gray known as **grayscale**
- Grayscale can be an economical scheme (the only color values might be 8, 16, or 256 shades of gray)
- A simple method:

  average = (r + g + b) // 3

  image.setPixel(x, y, (average, average, average)

- Problem: Does not reflect manner in which different color components affect human perception
- Scheme needs to take differences in **luminance** into account

Converting an Image to Grayscale (2 of 3)

```
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))
```

Converting an Image to Grayscale (3 of 3)

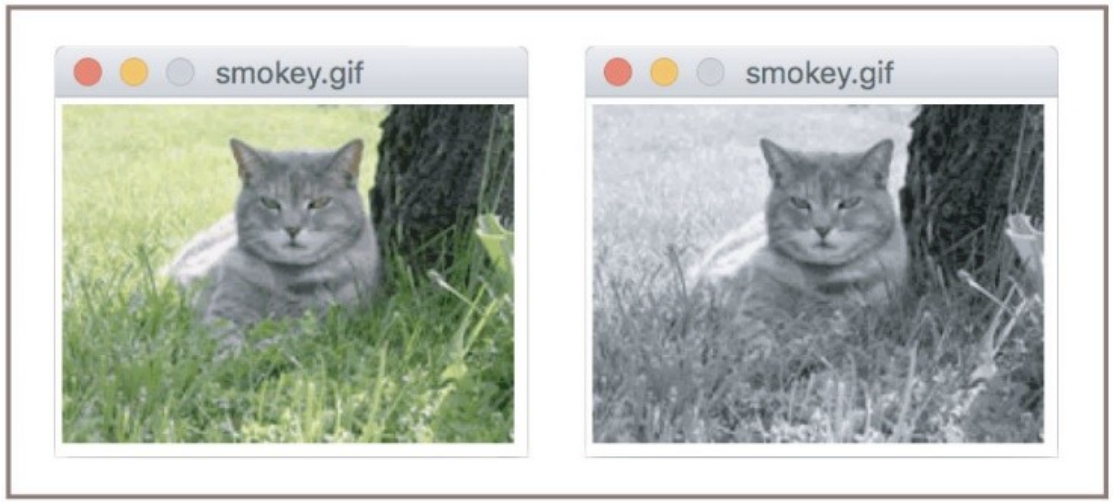


Figure 7-13 Converting a color image to grayscale

Copying an Image

- The method **clone** builds and returns a new image with the same attributes as the original one, but with an empty string as the filename

```
>>> from images import Image
>>> image = Image("smokey.gif")
>>> image.draw()
>>> newImage = image.clone() # Create a copy of image
>>> newImage.draw()
>>> grayscale(newImage) # Change in second window only
>>> newImage.draw()
>>> image.draw() # Verify no change to original
```

Blurring an Image

- **Pixilation** can be mitigated by **blurring**

```
def blur(image):
   """Builds and returns a new image which is a
   blurred copy of the argument image."""
   def tripleSum(triple1, triple2):
#1
        (r1, g1, b1) = triple1
        (r2, g2, b2) = triple2
        return (r1 + r2, g1 + g2, b1 + b2)
   new = image.clone()
    for y in range(1, image.getHeight() - 1):
        for x in range(1, image.getWidth() - 1):
           oldP = image.getPixel(x, y)
           left = image.getPixel(x - 1, y) # To left
           right = image.getPixel(x + 1, y) # To right
           top = image.getPixel(x, y - 1) # Above
           bottom = image.getPixel(x, y + 1) # Below
           sums = reduce(tripleSum,
           [oldP, left, right, top, bottom])
#2
           averages = tuple(map(lambda x: x // 5, sums))
#3
           new.setPixel(x, y, averages)
        return new
```

Edge Detection (1 of 3)

- **Edge detection** removes the full colors to uncover the outlines of the objects represented in the image

```
def detectEdges(image, amount):
    """Builds and returns a new image in which the edges of
    the argument image are highlighted and the colors are
    reduced to black and white."""

    def average(triple):
        (r, g, b) = triple
        return (r + g + b) // 3

    blackPixel = (0, 0, 0)
    whitePixel = (255, 255, 255)
    new = image.clone()
```

Edge Detection (2 of 3)

