

Case Studies

- CS.2 [Turtle Graphics](#) 442
- CS.3 [Automating Turtle Graphics](#) 448
- CS.4 [Processing Image Files](#) 452
- CS.5 [Image-Processing Algorithms](#) 458
- CS.6 [Games of Chance](#) 465
- CS.7 [Debugging with a Debugger](#) 471
- CS.8 [Indexing and Iterators](#) 479
- CS.9 [Developing a Calculator](#) 486
- CS.10 [Tower of Hanoi](#) 492
- CS.11 [Web Crawlers](#) 498
- CS.12 [Data Interchange](#) 506

CS.2 Turtle Graphics

In our first case study, we will use a graphics tool to (visually) illustrate the concepts covered in Chapter 2: objects, classes and class methods, object-oriented programming, and modules. The tool, Turtle graphics, allows a user to draw lines and shapes in a way that is similar to using a pen on paper.

DETOUR



Turtle Graphics

Turtle graphics has a long history going all the way back to the time when the field of computer science was developing. It was part of the Logo programming language developed by Daniel Bobrow, Wally Feurzig, and Seymour Papert in 1966. The Logo programming language and its most popular feature, turtle graphics, was developed for the purpose of teaching programming.

The turtle was originally a robot, that is, a mechanical device controlled by a computer operator. A pen was attached to the robot and it left a trace on the surface as the robot moved according to functions input by the operator.

Turtle graphics is available to Python developers through the `turtle` module. In the module are defined 7 classes and more than 80 class methods and functions. We will not exhaustively cover all the features of the module `turtle`. We only introduce a sufficient number to allow us to do interesting graphics while cementing our understanding of objects, classes, class methods, functions, and modules. Feel free to explore this fun tool on your own.

Classes Screen and Turtle

We start by importing the turtle module and then instantiating a Screen object.

```
>>> import turtle
>>> s = turtle.Screen()
```

You will note that a new window appears with a white background after executing the second statement. The Screen object is the canvas on which we draw. The Screen class is defined in the turtle module. Later we will introduce some Screen methods that change the background color or close the window. Right now, we just want to start drawing.

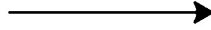
To create our pen or, using the turtle graphics terminology, our turtle, we instantiate a Turtle object we name `t`:

```
>>> t = turtle.Turtle()
```

A Turtle object is essentially a pen that is initially located at the center of the screen, at coordinates (0,0). The Turtle class, defined in the turtle module, provides many methods for moving the turtle. As we move the turtle, it leaves a trace behind. To make our first move, we will use the `forward()` method of the Turtle class. So, to move forward 100 pixels, the method `forward()` is invoked on Turtle object `t` with 100 (pixels) as the distance:

```
>>> t.forward(100)
```

The effect is shown in Figure CS.1.



Note that the move is to the right. When instantiated, the turtle faces right (i.e., to the east). To make the turtle face a new direction, you can rotate it counterclockwise or clockwise using the `left()` or `right()` methods, both Turtle class methods. To rotate 90 degrees counterclockwise, the method `left()` is invoked on Turtle object `t` with the argument 90:

```
>>> t.left(90)
```

We can have several Turtle objects simultaneously on the screen. Next, we create a new Turtle instance that we name `u` and make both turtles do some moves:

```
>>> u = turtle.Turtle()
>>> u.left(90)
>>> u.forward(100)
>>> t.forward(100)
>>> u.right(45)
```

The current state of the two turtles and the trace they made is shown in Figure CS.2.

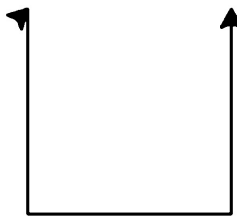


Figure CS.2 Two Turtle objects. The Turtle object on the left is facing northeast whereas the Turtle object on the right is facing north.

In the example we just completed, we used three methods of class Turtle: `forward()`, `left()`, and `right()`. In Table CS.1, we list those and some other methods (but by no means all). To illustrate some of the additional methods listed in the table, we go through the steps necessary to draw a smiley face emoticon shown in Figure CS.3.

We start by instantiating a Screen and a Turtle object and setting the pen size.

```
>>> import turtle
>>> s = turtle.Screen()
```

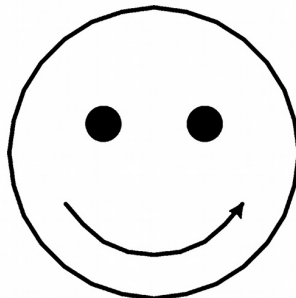


Figure CS.3 A Turtle smiley face drawing.

Table CS.1 Some methods of the Turtle class.

After importing the module turtle, you can obtain the full list of Turtle methods in your interactive shell using `help(turtle.Turtle)`

Usage	Explanation
<code>t.forward(distance)</code>	Move turtle in the direction the turtle is headed by distance pixels
<code>t.left(angle)</code>	Rotate turtle counterclockwise by angle degrees
<code>t.right(angle)</code>	Rotate turtle clockwise by angle degrees
<code>t.undo()</code>	Undo the previous move
<code>t.goto(x, y)</code>	Move turtle to coordinates defined by x and y; if pen is down, draw line
<code>t.setx(x)</code>	Set the turtle's first coordinate to x
<code>t.sety(y)</code>	Set the turtle's second coordinate to y
<code>t.setheading(angle)</code>	Set orientation of turtle to angle, given in degrees; Angle 0 means east, 90 is north, and so on
<code>t.circle(radius)</code>	Draw a circle with given radius; the center of the circle is radius pixels to the left of the turtle
<code>t.circle(radius, angle)</code>	Draw only the part the circle (see above) corresponding to angle
<code>t.dot(diameter, color)</code>	Draw a dot with given diameter and color
<code>t.penup()</code>	Pull pen up; no drawing when moving
<code>t.pendown()</code>	Put pen down; drawing when moving
<code>t.pensize(width)</code>	Set the pen line thickness to width
<code>t.pencolor(color)</code>	Set the pen color to color described by string color

```
>>> t = turtle.Turtle()
>>> t.pensize(3)
```

We then define the coordinates where the chin of the smiley face will be located and then move to that location.

```
>>> x = -100
>>> y = 100
>>> t.goto(x, y)
```

Oooops! We drew a line from coordinate (0, 0) to coordinate (−100, 100); all we wanted was to move the pen, without leaving a trace. So we need to undo the last move, lift the pen, and then move it.

```
>>> t.undo()
>>> t.penup()
>>> t.goto(x, y)
>>> t.pendown()
```

Now we want to draw the circle outlining the face of our smiley face. We call the method `circle()` of the class `Turtle` with one argument, the radius of the circle. The circle is drawn as follows: The current turtle position will be a point of the circle, and the center of the circle is defined to be to the turtle's left, with respect to the current turtle heading.

```
>>> t.circle(100)
```

Now we want to draw the left eye. We choose the left eye coordinates relative to (x, y) (i.e., the chin position) and “jump” to that location. We then use the `dot` function to draw a black dot of diameter 10.

Usage	Explanation
<code>s.bgcolor(color)</code>	Changes the background color of screen <code>s</code> to color described by string <code>color</code>
<code>s.clearscreen()</code>	Clears screen <code>s</code>
<code>s.turtles()</code>	Returns the list of all turtles in the screen <code>s</code>
<code>s.bye()</code>	Closes the screen <code>s</code> window

Table CS.2 Methods of the Screen class. Shown are only some of the Screen class methods. After importing module `turtle`, you can obtain the full list of Screen methods in your interactive shell using `help(turtle.Screen)`

```
>>> t.penup()
>>> t.goto(x - 35, y + 120)
>>> t.pendown()
>>> t.dot(25)
```

Next, we jump and draw the right eye.

```
>>> t.penup()
>>> t.goto(x + 35, y + 120)
>>> t.pendown()
>>> t.dot(25)
```

Finally, we draw the smile. I chose the exact location of the left endpoint of the smile using trial and error. You could also use geometry and trigonometry to get it right if you prefer. We use here a variant of the method `circle()` that takes a second argument in addition to the radius: an angle. What is drawn is just a section of the circle, a section corresponding to the given angle. Note that we again have to jump first.

```
>>> t.penup()
>>> t.goto(x - 60.62, y + 65)
>>> t.pendown()
>>> t.setheading(-60)
>>> t.circle(70, 120)
```

We're done! As we end this case study, you may wonder how to close cleanly your turtle graphics window. The Screen method `bye()` closes it:

```
>>> s.bye()
```

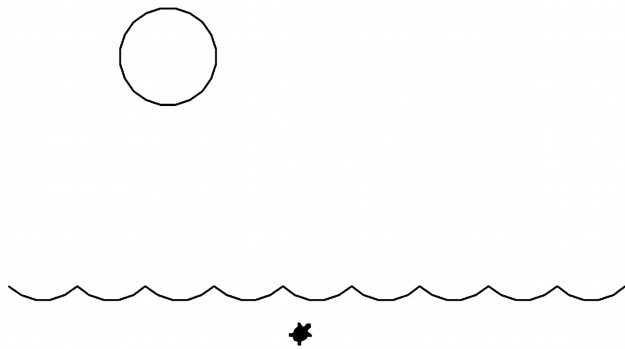
This method and several other Screen methods are listed in Table CS.2.

Start by executing these statements:

```
>>> import turtle
>>> s = turtle.Screen()
>>> t = turtle.Turtle(shape='turtle')
>>> t.penup()
>>> t.goto(-300, 0)
>>> t.pendown()
```

A turtle pen will appear on the left side of the screen. Then execute a sequence of Python turtle graphics statements that will produce this image:

Practice Problem
CS.1



Solution to the Practice Problem

CS.1 We assume the starting position is the leftmost point of the "wave" curve. To draw the first "valley," we need to make the turtle point southeast and then draw a 90° section of the circle:

```
>>> t.setheading(-45)
>>> t.circle(50, 90)
```

We then repeat this pair of statements eight times. To draw the Sun, we need to lift the pen, move it, put the pen down, and draw a circle:

```
>>> t.penup()
>>> t.goto(-100, 200)
>>> t.pendown()
>>> t.circle(50)
>>> t.penup()
>>> t.goto(0, -50)
```

We end by moving the turtle so it can swim in the sea.

Problems

CS.2 Write Python statements that draw a square of side length 100 pixels using Turtle graphics. Make sure you import the module `turtle` first. Your first two and last statement should be as shown:

```
>>> s = turtle.Screen()    # create screen
>>> t = turtle.Turtle()    # create turtle

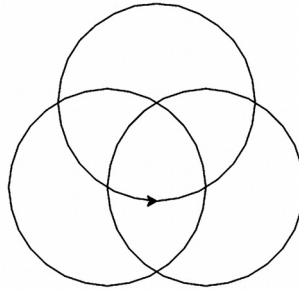
...                        # now write a sequence of statements
...                        # that draw the square

>>> s.bye()                # delete the Screen when done
```

CS.3 Using the approach from Problem CS.2, write Python statements that draw a diamond of side length 100 pixels using Turtle graphics.

CS.4 Using the approach from Problem CS.2, write Python statements that draw a pentagon of side length 100 pixels using Turtle graphics. Then do a hexagon, a heptagon, and an octagon.

CS.5 Using the approach from Problem CS.2, write Python statements that draw the intersecting circles of radius 100 pixels shown using Turtle graphics:



The sizes of the circles do not matter; their centers should be, more or less, the points of an equilateral triangle.

CS.6 Using the approach from Problem CS.2, write Python statements that draw four concentric circles similar to the concentric circles of a dartboard.

CS.7 Add three more swimming turtles to the picture shown in Practice Problem CS.1.

CS.8 Using Turtle graphics, illustrate the relative size of the Sun and the Earth by drawing two circles. The circle representing Earth should have a radius of 1 pixel. The circle representing the Sun should have a radius of 109 pixels.

CS.9 Using Turtle graphics, draw a five-pointed star by repeating the following five times: move the turtle 100 pixels and then rotate it right 144 degrees. When done, consider how to draw the six-pointed star (commonly referred to as the Star of David).

CS.10 Using Turtle graphics, draw an image showing the six sides of a dice. You may represent each side inside a separate square.

CS.11 Using Turtle graphics, draw the lines of a basketball field. You may choose the National Basketball Association (NBA) or International Basketball Federation (FIBA) specifications, which you can easily find on the web.

CS.12 Using Turtle graphics, draw an image showing the (visible) phases of the moon as seen from your hemisphere: waxing crescent, first quarter, waxing gibbous, full, waning gibbous, third quarter, and waning crescent. You can find illustrations of the phases of the moon on the web.

CS.3 Automating Turtle Graphics

In Case Study CS.2 we implemented a sequence of Python statements—in other words, a program—that draws the picture of a smiley face. Take another look at that sequence of statements. You will notice that the statements were repetitive and somewhat tedious to type. This sequence of commands appeared several times:

```
t.penup()
t.goto(x, y)
t.pendown()
```

This sequence of Turtle method calls was used to move the Turtle object `t` to a new location (with coordinates (x, y)) without leaving a trace; in other words, it was used to make the Turtle object jump to the new location.

Function `jump()`

It would save us a lot of typing if we could replace that sequence of Python statements with just one. That is exactly what functions are for. What we want to do is to develop a function that takes a Turtle object and coordinates x and y as input arguments and makes the Turtle object jump to coordinate (x, y) . Here is that function:

Module: `turtlefunctions.py`

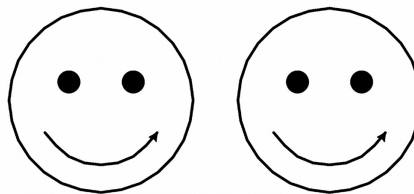
```
1 def jump(t, x, y):
2     'makes turtle t jump to coordinates (x, y)'
3     t.penup()
4     t.goto(x, y)
5     t.pendown()
```

Using this function instead of three statements shortens the process of drawing the smiley face image. It also makes the program more understandable because the function call `jump(t, x, y)`:

1. Better describes the action performed by the Turtle object
2. Hides the low-level and technical pen-up and -down operations, thus removing complexity from the program.

Suppose now we want to be able to draw several smiley faces next to each other as shown in Figure CS.4.

Figure CS.4 Two smiley faces. Ideally, each smiley face should be drawn with just one function call.



To do this, it would be useful to develop a function that takes as input a Turtle object and coordinates x and y and draws a smiley face at coordinate (x, y) . If we name this function `emoticon()`, we could use and reuse it to draw the image.

```
>>> import turtle
>>> s = turtle.Screen()
>>> t = turtle.Turtle()
```



```
>>> emoticon(t, -100, 100)
>>> emoticon(t, 150, 100)
```

Here is the implementation of the function:

```
1 def emoticon(t,x,y):
2     'turtle t draws a smiley face with chin at coordinate (x, y)'
3     # set turtle direction and pen size
4     t.pensize(3)
5     t.setheading(0)
6
7     # move to (x, y) and draw head
8     jump(t, x, y)
9     t.circle(100)
10
11    # draw right eye
12    jump(t, x+35, y+120)
13    t.dot(25)
14
15    # draw left eye
16    jump(t, x-35, y+120)
17    t.dot(25)
18
19    # draw smile
20    jump(t, x-60.62, y+65)
21    t.setheading(-60) # smile is a 120 degree
22    t.circle(70, 120) # section of a circle
```

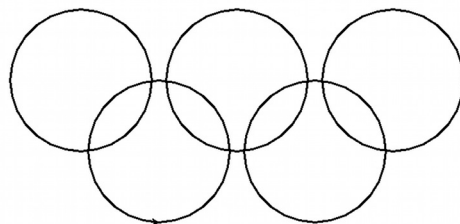
Module: turtlefunctions.py

We should note a few things about the program. Note the docstring with the strange triple quotes. In Python, strings, statements, and most expressions usually cannot span multiple lines. A string, whether defined with single quotes, as in 'example', or with double quotes, as in "example", cannot span multiple lines of a Python program. If, however, we need to define a string that does contain multiple lines, we must use triple quotes, as in '''example''' or """example""".

The rest of the function follows the steps we have already developed in the case study in Case Study CS.2. Note how we use the `jump()` function to make the program shorter and the steps of the program more intuitive.

Implement function `olympic(t)` that makes turtle `t` draw the Olympic rings shown below.

Practice Problem
CS.13



Use the `jump()` function from module `turtlefunctions`. You should be able to get the image drawn by executing:

```
>>> import turtle
>>> s = turtle.Screen()
>>> t = turtle.Turtle()
>>> olympic(t)
```

Solution to the Practice Problem

CS.13 The solution uses the `jump()` functions from module `turtlefunctions` we developed in the case study. In order for Python to import this module, it must be in the same folder as the module containing the `olympic()` function.

```
import turtlefunctions
def olympic(t):
    'has turtle t draw the olympic rings'
    t.pensize(3)
    turtlefunctions.jump(t, 0, 0)    # bottom of top center circle
    t.setheading(0)

    t.circle(100) # top center circle
    turtlefunctions.jump(t, -220, 0)
    t.circle(100) # top left circle
    turtlefunctions.jump(t, 220, 0)
    t.circle(100) # top right circle
    turtlefunctions.jump(t, 110, -100)
    t.circle(100) # bottom right circle
    turtlefunctions.jump(t, -110, -100)
    t.circle(100) # bottom left circle
```

Problems

CS.14 (This problem builds on Problem CS.4.) Implement function `polygon()` that takes a number $n \geq 3$ as input and draws, using Turtle graphics, an n -sided regular polygon.

CS.15 Using Turtle graphics, implement a function `grid()` that takes two positive integers m and n and draws a grid of size $m \times n$.

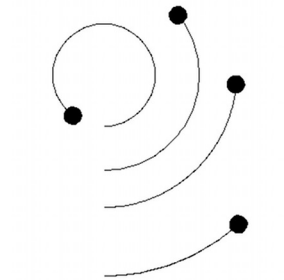
CS.16 Using Turtle graphics, implement function `spiral()` that draws a spiral with given initial length, angle, and multiplier.

CS.17 Using Turtle graphics, implement function `planets()`, which will simulate the planetary motion of Mercury, Venus, Earth, and Mars during one rotation of planet Mars. You can assume that:

- (a) At the beginning of the simulation, all planets are lined up (say along the negative y -axis).
- (b) The distances of Mercury, Venus, Earth, and Mars from the Sun (the center of rotation) are 58, 108, 150, and 228 pixels.

- (c) For every 1 degree circular motion of Mars, Earth, Venus, and Mercury will move 2, 3, and 7.5 degrees, respectively.

The figure below shows the state of the simulation when Earth is about a quarter of the way around the Sun. Note that Mercury has almost completed its first rotation.



CS.4 Processing Image Files

In this case study, we explore processing images using Python. Images are typically stored as binary files rather than text files, and so image files are not processed using standard text file I/O tools. Instead, special-purpose I/O tools are required to read, write, and process images. By “processing” we mean modifying an image in some way; for example, re-sizing, rotating, or blurring an image or cutting part of an image and pasting the part into another would all be basic image-processing tasks.

We also use this case study to show how one installs Python modules that are not in the Python Standard Library so that, once installed, they can be imported just like a Standard Library module. The image-processing library we need to install is *Pillow*, which is located in the *Python Package Index*, the official third-party software repository for Python.

DETOUR



Python Package Index (PyPI)

The Python Standard Library contains Python modules that extend the core Python language with additional functions and classes. These functions and classes are mostly general-purpose rather than specific to a particular application domain. Python software that is meant to be used in a particular application domain such as image processing is not included in the Python Standard Library.

The Python Package Index, or PyPI for short, is a repository for open-source Python software developed by third parties. At the time of writing, PyPI includes about 50,000 packages containing Python software for a huge variety of application domains. The image-processing library Pillow is one of the packages in the PyPI repository. In order to install a PyPI package, you need to open the terminal window for your particular operating system and run the program `pip3`. The `pip3` program is a tool for installing and managing Python software; the `3` in `pip3` ensures that you are installing software compatible with Python 3 (rather than 2). While `pip3` can be used to install software from other depositories, the default is PyPI.

For example, to install the Pillow library you would need to use:

```
$ pip3 install Pillow
```

The command `install` used above is used when installing; other `pip3` commands include `uninstall` and `list` (to view already installed packages). For more information about PyPI, visit

<http://pypi.python.org>.

Class Image in Module PIL.Image

Assuming that the Pillow library has already been installed on your computer, we now use it to open a file containing an image. The library consists of several dozen modules contained in a *package* called PIL (which stands for Python Imaging Library). A *package* is a way to collect related modules. One of the modules in PIL is `PIL.Image`; we import it as follows:

```
>>> import PIL.Image
```

Note the dotted notation used to access module `PIL.Image` in package `PIL`.

We can now use the function `open()`, defined in the `PIL.Image` module, to open the JPEG file with absolute pathname, say, `/Users/me/montana.jpg`:

```
>>> im = PIL.Image.open('/Users/me/montana.jpg') File: montana.jpg
```

Function `open()` returns an object of type `Image`. This class is defined in the `PIL.Image` module. Among the methods supported by the class is method `show()` that displays on the screen the image we just opened:

```
>>> im.show()
```

You should see the gorgeous landscape of Glacier National Park in Montana.

Image Size, Format, and Mode

Before we explore some of the other methods supported by class `Image`, we obtain some information about the image we just opened:

```
>>> im.size
(1248, 936)
>>> im.format
'JPEG'
>>> im.mode
'RGB'
```

What these tell us is that the image is 1248 pixels wide and 936 pixels tall, that it is stored in a compressed format using JPEG compression, and that the color of each pixel is described using the RGB model (see also Table CS.3). Before we go into what this all means, we take a moment to discuss what kind of variables `size`, `format`, and `mode` are. They are variables associated with object `im`, an object of type `Image`. They are commonly referred to as *instance variables* and are accessed using dotted notation and the name of the object as the prefix. We discuss such variables in more depth in Section 8.1.

Usage	Explanation
<code>im.format</code>	String containing the image format (JPEG, GIF, TIFF, BMP, ...)
<code>im.mode</code>	String containing the image mode (RGB, CYMK, Grayscale, ...)
<code>im.size</code>	Tuple containing the width and height of the image in pixels

Table CS.3 Image object metadata. The metadata is stored in instance variables associated with the object.

Figure CS.5 illustrates how our image is represented as a grid of 1248×936 pixels. Associated with each pixel is a pair of coordinates (i, j) that identify the column i and the row j of the pixel. The columns are numbered from left to right, starting with 0, and the rows are numbered from top to bottom, also starting with 0.

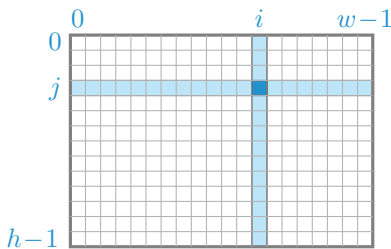


Figure CS.5 An image of size $w \times h$. A digital image is represented as a grid of pixels. Each pixel is assigned a color and can be identified by specifying a (column, row) pair. The pixel in column i and row j has location (i, j) .