- **Edge detection** (continued)

```
for y in range(image.getHeight() – 1):
        for x in range(1, image.getWidth()):
          oldPixel = image.getPixel(x, y)
          leftPixel = image.getPixel(x – 1, y)
          bottomPixel = image.getPixel(x, y + 1)
          oldLum = average(oldPixel)
          leftLum = average(leftPixel)
          bottomLum = average(bottomPixel)
          if abs(oldLum – leftLum) > amount or \
              abs(oldLum – bottomLum) > amount:
              new.setPixel(x, y, blackPixel)
          else:
              new.setPixel(x, y, whitePixel)
return new
```

Edge Detection (3 of 3)

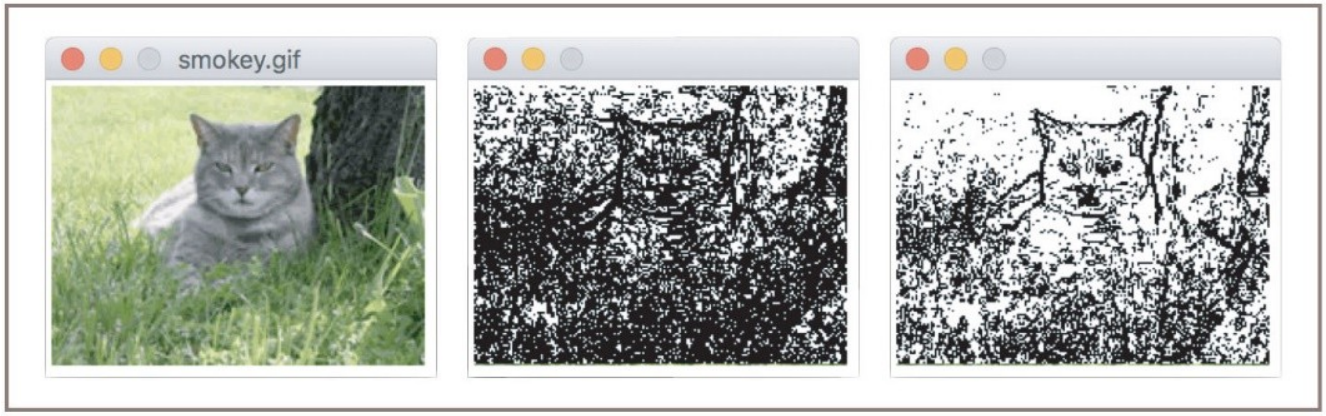


Figure 7-14 Edge detection: the original image, a luminance threshold of 10, and a luminance threshold of 20

Reducing the Image Size (1 of 2)

- The size and the quality of an image on a display medium depend on two factors:
  - Image's width and height in pixels
  - Display medium's **resolution**
    - Measured in pixels, or dots per inch (DPI)
- The resolution of an image can be set before the image is captured
  - A higher DPI causes sampling device to take more samples (pixels) through the two-dimensional grid
- A size reduction usually preserves an image's **aspect ratio**
- Reducing an image's size throws away some of its pixel information

Reducing the Image Size (2 of 2)

```
def shrink(image, factor):
    """Builds and returns a new image which is a smaller
    copy of the argument image, by the factor argument."""
    width = image.getWidth()
    height = image.getHeight()
    new = Image(width // factor, height // factor)
    oldY = 0
    newY = 0
    while oldY < height - factor:
      oldX = 0
      newX = 0
      while oldX < width - factor:
          oldP = image.getPixel(oldX, oldY)
          new.setPixel(newX, newY, oldP)
          oldX += factor
          newX += 1
      oldY += factor
      newY += 1
    return new
```

Chapter Summary (1 of 3)

- Object-based programming uses classes, objects, and methods to solve problems
- A class specifies a set of attributes and methods for the objects of that class
- The values of the attributes of a given object make up its state
- A new object is obtained by instantiating its class
- The behavior of an object depends on its current state and on the methods that manipulate this state
- The set of a class's methods is called its interface

Chapter Summary (2 of 3)

- Turtle graphics is a lightweight toolkit used to draw pictures in a Cartesian coordinate system
- RGB system represents a color value by mixing integer components that represent red, green, and blue intensities
- A grayscale system uses 8, 16, or 256 distinct shades of gray
- Digital images are captured by sampling analog information from a light source, using a device such as a digital camera or a flatbed scanner
  - Can be stored in several formats, like JPEG and GIF

Chapter Summary (3 of 3)

- When displaying an image file, each color value is mapped onto a pixel in a two-dimensional grid
  - A nested loop structure is used to visit each position
- Image-manipulation algorithms either transform pixels at given positions or create a new image using the pixel information of a source image