Each pixel of an image is assigned a color. There are different ways to describe colors digitally and the *image mode* refers to a particular way, that is, a particular color encoding. The image we just opened has mode *RGB*, which means that colors are described using varying amounts of colors red, green and blue. Other image modes include mode *CMYK*, in which colors are described using varying amounts of cyan, magenta, yellow, and black, and also mode *L*, in which colors are described using shades of gray (for Grayscale images). We will take a closer look at the RGB color mode in Case Study CS.5.

The image *format* describes how the image file is organized and stored. While an image that has been opened for processing is represented by a grid of pixels, that is not necessarily how the image is stored in a file. Various formats exist today, and JPEG is just one of them. (JPEG stands for Joint Photographic Experts Group, the name of the committee of image-processing professionals that created the standard.) Other file formats that are fully supported by Pillow include BMP, EPS, GIF, PNG, and TIFF.

Image Class Methods

We now explore some of the methods of the `Image` class that allow us to manipulate images. Method `copy()` simply returns a copy of the image:

```
>>> copied = im.copy()
```

You can, of course, verify that `copied` is an exact copy of the original image:

```
>>> copied.show()
```

Method `rotate()` returns a copy of the image rotated counterclockwise by the given angle (in degrees).

```
>>> rotation = im.rotate(90)
```

If the angle argument is negative, then the image is rotated clockwise.

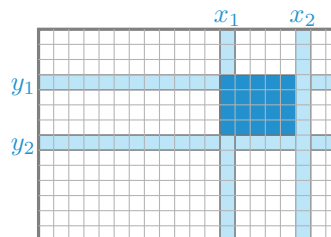
To crop our image `im`, we first need to define the rectangular region that we want to crop. To define the rectangular region of `im` from column x_1 on the left and up to but not including column x_2 on the right and from row y_1 on top and up to but not including row y_2 on the bottom, the 4-integer tuple (x_1, y_1, x_2, y_2) is used. For example

```
>>> box = (600, 600, 1000, 800)
```

would be used to define the 400×200 rectangular region whose upper left pixel is $(600, 600)$ and lower right pixel is $(999, 799)$. Once the rectangular region to be cropped has been defined we use method `crop()` to obtain a cropped copy of the image:

```
>>> cropped = im.crop(box)
```

Figure CS.6 Crop box. The rectangular region defined by tuple (x_1, y_1, x_2, y_2) goes from column x_1 on the left up to but not including column x_2 on the right and from row y_1 on top and up to but not including row y_2 on the bottom.



Usage	Explanation
<code>im.copy()</code>	Return copy of image <code>im</code>
<code>im.paste(im2, box)</code>	Paste image <code>im2</code> into image <code>im</code> at rectangular region defined by tuple <code>box</code>
<code>im.rotate(d)</code>	Return copy of image rotated <code>d</code> degrees counterclockwise
<code>im.show()</code>	Display image on the screen
<code>im.save(filename)</code>	Save image under given filename; the format of the saved file is determined from the filename extension
<code>im.crop(box)</code>	Return copy of the rectangular region defined by tuple <code>box</code>
<code>im.filter(fltr)</code>	Return copy of <code>im</code> filtered using filter <code>fltr</code>
<code>im.close()</code>	Close the file associated with the image

Table CS.4 Image class methods. Shown are some of the methods of class `Image`. For the full list use `help(PIL.Image.Image)` or the online documentation.

The method `filter()` returns a copy of the image that has been modified using an *image filter*. An image filter can sharpen an image or make it smoother, or it can provide more or less contrast, for example. Package PIL includes a module, `PIL.ImageFilter` that contains several predefined filters. One such filter is `SMOOTH`. We use it to obtain a smoother (i.e., less sharp) copy of the original image:

```
>>> import PIL.ImageFilter
>>> smoothed = im.filter(PIL.ImageFilter.SMOOTH)
```

You can then view the image obtained and compare it with the original. Other image filters defined in module `PIL.ImageFilter` include `SHARPEN` (essentially the opposite of `SMOOTH`), `CONTOUR`, `DETAIL`, `EMBOSS`, `FIND_EDGES`, etc.

Finally, method `paste()` is used to paste an image into another. In addition to the image to be pasted, the method takes another argument that specifies where the image should be pasted. The rectangular region is specified the same way as for cropping, using a tuple of four integers (see Figure CS.6). For example, we can paste `cropped`, the image of size 400×200 we cropped earlier, into the rectangular region of size 400×200 defined by `box` of image `smoothed` as follows:

```
>>> smoothed.paste(cropped, box)
```

The methods we have covered are summarized in Table CS.4.

Creating and Saving a New Image

Sometimes, rather than processing an existing image, we would like to create a new one. The function `new()` in module `PIL.Image` creates a new `Image` object with given mode and size (see Table CS.5). For example, the following creates a new RGB image of size 400×200 .

```
>>> im2 = PIL.Image.new('RGB', (400,200))
```

Usage	Explanation
<code>open(filename)</code>	Opens image file <code>filename</code> and returns <code>Image</code> object representing the image
<code>new(mode, size)</code>	Returns an <code>Image</code> object with given mode and size

Table CS.5 Some functions in module `PIL.Image`. Both return an `Image` object.

We can now choose to paste an existing image (or images) into this new image. For example, we can paste our cropped image:

```
>>> im2.paste(cropped, (0,0,400,200))
```

Finally, we can use the Image class method `save()` to save the new image:

```
>>> im2.save('crop.jpg')
```

This saves the image in JPEG format. Alternatively, we can save it in another format, say GIF, by simply changing the file extension from `.jpg` to `.gif`:

```
>>> im2.save('crop.gif')
```

We quickly verify that the saved image is indeed in a GIF format:

```
>>> im3 = PIL.Image.open('crop.gif')
>>> im3.format
'GIF'
```

DETOUR



More on the Python Imaging Library and Pillow

In this case study, we only scratched the surface of what one can do with images using Python and Pillow. The image-processing tools provided by Pillow include module `ImageEnhance` to adjust the contrast, brightness, and color balance of an image, module `PIL.ImageDraw` for basic 2D drawing, and module `PIL.ImageSequence` to process GIF animations, among others. For more information about Pillow check out the online documentation at

<http://pillow.readthedocs.org>.

Practice Problem CS.18

Implement function `warhol()` that takes an image (i.e., an Image object) as input and returns an image consisting of four copies of the input image, arranged in a 2×2 grid, modified as follows. The copy in the top left should be the original image, whereas the images in the top right, bottom left, and bottom right should be filtered using the `CONTOUR`, `EMBOSS`, and `FIND_EDGES` built-in filters defined in module `PIL.ImageFilter`.

Solution to the Practice Problem

CS.18 In order to contain four copies of the original image arranged in a 2×2 grid, the new image has to have twice the width and twice the height of the original. If w and h refer to the width and height of the original, the location in the new image where the top left copy of the original should be placed can be described using the tuple $(0, 0, w, h)$. Similarly, the top right, bottom left, and bottom right locations are described by tuples $(w, 0, 2w, h)$, $(0, h, w, 2h)$, and $(w, h, 2w, 2h)$, respectively.

```
def warhol(photo):
    '''returns image consisting of 4 copies of photo arranged
```



```

    in a 2x2 grid, with the top right, bottom left, and bottom
    right copies modified using filters CONTOUR, EMBOSS, and
    FIND_EDGES, respectively'''
width, height = photo.size # width, height = size[0], size[1]

# create new image big enough to contain 4 copies
# of photo arranged in a 2x2 grid pattern
res = PIL.Image.new(photo.mode, (2*width, 2*height))

# put original photo in top left corner of res
res.paste(photo, (0, 0, width, height))

# put CONTOUR filtered image in top right corner of res
contour = photo.filter(PIL.ImageFilter.CONTOUR)
res.paste(contour, (width, 0, 2*width, height))

# put EMBOSS filtered image in bottom left corner of res
emboss = photo.filter(PIL.ImageFilter.EMBOSS)
res.paste(emboss, (0, height, width, 2*height))

# put FIND_EDGES filtered image in bottom right corner of res
edges = photo.filter(PIL.ImageFilter.FIND_EDGES)
res.paste(edges, (width, height, 2*width, 2*height))

return res

```

Problems

CS.19 Open a file containing an image of yours and find its size, format, and mode.

CS.20 The class `Image` in module `PIL.Image` has a method `resize()` that takes a tuple (x, y) as input. When invoked on an `Image` object `im`, the method returns a resized copy of the image of size $x \times y$. Use this method to obtain a copy of image `montana.jpg` of size 312×234 .

File: `montana.jpg`

CS.21 Implement function `convert()` that takes the pathname (as a string) of a JPEG image file and saves a GIF format copy of the image in the current working directory.

CS.22 Open a file of yours containing an image with a group of people. Crop one of the faces and save it in a new file.

CS.23 Open a file of yours containing an image with a group of people. Create a new image in which two of the faces have been (cropped and) swapped.

CS.24 Implement function `album()` that takes a list of up to eight `Image` objects as input and returns an image containing all images arranged in a two column album (i.e., grid).

CS.25 Write a function `frame()` that takes as input an `Image` object and a tuple (w, h) of two integers. The function should crop the image so the result is the largest image whose width to height ratio is w/h and that is centered at the center of the original image.

CS.5 Image-Processing Algorithms

In Case Study CS.4 we learned how to process images using Python and third-party image-processing library Pillow. In particular, we saw how to copy, rotate, crop, and smooth an image. In this case study we take a look “underneath the hood” of image processing and see how such image-processing tools can be implemented. Because images are two-dimensional grids of pixels, it is no surprise that the nested loop pattern is going to be used quite a bit.

Accessing Pixels

In order to manipulate images, we need to be able to read and modify the color of individual pixels. Class `Image` (defined in module `PIL.Image`) provides methods `getpixel()` and `putpixel()` for that purpose (see Table CS.6). Method `getpixel()` takes the location of a pixel as input and returns its color. The location is specified with a tuple of two integers, and the color returned is a tuple representing the encoding of the color, where the encoding is determined by the image mode. In the case of the RGB mode, each RGB color is described with a tuple of three integer values in the range from 0 to 255 representing the amount of red, blue, and green colors used to produce the color. For example, after importing module `PIL.Image` and opening our working example,

File: montana.jpg

```
>>> import PIL.Image
>>> im = PIL.Image.open('/Users/me/montana.jpg')
```

we can obtain the color of the pixel at location `(1247, 0)` as follows:

```
>>> im.getpixel((1247,0))
(131, 194, 245)
```

The RGB color encoding `(131, 194, 245)` represents a sky blue color as seen in the upper right corner of the image. (The location `(1247, 0)` refers to the uppermost, rightmost pixel of our 1248×936 image.)

The method `putpixel()` takes the location of a pixel and a color as input and modifies the image by coloring the pixel with the given color. To change, in image `im`, the color of the pixel at location `(1247, 0)` to RGB color `(255, 0, 0)` (red) you would do:

```
>>> im.putpixel((125,175), (255,0,0))
```

View the image `im`

```
>>> im.show()
```

and note that the upper right pixel is now red. You may have to magnify the image to see the pixel!

We are now ready to see how image-processing functions can be implemented. We will focus, in particular, on functions that copy, rotate by 90 degrees, crop, and smooth out images.

Table CS.6 Additional Image class methods. These methods are used to access individual pixels.

Usage	Explanation
<code>im.getpixel(loc)</code>	Return the color of the pixel at location <code>loc</code>
<code>im.putpixel(loc, pix)</code>	Replace pixel color at location <code>loc</code> with color <code>pix</code>

Copying an Image

In our first example, we develop a function `copy()` that takes an image as input and returns a copy. This function should simply open the original image, create a new blank image of the same size, and then copy pixel colors from the old image to corresponding pixels in the new image. More precisely, the pixel (at location) (i, j) in the new image should receive the color of the pixel (i, j) in the original. In order to do this for all pixels, over all columns i and rows j of the image, a nested loop is clearly required.

```

1 def copy(original):
2     'returns copy of image photo'
3
4     # create a new image of the same mode and size as original
5     res = PIL.Image.new(original.mode, original.size)
6     # width = original.size[0], height = original.size[1]
7     width, height = original.size
8
9     # nested loop pattern to access individual pixels
10    for i in range(0, width):
11        for j in range(0, height):
12            # set pix to color of pixel
13            # at location (i,j) of original
14            pix = original.getpixel((i,j))
15            # set pixel at location (i,j) of res to pix
16            res.putpixel((i,j), pix)
17    return res

```

Module: image.py

This function can be used as follows:

```
>>> copied = copy(im)
```

You can view image copied to verify that the `copy()` function worked:

```
>>> copied.show()
```

Implement function `crop()` that takes, as input, an Image object and a tuple of four integers defining the rectangular region to be cropped (as shown in Figure CS.6) and returns a cropped copy of the image.

Practice Problem
CS.26

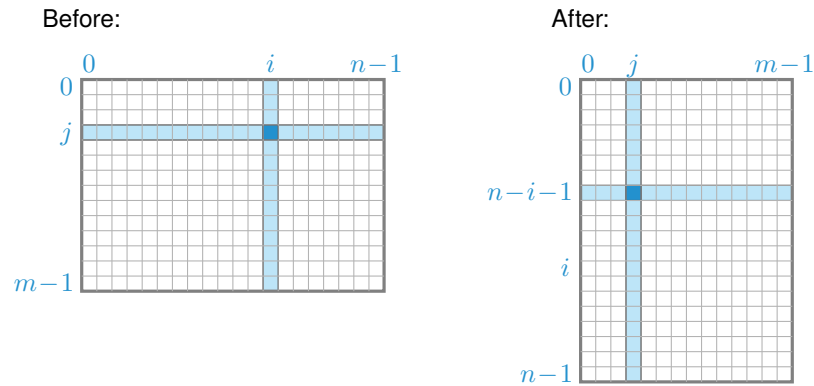
Rotating an Image by 90 Degrees

In our next example we implement a function that takes an image and returns a copy of it that is rotated 90 degrees counterclockwise. Just as for function `copy()`, colors of pixels need to be copied from the original to the new image. What we need to figure out is where in the new image should we put the color of pixel (i, j) of the original.

As Figure CS.7 illustrates, the rotation of the original $n \times m$ image results in a $m \times n$ image. Note that row j in the original image becomes column j in the new image. It is a little harder to see what happens to column i of the original image. To make sense of it, let's consider some special columns. Column 0 in the original image becomes the bottom row

Figure CS.7 Image

rotation. The rotation of an $n \times m$ image is of size $m \times n$. Row j in the original is mapped to column j of the rotated image, and column i is mapped to row $n - i - 1$. So pixel (i, j) of the original image is mapped to pixel $(j, n - i - 1)$ of the rotated image.



in the rotated image, that is, row $n - 1$. Similarly, column 1 becomes row $n - 2$, column 2 becomes row $n - 3$. In general, column i of the original image becomes row $n - i - 1$ in the rotated image. Therefore, the color of pixel (i, j) in the original is mapped to pixel $(j, n - i - 1)$ as shown in Figure CS.7.

Once we have figured out this mapping, the implementation of function `rotateCC()` is essentially the same as the implementation of function `copy()`:

Module: `image.py`

```

1  def rotateCC(original):
2      'returns copy of photo rotated counterclockwise 90 degrees'
3
4      # n and m are width and height, resp., of original
5      n, m = original.size
6      # m and n are width and height, resp., of new image
7      res = PIL.Image.new(original.mode, (m,n))
8
9      # nested loop pattern copies colors from original to res
10     for i in range(0, n):
11         for j in range(0, m):
12             pixel = original.getpixel((i,j))
13             # pixel at location (i,j) in original is
14             # mapped to pixel at location (j,n-i-1) in res
15             res.putpixel((j, n-i-1), pixel)
16     return res

```

This function can be used as follows:

```
>>> rotatedCC = rotateCC(im)
```

You may verify that the new image is the counterclockwise rotation of the original:

```
>>> rotatedCC.show()
```

Practice Problem CS.27

Implement function `rotateCL()` that takes an image (i.e., an `Image` object) as input and returns a copy of the image rotated clockwise 90 degrees.

Applying an Image Filter

Functions `copy()` and `rotateCC()` simply copy colors of pixels from the original image to pixels in the new image. By contrast, applying an image filter to an image produces an image that is not just another mapping of colors from the original to the new image. An image filter produces an image in which the color of pixel (i, j) is obtained by combining *in some way* the colors of pixels in the original image that are *close* to location (i, j) . By specifying what *close* and *in some way* mean exactly, different image filters can be defined and different image altering effects are obtained.

In Case Study CS.4, we used filter `SMOOTH` from module `PIL.ImageFilter` to make an image look smoother. An image looks smoother if there is less differentiation between neighboring pixels. One way to achieve this effect is by setting the color of every pixel (i, j) in the new image to the *average* of the colors of pixel (i, j) and its *neighbors* in the original.

We define what we mean by *average* of several colors first. We assume that we are working with RGB images, and thus the color of a pixel is encoded as a tuple of three integers representing the amount of red, green, and blue colors needed to produce the pixel color. The average of the colors of several pixels is simply the average amount of red, blue, and green in those pixels. We define the *neighbors* of pixel (i, j) to be those that are adjacent—vertically, horizontally, or diagonally—to it. Figure CS.8 shows the eight neighbors of pixel (i, j) , assuming that the pixel does not lie on the image border.

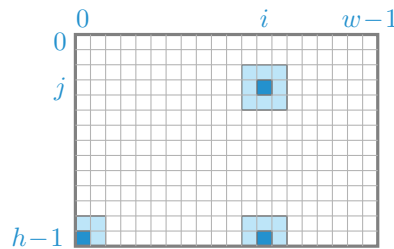


Figure CS.8 Neighbor pixels. The neighbors of pixel (i, j) are those that are adjacent to it vertically, horizontally, or diagonally. If pixel (i, j) does not lie on the image border, then it and its neighbors lie in columns $i - 1$ to $i + 1$ and rows $j - 1$ to $j + 1$.

Assuming that pixel (i, j) is not on the border, pixel (i, j) and its neighbors can be described as those that lie in columns $i - 1$ to $i + 1$ and rows $j - 1$ to $j + 1$. With this insight, we can write the code fragment that computes the color of pixel (i, j) in new image `res` as the average of the colors of pixel (i, j) and its neighbors in the image `original`:

```
# initialize sums of colors red, green and blue
red, green, blue = 0, 0, 0
# nested loop pattern generates locations
# (c, r) of pixel (i, j) and its neighbors
for c in range(i-1, i+2):
    for r in range(j-1, j+2):
        # add colors of pixel (c, r) to sums red, green, and blue
        pixel = original.getpixel((c, r))
        red = red + pixel[0]
        green = green + pixel[1]
        blue = blue + pixel[2]
# compute average of red, blue, and green colors
red, green, blue = red//9, green//9, blue//9
# color pixel (i, j) of res with average color
res.putpixel((i, j), (red, green, blue))
```

In this code, we separately add the amounts of red, green, and blue colors in pixel (i, j) and its neighbors in image original. Since pixel (i, j) is not on the border, a total of 9 pixels are involved in the three sums. Therefore, the average amount of red, green, and blue is obtained by dividing each of the three sums by 9.

The previous code is correct only for computing the color of pixels in res that are not on the image border. Figure CS.8 shows two cases when the pixel lies on the image border and not all eight pixel neighbors are defined. These special cases are exactly those for which the code fragment generates invalid row and column indexes. For example, for the pixel $(0, h - 1)$ in the lower right corner, the code would generate column -1 and row h , both of which are not valid. To ensure that invalid column indexes are not generated, we replace

```
for c in range(i-1, i+2):
```

with

```
for c in range(max(0, i-1), min(width, i+2)):
```

This second loop will not produce invalid column indexes -1 and w . A similar change for row indexes is shown in the complete implementation of function `smooth()`:

Module: image.py

```

1  def smooth(original):
2      'returns smooth copy of original'
3      # new image has same mode and size as original
4      res = PIL.Image.new(original.mode, original.size)
5      width, height = original.size
6      # nested loop pattern computes the color of every pixel in res
7      for i in range(0, width):
8          for j in range(0, height):
9              # initialize sums of colors red, green and blue
10             red, green, blue = 0, 0, 0
11             # initialize counter of neighbors of pixel (i, j)
12             numPixels = 0
13             # nested loop pattern generates locations
14             # (c, r) of pixel (i, j) and its neighbors
15             for c in range(max(0, i-1), min(width, i+2)):
16                 for r in range(max(0, j-1), min(height, j+2)):
17                     # increment neighbor count
18                     numPixels += 1
19                     # add colors of pixel (c, r) to sums
20                     # red, green, blue
21                     pixel = original.getpixel((c, r))
22                     red = red + pixel[0]
23                     green = green + pixel[1]
24                     blue = blue + pixel[2]
25             # compute average of red, green, and blue colors
26             red = red//numPixels
27             green = green//numPixels
28             blue = blue//numPixels
29             # color pixel (i, j) of res with average color
30             res.putpixel((i, j), (red, green, blue))
31     return res

```

Since the number of neighbors of a pixel varies, we had to replace the constant 9 in the average computation in lines 26 to 28 by the actual number of pixels involved in the three sums red, green, and blue. The actual number is obtained by incrementing numPixels for every pixel involved in the sums.

Solutions to Practice Problems

CS.26 This function takes as input a tuple (x_1, y_1, x_2, y_2) that defines the rectangular region to be cropped. The width and height of this region are $x_2 - x_1$ and $y_2 - y_1$, respectively. Therefore, the image res created and returned by the function should have those dimensions. The pixel at location (i, j) of res should receive the color of the pixel at location $(x_1 + i, y_1 + j)$ of the original image.

```
def crop(original, box):
    '''returns copy of image original cropped using
       the rectangular region defined by box'''
    # rows and columns that define region to be cropped
    x1, y1, x2, y2 = box
    # width and height of new image
    width, height = x2 - x1, y2 - y1
    # create new image to contain cropped image
    res = PIL.Image.new(original.mode, (width, height))
    # nested loop pattern copies pixel colors from original to res
    for i in range(width):
        for j in range(height):
            # set pixel at location (i, j) of res to color of
            # pixel at location (x1+i, y1+j) of original
            pix = original.getpixel((x1+i, y1+j))
            res.putpixel((i, j), pix)
    return res
```

CS.27 The main problem is to figure out where in the new, rotated image we should put the color of pixel (i, j) of the original. Since column i in the original becomes row i in the rotated image and row j in the original becomes column $m - j - 1$, it follows that the color of pixel (i, j) of the original is mapped to pixel $(m - j - 1, i)$ of the rotated image.

```
def rotateCL(original):
    'returns copy of original image rotated clockwise 90 degrees'
    # n and m are width and height, resp., of original
    n, m = original.size
    # m and n are width and height, resp., of new image
    res = PIL.Image.new(original.mode, (m, n))
    # nested loop pattern copies colors from original to res
    for i in range(0, n):
        for j in range(0, m):
            pixel = original.getpixel((i, j))
            # pixel at location (i, j) in original is
            # mapped to pixel at location (m-j-1, i) in res
            res.putpixel((m-j-1, i), pixel)
    return res
```

Problems

CS.28 The negative of an RGB image is obtained by changing the color (i, j, k) of every pixel to $(255 - i, 255 - j, 255 - k)$. Implement function `negative()` that takes an `Image` object as input and returns a negative copy of it.

CS.29 Implement function `mirror()` that takes an `Image` object as input and returns a mirror image copy of it. In other words, the first and last columns are flipped, and so are the second and next to last columns, and so on.

CS.30 Implement function `rotate180()` that takes an `Image` object as input and returns a copy that has been rotated 180 degrees.

CS.31 In RGB mode, color black is encoded as tuple $(0, 0, 0)$, color white is encoded as $(255, 255, 255)$, and shades of gray between black and white are encoded as (x, x, x) for values of x between 0 and 255. Therefore, to make an RGB image Black & White, each pixel color must be modified to a shade of gray or, more precisely, to a tuple containing the same amount of red, green, and blue. One way to do that is to set that amount to the average of colors red, green, and blue in the original pixel. Using this scheme, implement function `blackAndWhite()` that takes an `Image` object as input and returns a Black & White copy of it.

CS.32 Our implementation of function `smooth()` in this case study is different from the implementation of the built-in filter `SMOOTH` in module `PIL.ImageFilter`. In our implementation, the color of pixel (i, j) in the new image is the average of the colors of pixel (i, j) and neighboring pixels in the original image. Each pixel is given equal weight when computing the average. The built-in filter `SMOOTH` gives higher weight to pixel (i, j) : Pixel (i, j) is given a weight of five, whereas all neighboring pixels have the (usual) weight of one. Implement function `smooth2()` that uses this weighing in computing the average.

CS.33 The image filter `SHARPEN` is a built-in filter in module `PIL.ImageFilter` that sharpens an image, an effect that is essentially the opposite from the effect of filter `SMOOTH`. Like filter `SMOOTH` (see Problem CS.32), filter `SHARPEN` produces an image in which the color of pixel (i, j) is a weighted average of colors of pixel (i, j) and its neighbors in the original image: pixel (i, j) is assigned weight 32 while neighboring pixels as assigned weight -2 . Implement function `sharpen()` that takes an `Image` object as input and returns a copy that has been sharpened.

CS.34 The image filter `BLUR` is a built-in filter in module `PIL.ImageFilter` that blurs an image. Like filter `smooth()` that we developed in this case study, filter `BLUR` produces an image in which the color of pixel (i, j) is the average of colors of certain pixels surrounding pixel (i, j) . The pixels used to compute the average are those that are *neighbors of the neighbors* of pixel (i, j) but *not including* pixel (i, j) and its neighbors. Using this approach, implement function `blur()` that takes an `Image` object and returns a blurred copy of it.

CS.6 Games of Chance

Games of chance such as poker and blackjack have transitioned to the digital age very successfully. In this case study, we show how to develop a blackjack application. As we develop this application, we will make use of several concepts introduced in Chapter 6: sets, dictionaries, Unicode characters, and of course randomness through card shuffling.

Blackjack

Blackjack can be played with a standard deck of 52 cards. In a one-person blackjack game, the player plays against *the house* (i.e., the dealer). The house deals the cards, plays using a fixed strategy, and wins in case of a tie. Our blackjack application will simulate the house. The player (i.e., the user of the application) is trying to beat the house.

Blackjack Game Rules

The game starts with the house dealing cards from a shuffled deck to the player and to itself (the house). The first card is handed to the player, the second to the house, the third to the player, and the fourth to the house. Each gets two cards. Then the player is offered the opportunity to take an additional card, which is usually referred to as “to hit.”

The goal in blackjack is to get a hand whose card values add up to as close to 21 as possible without going over. The value of each number card is its rank, and the value of each face card is 10. The ace is valued at either 1 or 11, whichever works out better (i.e., gets closer to 21 without exceeding it). By hitting one or more times, the player tries to get his hand value closer to 21. If the player’s hand value goes over 21 after a hit, he loses.

When the player decides to *stand* (i.e., pass the opportunity to hit), it is the house’s turn to take additional cards. The house is required to use a fixed strategy: It must hit if its best hand value is less than 17 and stand if it is 17 or greater. Of course, if the house’s best hand value goes over 21, the player wins.

When the house stands, the player’s hand is compared with the house’s hand. If the player has a higher-valued hand, he wins. If he has a lower-valued hand, he loses. In case of a tie, no one wins (i.e., the player keeps his bet) except if the player’s and the house’s hands tied at 21 and either the house or the player has a *blackjack* hand (an ace and a value 10 card), in which case the blackjack hand wins.

DETOUR



Let’s illustrate with a few examples how we want the blackjack application to work. When you start the app, the house should deal two cards to you and two to itself:








```
>>> blackjack()
House:  7 ♠    A ♥
You:    6 ♠    10 ♠
Would you like to hit (default) or stand?
```

The house dealt a 6 and a 10 of spades to you and a 7 of spades and an ace of heart to itself. The house then asks you whether you want to hit. Suppose you hit:

```
You got  8 ♣
```







You went over... You lose.

You receive an 8 of clubs; since your hand value is $10 + 8 + 6 > 21$, you lose. Let's try another example:

```
>>> blackjack()
House:    5     7 
You:      2     8 
Would you like to hit (default) or stand?
You got    9 
Would you like to hit (default) or stand? s
House got   A 
House got   5 
You win.
```

After getting your first two cards, you decide to hit and receive a 9 of clubs. With a hand value of 19, you decide to stand. The house then hits once and gets an ace. With an ace value of 11, the house's total is $5+7+11=23$, so the ace value of 1 is used instead, making the house's hand value $5 + 7 + 1 = 13$. Since $13 < 17$, the house must hit again and gets a 5. With a hand value of 18, the house must stand and the player wins.

In this final example, the house loses because it went over:

```
>>> blackjack()
House:    2     10 
You:      4     8 
Would you like to hit (default) or stand?
You got    A 
Would you like to hit (default) or stand? s
House got   10 
House went over... You win.
```

Rather than develop the blackjack application as a single function, we develop it in modular fashion, using several small functions. The modular approach has two main benefits. One is that smaller functions are easier to write, test, and debug. Another is that some of the functions may be reused in some other card-playing game app. In fact, the first function we implement returns a shuffled deck, which is useful in most card games.

Creating and Shuffling the Deck of Cards

The game starts with a shuffled deck of 52 cards. Each card is defined by its rank and suit, and every combination of rank and suit defines a card. To generate all 52 combinations of rank and suit, we first create a set of ranks and a set of suits (using Unicode suit characters). Then we use a nested loop pattern to generate every combination of rank and suit. Finally, we use the `shuffle()` function of the `random` module to shuffle the deck:

Module: blackjack.py

```
1 def shuffledDeck():
2     'returns shuffled deck'
3
4     # suits is a set of 4 Unicode symbols: black spade and club,
5     # and white diamond and heart
6     suits = {'\u2660', '\u2661', '\u2662', '\u2663'}
7     ranks = {'2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A'}
```

```

8     deck = []
9
10    # create deck of 52 cards
11    for suit in suits:
12        for rank in ranks:                # card is the concatenation
13            deck.append(rank+' '+suit)    # of suit and rank
14
15    # shuffle the deck and return
16    random.shuffle(deck)
17    return deck

```

A list is used to hold the cards in the shuffled deck because a list defines an ordering on the items it contains. A blackjack hand, however, does not need to be ordered. Still, we choose lists to represent the player's and the house's hands. Next we develop a function that deals a card to either the player or the house.

Dealing a Card

The next function is used to deal the top card in a shuffled deck to one of the blackjack participants. It also returns the card dealt.

```

1  def dealCard(deck, participant):
2      'deals single card from deck to participant'
3      card = deck.pop()
4      participant.append(card)
5      return card

```

Module: blackjack.py

Note that this function can also be reused in other card game apps. The next function, however, is blackjack specific.

Computing the Value of a Hand

Next we develop the function `total()` that takes a blackjack hand (i.e., a list of cards) and uses the best assignment of values to aces to return the best possible value for the hand. Developing this function makes sense, not because it can be reused in other card games but because it encapsulates a very specific and somewhat complex computation.

The mapping of cards to their blackjack values is somewhat tricky. Therefore, we use a dictionary to map the assignments of values to the ranks (dictionary keys) with the ace being assigned 11. The hand value is computed with these assignments using an accumulator loop pattern. In parallel, we also count the number of aces, in case we want to switch the value of an ace down to 1.

If the hand value obtained is 21 or below, it is returned. Otherwise, the value of each ace in the hand, if any and one by one, is converted to 1 until the hand value drops below 21.

```

1  def total(hand):
2      'returns the value of the blackjack hand'
3      values = {'2':2, '3':3, '4':4, '5':5, '6':6, '7':7, '8':8,
4               '9':9, '10':10, 'J':10, 'Q':10, 'K':10, 'A':11}
5      result = 0
6      numAces = 0

```

Module: blackjack.py

```

7
8     # add up the values of the cards in the hand
9     # also add the number of aces
10    for card in hand:
11        result += values[card[0]]
12        if card[0] == 'A':
13            numAces += 1
14
15    # while value of hand is > 21 and there is an ace
16    # in the hand with value 11, convert its value to 1
17    while result > 21 and numAces > 0:
18        result -= 10
19        numAces -= 1
20
21    return result

```

Comparing the Player's and the House's Hands

Another part of the blackjack implementation that we can develop as a separate function is the comparison between the player's hand and the house's. Blackjack rules are used to determine, and announce, the winner.

Module: blackjack.py

```

1  def compareHands(house, player):
2      'compares house and player hands and prints outcome'
3
4      # compute house and player hand total
5      houseTotal, playerTotal = total(house), total(player)
6
7      if houseTotal > playerTotal:
8          print('You lose.')
9      elif houseTotal < playerTotal:
10         print('You win.')
11     elif houseTotal == 21 and 2 == len(house) < len(player):
12         print('You lose.') # house wins with a blackjack
13     elif playerTotal == 21 and 2 == len(player) < len(house):
14         print('You win.') # player wins with a blackjack
15     else:
16         print('A tie.')

```

Main Blackjack Function

We now implement the main function, `blackjack()`. The functions we have developed so far make the program easier to write and easier to read as well.

Module: blackjack.py

```

1  def blackjack()
2      'simulates the house in a game of blackjack'
3
4      deck = shuffledDeck() # get shuffled deck

```

```

5
6     house = [] # house hand
7     player = [] # player hand
8
9     for i in range(2): # dealing initial hands in 2 rounds
10         dealCard(deck, player) # deal to player first
11         dealCard(deck, house) # deal to house second
12
13     # print hands
14     print('House:{>7}{>7}'.format(house[0] , house[1]))
15     print('  You:{>7}{>7}'.format(player[0], player[1]))
16
17     # while user requests an additional card, house deals it
18     answer = input('Hit or stand? (default: hit): ')
19     while answer in {'', 'h', 'hit'}:
20         card = dealCard(deck, player)
21         print('You got{>7}'.format(card))
22
23         if total(player) > 21: # player total is > 21
24             print('You went over... You lose.')
25             return
26
27         answer = input('Hit or stand? (default: hit): ')
28
29     # house must play the "house rules"
30     while total(house) < 17:
31         card = dealCard(deck, house)
32         print('House got{>7}'.format(card))
33
34         if total(house) > 21: # house total is > 21
35             print('House went over... You win.')
36             return
37
38     # compare house and player hands and print result
39     compareHands(house, player)

```

In lines 6 and 7, the shuffled deck is used to deal the initial hands, which are then printed. In lines 18 to 25, the interactive loop pattern is used to implement the player's requests for additional cards. After each card is dealt, the value of the player's hand is checked. Lines 30 to 36 implement the house rule for completing the house hand.

Problems

CS.35 Most casinos combine multiple decks for blackjack. Modify the `shuffledDeck()` function so it shuffles six decks of 52 cards and returns a shuffled deck (list) of 312 cards.

CS.36 Implement function `pokerHand()` that takes a poker hand (i.e., a list of five strings such as '7 ♠') and returns the type of poker hand. The poker hand types include a straight flush, four of a kind, full house, flush, straight, three of a kind, two pair, one pair, and high

card.

```
>>> pokerHand(['7 ♠', '8 ♠', '9 ♠', '10 ♠', 'J ♠'])
Straight Flush
```

CS.37 Implement function `pokerComp()` that takes two poker hands (i.e., lists of five strings such as '7 ♠') and returns -1 if the first hand is stronger, 1 if the second is stronger, and 0 if neither is stronger than the other.

```
>>> hand1 = ['7 ♠', '8 ♠', '9 ♠', '10 ♠', 'J ♠']
>>> hand2 = ['4 ♦', '4 ♣', '4 ♠', '4 ♥', 'K ♠']
>>> pokerHand(hand1, hand2)
-1
```

CS.38 Implement function `blackjackProb()` that takes as input an integer n and simulates n rounds of blackjack in which the player uses the house strategy. You should reuse and appropriately modify the functions from this section.

CS.7 Debugging with a Debugger

In this case study we illustrate the use of a debugger to analyze the execution of a program and, if necessary, find bugs in it. The debugger we use is `pdb`, the python debugger that is included in the Python Standard Library. While fancier Python debuggers do exist, `pdb` is simple, has the features one expects in a debugger, and, because it is part of the Standard Library, has the advantage of always being around when working on Python code.

To illustrate how `pdb` is used, we make use of functions `f()`, `g()`, and `h()` that we defined in Section 7.1:

```

1 def h(n):
2     print('Start h')
3     print(1/n)
4     print(n)
5
6 def g(n):
7     print('Start g')
8     h(n-1)
9     print(n)
10
11 def f(n):
12     print('Start f')
13     g(n-1)
14     print(n)
```

Module: stack.py

We use `pdb` to analyze the execution of function call `f(2)` which, as we saw in Section 7.3, results in a `ZeroDivisionError` exception being thrown. To get started, we first import module `pdb` and then call function `run()` defined in the module:

```

>>> import pdb
>>> pdb.run('f(2)')
> <string>(1)<module>()
(Pdb)
```

Function `run()` takes as input the Python expression or statement whose execution we want to analyze; in our case it is the function call `f(2)`. The expression or statement must be given as a string, just as for function `eval()`. The result of executing `run()` is a new, interactive debugging session which is indicated by the `(Pdb)` prompt. This prompt tells us that the debugger is expecting a debugging command from us.

Debugging Commands

A basic debugging task is to run a program and stop its execution at a particular statement in order to inspect the variable values just before the statement is executed. To do this, we must first tell the debugger where to stop, which is done by associating a *breakpoint* with the statement using the `pdb` command `break`. We consider first the special case when we want to stop the execution just before a function is executed, that is, just before the first statement in the body of the function. For example, to set a breakpoint at the beginning of the body of function `f`, we use:

```

(Pdb) b f
Breakpoint 1 at /Users/me/stack.py:11
```

Table CS.7 Debugger commands. A few pdb debugger commands are shown; to see the full list, use the `help debugger` command.

Usage	Explanation
<code>b(break) func</code>	Set a break at the first statement of function <code>func</code>
<code>b(break) lineno</code>	Set a breakpoint at line numbered <code>lineno</code> of the program
<code>b(break)</code>	List all breakpoints
<code>condition bp cd</code>	Associate condition <code>cd</code> with breakpoint <code>bp</code> so it is ignored if <code>cd</code> evaluates to <code>False</code>
<code>c(continue)</code>	Continue execution until next breakpoint (or end of program)
<code>cl(ear) bp</code>	Clear breakpoint <code>bp</code> or all breakpoints if no argument is given
<code>d(own)</code>	Move the program stack frame view to the frame one level down (as seen in the program trace)
<code>h(elp)</code>	Print the list of available commands
<code>l(ist)</code>	List the 11 lines of source code around the current line
<code>p exp</code>	Evaluate expression <code>exp</code> and print its value
<code>q(uit)</code>	Quit the debugger
<code>n(ext)</code>	Execute the next line of the current function and stop when done
<code>u(p)</code>	Move the program stack frame view to the frame one level up (as seen in the program trace)
<code>w(here)</code>	Print the stack trace, with the most recent frame at the bottom

Note the printed message saying that the breakpoint is associated with the statement in line 11 of module `stack.py`. We use `b`, the abbreviated version of command `break`; since debugging often involves typing a lot of debugging commands, to make debugging more efficient most commands have abbreviated versions. Table CS.7 lists the commands, and their abbreviations, we will use in this case study.

Once the breakpoint is set, we can run the program within the debugger using the `c(ontinue)` command:

```
(Pdb) c
> /Users/me/stack.py(12)f()
-> print('Start f')
```

The program executes until a breakpoint is reached or the program terminates. In our case the execution stops at the breakpoint we just set, just before statement `print('Start f')` which is the first statement in the body of function `f`. This is shown in the printed message with an arrow pointing to that statement. We can also see that with command `l(ist)` which prints the 11 lines of source code around the statement at which the execution stopped:

```
(Pdb) l
7      print('Start g')
8      h(n-1)
9      print(n)
10
11 B    def f(n):
12 ->   print('Start f')
13       g(n-1)
14       print(n)
15
16
[EOF]
```


At any point, we can make queries about variable values using command `p(rint)`. For example, we can verify the current value of variable `n`:

```
(Pdb) p n
2
```

Let's continue the execution of the program. To execute just a single statement (the one pointed to by the arrow) and stop, we can use command `n(ext)`:

```
(Pdb) n
Start f
> /Users/me/stack.py(13)f()
-> g(n-1)
```

Note that the string `'Start f'` was printed, which means that statement `print('Start f')` was executed. In addition, a message is printed that shows the arrow pointing to function call `g(n-1)`, which means that execution has stopped just prior to executing this statement.

Suppose that at this point we want to continue execution and stop just before the body of function `g` is executed. We simply set another breakpoint at `g` and then continue execution:

```
(Pdb) b g
Breakpoint 2 at /Users/me/stack.py:6
(Pdb) c
> /Users/me/stack.py(7)g()
-> print('Start g')
```

Note that the execution stopped just before executing the first statement of `g()`. The value of `n` is now:

```
(Pdb) p n
1
```

Before we continue the execution, we set another breakpoint. This time it will be at function `h`:

```
(Pdb) b h
Breakpoint 3 at /Users/me/stack.py:1
```

We also take a moment to see all the breakpoints we have set:

```
(Pdb) b
Num Type      Disp Enb  Where
1  breakpoint keep yes   at /Users/me/stack.py:11
    breakpoint already hit 1 time
2  breakpoint keep yes   at /Users/me/stack.py:6
    breakpoint already hit 1 time
3  breakpoint keep yes   at /Users/me/stack.py:1
```

The `pdb` command `b(reak)`, when used without arguments, lists the current breakpoints. We now continue the execution using the command `c(ontinue)`:

```
(Pdb) c
Start g
> /Users/me/stack.py(2)h()
-> print('Start h')
```

We have stopped at the breakpoint we set at function `h`. We take one more step and then verify the value of `n`:

```
(Pdb) n
Start h
> /Users/me/stack.py(3)h()
-> print(1/n)
(Pdb) p n
0
```

The value of `n` while executing function `h` is 0 and the next statement in `h` to be executed is `print(1/n)`. Before we proceed and execute that statement, we take this opportunity and show how to analyze the current program stack.

Analyzing the Program Stack

The command `w`(here) prints the *stack trace* which is a summary of the current program stack.

```
(Pdb) w
/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/
bdb.py(431)run()
-> exec(cmd, globals, locals)
<string>(1)<module>()
/Users/me/stack.py(13)f()
-> g(n-1)
/Users/me/stack.py(8)g()
-> h(n-1)
> /Users/me/stack.py(3)h()
-> print(1/n)
```

A two-line summary of the frame at the *top* of the program stack, corresponding to the most recent function call and marked with symbol `>`, is shown at the *bottom*. The reason for this inconsistency (the top frame in the stack is shown at the bottom of the stack trace) is that on Intel-CPU-based systems, the convention is to draw the stack upside-down. The stack trace shows that the frame at the top of the stack corresponds to function `h()` and that we are about to execute statement `print(1/n)` in `h()`.

The stack trace also shows that the stack frame just below the top frame corresponds to the call to function `g()` and that we are in the process of executing function call `h(n-1)` in `g()`, and that the frame below that corresponds to the call to function `f()` and that we are in the process of executing function call `g(n-1)` in `f()`.

Debugger `pdb` provides commands that allow us to go up and down the program stack and visit frames currently in the stack. The command `u(p)` is used to move from the frame we are currently viewing to the one above it as shown in the program trace. Let's use this command to move up from the top frame corresponding to the call to function `h()` and in which variable `n` has value 0:

```
(Pdb) u
> /Users/me/stack.py(8)g()
-> h(n-1)
```

The message shows that we are now viewing the frame corresponding to the execution of function `g()`. We verify this using command `l(ist)` and by printing the value of `n`.

```

(Pdb) l
3         print(1/n)
4         print(n)
5
6 B    def g(n):
7         print('Start g')
8 ->    h(n-1)
9         print(n)
10
11 B    def f(n):
12         print('Start f')
13         g(n-1)
(Pdb) p n
1

```

Let's move up one more frame to view the frame corresponding to the call to function `f()`:

```

(Pdb) u
> /Users/me/stack.py(13)f()
-> g(n-1)
(Pdb) p n
2

```

The value of `n` in this frame is 2. In Figure CS.9, we illustrate the different values of `n` stored in different frames of the stack.

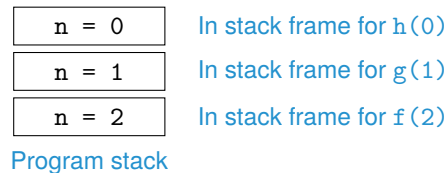


Figure CS.9 Current stack. Shown are the different values of `n` stored in frames on the program stack.

The command `d(own)` is used to move from the frame we are currently viewing to the one below it as shown in the program trace. We use twice to get back to the top (i.e., most recent) frame on the stack:

```

(Pdb) d
> /Users/me/stack.py(8)g()
-> h(n-1)
(Pdb) d
> /Users/me/stack.py(3)h()
-> print(1/n)

```

The printed message shows that we are again viewing the frame corresponding to the call to function `h` and showing that the execution stopped just prior to executing the statement `print(1/n)`. Since the value of `n` in this frame is 0, the execution of the statement will cause a `DivisionByZero` exception to be thrown, crashing our program:

```

(Pdb) n
ZeroDivisionError: division by zero
> /Users/me/stack.py(3)h()
-> print(1/n)

```

We can now use command `c(ontinue)` to continue the execution of the program which will result in the stack trace being printed and the program terminating. Alternatively, we can simply quit the debugger using command `q(uit)`:

```
(Pdb) q
```

Practice Problem CS.39

Module: arithmetic.py

Function `arithmetic` is supposed to take a list of integers as input and return `True` if the numbers in the list form an arithmetic sequence and `False` otherwise:

```
1 def arithmetic(numList):
2     '''returns True if list of integers numList
3         is an arithmetic sequence, False otherwise'''
4     if len(numList) < 2:
5         return True
6     diff = numList[1] - numList[0]
7     for i in range(len(numList)):
8         if numList[i+1] - numList[i] != diff:
9             return False
10    return True
```

This implementation has a bug, however:

```
>>> arithmetic([2,4,6,8])
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    arithmetic([2,4,6,8])
  File "/Users/me/stack.py", line 8, in arithmetic
    if numList[i+1] - numList[i] != diff:
IndexError: list index out of range
```

Using the `pdb` debugger, find the value of `i` that causes the `IndexError` exception. To do this, you will find it helpful to read, in the online Python Standard Library `pdb` documentation, about setting a conditional breakpoint so execution stops at a breakpoint only when a condition is true.

Solution to the Practice Problem

CS.39 Let's start the debugger session, set a breakpoint at function `arithmetic`, execute the program up to this breakpoint, and list the source code:

```
>>> import pdb
>>> pdb.run('arithmetic([2,4,6,8])')
> <string>(1)<module>()
(Pdb) b arithmetic
Breakpoint 1 at /Users/me/stack.py:1
(Pdb) c
> /Users/me/stack.py(2)arithmetic()
-> if len(numList) < 2:
```

```
(Pdb) l
1 B   def arithmetic(numList):
2 ->   if len(numList) < 2:
3       return True
4       diff = numList[1] - numList[0]
5       for i in range(len(numList)):
6           if numList[i+1] - numList[i] != diff:
7               return False
8       return True
[EOF]
```

The stack trace shows that the `IndexError` occurred while executing the statement

```
if numList[i+1] - numList[i] != diff:
```

so we set a breakpoint at that line *and* we associate with it a condition that must be true for an `IndexError` exception to occur. i would either have to be less than 0 or $i + 1$ would have to be at least `len(numList)`:

```
(Pdb) b 6
Breakpoint 2 at /Users/me/stack.py:6
(Pdb) condition 2 i < 0 or i + 1 >= len(numList)
New condition set for breakpoint 2.
(Pdb) c
> /Users/me/stack.py(6)arithmetic()
-> if numList[i+1] - numList[i] != diff:
```

Since the execution stopped at line 6, it must be that the condition associated with the breakpoint is true. We check this:

```
(Pdb) p i
3
(Pdb) p i+1
4
(Pdb) p len(numList)
4
```

Problems

CS.40 Function `slopes()` takes as input a list of points in the plane and then computes and prints the slope between each successive pair of points. Each point is described using a tuple of two numbers.

```
1 def slopes(points):
2     for i in range(1, len(points)):
3         x1, y1 = points[i-1]
4         x2, y2 = points[i]
5         slope = (y2 - y1) / (x2 - x1)
```

Module: slopes.py

An error occurs when the function is run with the list `points` as input:

```
>>> points = [(2, 3), (5, 6), (4, 5), (4, 3), (2, 5), (9, 4), \
              (7, 4), (3, 7), (6, 8), (8, 5), (2, 9)]
```

```
>>> slopes(points)
Traceback (most recent call last):
  File "<pyshell#372>", line 1, in <module>
    slopes(points)
  File "/Users/me/slopes.py", line 5, in slopes
    slope = (y2 - y1) / (x2 - x1)
ZeroDivisionError: division by zero
```

Use the pdb debugger and a conditional breakpoint to quickly find the value of index *i* when the error occurs.

CS.41 The following program generates a secret six-digit key based on your computer's network name:

Module: blackbox.py

```
1  from random import randrange, seed
2  from platform import node
3
4  def blackbox():
5      seed(node()) # generate pseudorandom number generator seed
6                  # based on host's network name
7      secret = randrange(10**5, 10**6) # generate six digit integer
8                                          # based on seed
9
10     try:
11         key = int(input('Enter key: '))
12         if key != secret:
13             print('Failed...')
14         else:
15             print('Success!')
16     except:
17         print('Failed...')
```

Use the pdb debugger to find the secret key and test the key you found. On the author's laptop, the key 892854 worked:

```
>>> blackbox()
Enter key: 892854
Success!
```

CS.8 Indexing and Iterators

In this case study, we will learn how to make a container class feel more like a built-in class. We will see how to enable indexing of items in the container and how to enable iteration, using a for loop, over the items in the container.

Because iterating over a container is an abstract task that generalizes over different types of containers, software developers have developed a general approach for implementing iteration behavior. This approach, called the *iterator design pattern*, is just one among many OOP design patterns that have been developed and cataloged for the purpose of solving common software development problems.

Overloading the Indexing Operators

Suppose that we are working with a queue, whether of type `Queue` or `Queue2`, and would like to see what item is in the 2nd, 3rd, or 24th position in the queue. In other words, we would like to use the indexing operator `[]` on the queue object.

We implemented the class `Queue2` as a subclass of `list`. Thus `Queue2` inherits all the attributes of class `list`, including the indexing operator. Let's check that. We first build the `Queue2` object:

```
>>> q2 = Queue2()
>>> q2.enqueue(5)
>>> q2.enqueue(7)
>>> q2.enqueue(9)
```

Now we use the indexing operator on it:

```
>>> q2[1]
7
```

Let's now turn our attention to the original implementation, `Queue`. The only attributes of class `Queue` are the ones we implemented explicitly. It therefore should not support the indexing operator:

```
>>> q = Queue()
>>> q.enqueue(5)
>>> q.enqueue(7)
>>> q.enqueue(9)
>>> q
[5, 7, 9]
>>> q[1]
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    q[1]
TypeError: 'Queue' object does not support indexing
```

In order to be able to access `Queue` items using the indexing operator, we need to add method `__getitem__()` to the `Queue` class. This is because when the indexing operator is used on an object, as in `q[i]`, the Python interpreter will translate that to a call to method `__getitem__()`, as in `q.__getitem__(i)`; if method `__getitem__()` is not implemented, then the object's type does not support indexing.

Here is the implementation of `__getitem__()` we will add to class `Queue`:

```
def __getitem__(self, key):
    return self.q[key]
```

The implementation relies on the fact that lists support indexing: To get the queue item at index `key`, we return the item at index `key` of the list `self.q`. We check that it works:

```
>>> q = queue()
>>> q.enqueue(5)
>>> q.enqueue(7)
>>> q.enqueue(9)
>>> q[1]
7
```

OK, so we now can use the indexing operator to *get* the item of a Queue at index 1. Does this mean we can change the item at index 1?

```
>>> q[1] = 27
Traceback (most recent call last):
  File "<pyshell#48>", line 1, in <module>
    q[1] = 27
TypeError: 'queue' object does not support item assignment
```

That's a no. Method `__getitem__()` gets called by the Python interpreter only when we evaluate `self[key]`. When we attempt to assign to `self[key]`, the overloaded operator `__setitem__()` is called by the Python interpreter instead. If we wanted to allow assignments such as `q[1] = 27`, then we would have to implement a method `__setitem__()` that takes a key and an item as input and places the item at position key.

A possible implementation of `__setitem__()` could be:

```
def __setitem__(self, key, item):
    self.q[key] = item
```

This operation, however, does not make sense for a queue class, and we do not to add it.

One benefit of implementing the method `__getitem__()` is that it allows us to iterate over a Queue container, using the iteration loop pattern:

```
>>> for item in q:
    print(item)

5
7
9
```

Before implementing the method `__getitem__()`, we could not have done that.

Practice Problem CS.42

Recall that we can also iterate over a Queue container using the counter loop pattern (i.e., by going through the indexes):

```
>>> for i in range(len(q)):
    print(q[i])

3
5
7
9
```


What overloaded operator, in addition to the indexing operator, needs to be implemented to be able to iterate over a container using this pattern?

Iterators and OOP Design Patterns

Python supports iteration over all the built-in containers we have seen: strings, lists, dictionaries, tuples, and sets. We have just seen that by adding the indexing behavior to a user-defined container class, we can iterate over it as well. The remarkable thing is that the same iteration pattern is used for all the container types:

```
for c in s:      # s is a string
    print(char)

for item in lst: # lst is a list
    print(item)

for key in d:    # d is a dictionary
    print(key)

for item in q:   # q is a Queue (user-defined class)
    print(item)
```

The fact that the same code pattern is used to iterate over different types of containers is no accident. Iteration over items in a container transcends the container type. Using the same familiar pattern to encode iteration simplifies the work of the developer when reading or writing code. That said, because each container type is different, the work done by the `for` loop will have to be different depending on the type of container: Lists have indexes and dictionaries do not, for example, so the `for` loop has to work one way for lists and another way for dictionaries.

To explore iteration further, we go back to iterating over a `Queue` container. With our current implementation, iteration over a queue starts at the front of the queue and ends at the rear of the queue. This seems reasonable, but what if we really, really wanted to iterate from the rear to the front, as in:

```
>>> q = [5, 7, 9]
>>> for item in q:
    print(item)

9
7
5
```

Are we out of luck?

Fortunately, Python uses an approach to implement iteration that can be customized. To implement the *iterator pattern*, Python uses classes, overloaded operators, and exceptions in an elegant way. In order to describe it, we need to first understand how iteration (i.e., a `for` loop) works. Let's use the next `for` loop as an example:

```
>>> s = 'abc'
>>> for c in s:
    print(c)
```

```
a
b
c
```

What actually happens in the loop is this: The `for` loop statement causes the method `__iter__()` to be invoked on the container object (string `'abc'` in this case.) This method returns an object called an *iterator*; the iterator will be of a type that implements a method called `__next__()`; this method is then used to access items in the container one at a time. Therefore, what happens behind the scenes when the last `for` loop executes is this:

```
>>> s = 'abc'
>>> it = s.__iter__()
>>> it.__next__()
'a'
>>> it.__next__()
'b'
>>> it.__next__()
'c'
>>> it.__next__()
Traceback (most recent call last):
  File "<pyshell#173>", line 1, in <module>
    it.__next__()
StopIteration
```

After the iterator has been created, the method `__next__()` is called repeatedly. When there are no more elements, `__next__()` raises a `StopIteration` exception. The `for` loop will catch that exception and terminate the iteration.

In order to add custom iterator behavior to a container class, we need to do two things:

1. Add to the class method `__iter__()`, which returns an object of a iterator type (i.e., of a type that supports the `__next__()` method).
2. Implement the iterator type and in particular the method `__next__()`.

We illustrate this by implementing iteration on `Queue` containers in which queue items are visited from the rear to the front of the queue. First, a method `__iter__()` needs to be added to the `Queue` class:

Module: `queue.py`

```
1 class Queue:
2     'a classic queue class'
3
4     # other Queue methods implemented here
5
6     def __iter__(self):
7         'returns Queue iterator'
8         return QueueIterator(self)
```

The `Queue` method `__iter__()` returns an object of type `QueueIterator` that we have yet to implement. Note, however, that argument `self` is passed to the `QueueIterator()` constructor: In order to have an iterator that iterates over a specific queue, it better have access to the queue.

Now let's implement the iterator class `QueueIterator`. We need to implement the `QueueIterator` class constructor so that it takes in a reference to the `Queue` container

it will iterate over:

```
class QueueIterator:
    'iterator for Queue container class'
    def __init__(self, q):
        'constructor'
        self.q = q

    # method next to be implemented
```

The method `__next__()` is supposed to return the next item in the queue. This means that we need to keep track of what the next item is, using an instance variable we will call `index`. This variable will need to be initialized, and the place to do that is in the constructor. Here is the complete implementation:

```
1 class QueueIterator:
2     'iterator for Queue container class'
3     def __init__(self, q):
4         'constructor'
5         self.index = len(q)-1
6         self.q = q
7
8     def __next__(self):
9         '''returns next Queue item; if no next item,
10            raises StopIteration exception'''
11         if self.index < 0:          # no next item
12             raise StopIteration()
13
14         # return next item
15         res = self.q[self.index]
16         self.index -= 1
17         return res
```

Module: queue.py

The method `__next__()` raises an exception if there are no more items to iterate over. Otherwise, it stores the item at `index`, decrements `index`, and returns the stored item.

Develop subclass `oddList` of `list` that behaves just like a list except for the peculiar behavior of the `for` loop:

Practice Problem
CS.43

```
>>> lst = oddList(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
>>> lst
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> for item in lst:
    print(item, end=' ')
```

a c e g

The iteration loop pattern skips every other item in the list.

Solutions to Practice Problems

CS.42 The operator `len()`, which returns the length of the container, is used explicitly in a counter loop pattern.

CS.43 The class `oddList` inherits all the attributes of `list` and overloads the `__iter__()` method to return a `ListIterator` object. Its implementation is shown next.

```
class oddList(list):
    'list with peculiar iteration loop pattern'
    def __iter__(self):
        'returns list iterator object'
        return ListIterator(self)
```

An object of type `ListIterator` iterates over a `oddList` container. The constructor initializes the instance variables `lst`, which refers to the `oddList` container, and `index`, which stores the index of the next item to return:

```
class ListIterator:
    'peculiar iterator for oddList class'
    def __init__(self, lst):
        'constructor'
        self.lst = lst
        self.index = 0

    def __next__(self):
        'returns next oddList item'
        if self.index >= len(self.lst):
            raise StopIteration
        res = self.lst[self.index]
        self.index += 2
        return res
```

The `__next__()` method returns the item at position `index` and increments `index` by 2.

Problems

CS.44 Modify the solution of Practice Problem CS.43 so that two list items are skipped in every iteration of a `for` loop.

```
>>> lst = oddList(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
>>> for item in lst:
    print(item, end=' ')

a d g
```

CS.45 Implement an iterator for container class `Stat` from Problem 8.34. To do this, first add method `__iter__()` to class `Stat`; this method should just instantiate and return a `StatIterator` object. In order for this iterator object to be able to access the items contained in the corresponding `Stat` object, make sure you pass a reference to the list that

actually contains the items in the `Stat` object. Finally, implement class `StatIterator`.

CS.46 Add method `__iter__()` to class `PriorityQueue` from Problem 8.36; the method should just return a `PriorityQueueIterator` object. Then implement the iterator class `PriorityQueueIterator` that visits items of a `PriorityQueue` object in order, from smallest to largest.

CS.47 Implement an iterator for class `Deck` developed in Section 8.3.

CS.48 Implement an iterator for class `Hand` from Problem 8.29. Your iterator should visit the cards in the hand in order, where the order is determined by suit (clubs first, then diamonds, then hearts, and finally spades) and then by rank (2 through 10, and then J, Q, K, and A).

CS.9 Developing a Calculator

In this case study, we implement a basic calculator GUI, shown in Figure CS.10. We use OOP techniques to implement it as a user-defined widget class, from scratch. In the process, we explain how to write a single event-handling function that handles many different buttons.

Figure CS.10 GUI Calc. A calculator application with the usual four operators, a square root and a square function, and a memory capability.



The Calculator Buttons and Passing Arguments to Handlers

Let's get our hands dirty right away and tackle the code that creates the 24 buttons of the calculator. We can use the approach based on a two-dimensional list of button labels and a nested loop that we used in program `phone.py` from Section 9.1. Let's get started.

Module: `calc.py`

```

1  # calculator button labels in a 2D list
2  buttons = [['MC',      'M+',      'M-', 'MR'],
3             ['C',      '\u221a',  'x\u00b2', '+'],
4             ['7',      '8',        '9',  '-'],
5             ['4',      '5',        '6',  '*'],
6             ['1',      '2',        '3',  '/'],
7             ['0',      '.',        '+-', '=']]
8
9  # create and place buttons in appropriate row and column
10 for r in range(6):
11     for c in range(4):
12         b = Button(self,          # button for symbol buttons[r][c]
13                     text=buttons[r][c],
14                     width=3,
15                     relief=RAISED,
16                     command=???)    # method ??? to be done
17         b.grid(row=r+1, column=c)    # entry is in row 0

```

(We use Unicode characters `\u221a` and `\u00b2` for the square root and the superscript in x^2 .)

What’s missing in this code is the name of each event-handling function (note the question marks ??? in line 16). With 24 different buttons, we need to have 24 different event handlers. Writing 24 different handlers would not only be very painful, but it would also be quite repetitive since many of them are essentially the same. For example, the 10 handlers for the 10 “digit” buttons should all do essentially the same thing: append the appropriate digit to the string in the entry field.

Wouldn’t it be nicer if we could write just one event handler called `click()` for all 24 buttons? This handler would take one input argument, the label of the clicked button, and then handle the button click depending on what the label is.

The problem is that a button event handler cannot take an input argument. In other words, the `command` option in the `Button` constructor must refer to a function that can and will be called without arguments. So are we out of luck?

There is actually a solution to the problem, and it uses the fact that Python functions can be defined so that when called without an input value, the input argument receives a default value. Instead of having function `click()` be the official handler, we define, inside the nested `for` loop, the handler to be a function `cmd()` that takes one input argument `x`—which defaults to the label `buttons[r][c]`—and calls `self.click(x)`. The next module includes this approach (and the code that creates the `Entry` widget):

Module: `calc.py`

```

1  # use Entry widget for display
2  self.entry = Entry(self, relief=RIDGE, borderwidth=3,
3                    width=20, bg='gray',
4                    font=('Helvetica', 18))
5  self.entry.grid(row=0, column=0, columnspan=5)
6
7  # create and place buttons in appropriate row and column
8  for r in range(6):
9      for c in range(4):
10
11         # function cmd() is defined so that when it is
12         # called without an input argument, it executes
13         # self.click(buttons[r][c])
14         def cmd(x=buttons[r][c]):
15             self.click(x)
16
17         b = Button(self,          # button for symbol buttons[r][c]
18                   text=buttons[r][c],
19                   width=3,
20                   relief=RAISED,
21                   command=cmd)      # cmd() is the handler
22         b.grid(row=r+1, column=c)   # entry is in row 0

```

In every iteration of the innermost `for` loop, a new function `cmd` is defined. It is defined so that when called without an input value, it executes `self.clicked(buttons[r][c])`. The label `buttons[r][c]` is the label of the button being created in the same iteration. The button constructor will set `cmd()` to be the button’s event handler.

In summary, when the calculator button with label `key` is clicked, the Python interpreter will execute `self.click(key)`. To complete the calculator, we need only to implement the “unofficial” event handler `click()`.

Implementing the “Unofficial” Event Handler `click()`

The function `click()` actually handles every button click. It takes the text label `key` of the clicked button as input and, depending on what the button label is, does one of several things. If `key` is one of the digits 0 through 9 or the dot, then `key` should simply be appended to the digits already in the Entry widget:

```
self.entry.insert(END, key)
```

(We will see in a moment that this is not quite enough.)

If `key` is one of the operators `+`, `-`, `*`, or `/`, it means that we just finished typing an operand, which is displayed in the entry widget, and are about to start typing the next operand. To handle this, we use an instance variable `self.expr` that will store the expression typed so far, as a string. This means that we need to append the operand currently displayed in the entry box and also the operator `key`:

```
self.expr += self.entry.get()
self.expr += key
```

In addition, we need to somehow indicate that the next digit typed is the start of the next operand and should not be appended to the current value in the Entry widget. We do this by setting a flag:

```
self.startOfNextOperand = True
```

This means that we need to rethink what needs to be done when `key` is one of the digits 0 through 9. If `startOfNextOperand` is `True`, we need to first delete the operand currently displayed in the entry and reset the flag to `False`:

```
if self.startOfNextOperand:
    self.entry.delete(0, END)
    self.startOfNextOperand = False
self.entry.insert(END, key)
```

What should be done if `key` is `=`? The expression typed so far should be evaluated and displayed in the entry. The expression consists of everything stored in `self.expr` and the operand currently in the entry. Before displaying the result of the evaluation, the operand currently in the entry should be deleted. Because the user may have typed an illegal expression, we need to do all this inside a try block; the exception handler will display an error message if an exception is raised while evaluating the expression.

We can now implement a part of the `click()` function:

Module: `calc.py`

```
1 def click(self, key):
2     'handler for event of pressing button labeled key'
3     if key == '=':
4         # evaluate the expression, including the value
5         # displayed in entry and display result
6         try:
7             result = eval(self.expr + self.entry.get())
8             self.entry.delete(0, END)
9             self.entry.insert(END, result)
10            self.expr = ''
11        except:
12            self.entry.delete(0, END)
```



```

13         self.entry.insert(END, 'Error')
14
15     elif key in '+*-/':
16         # add operand displayed in entry and operator key
17         # to expression and prepare for next operand
18         self.expr += self.entry.get()
19         self.expr += key
20         self.startOfNextOperand = True
21     # the cases when key is '\u221a', 'x\u00b2', 'C',
22     # 'M+', 'M-', 'MR', 'MC' are left as an exercise
23
24     elif key == '+-':
25         # switch entry from positive to negative or vice versa
26         # if there is no value in entry, do nothing
27         try:
28             if self.entry.get()[0] == '-':
29                 self.entry.delete(0)
30             else:
31                 self.entry.insert(0, '-')
32         except IndexError:
33             pass
34
35     else:
36         # insert digit at end of entry, or as the first
37         # digit if start of next operand
38         if self.startOfNextOperand:
39             self.entry.delete(0, END)
40             self.startOfNextOperand = False
41         self.entry.insert(END, key)

```

Note that the case when the user types the +- button is also shown. Each press of this button should either insert a - operator in front of the operand in the entry if it is positive, or remove the - operator if it is negative. We leave the implementation of some of the other cases as a practice problem.

Lastly, we implement the constructor. We have already written the code that creates the entry and the buttons. Instance variables `self.expr` and `self.startOfNextOperand` should also be initialized there. In addition, we should initialize an instance variable that will represent the calculator's memory.

```

1  def __init__(self, parent=None):
2      'calculator constructor'
3      Frame.__init__(self, parent)
4      self.pack()
5
6      self.memory = ''           # memory
7      self.expr = ''            # current expression
8      self.startOfNextOperand = True # start of new operand
9
10     # entry and buttons code

```

Module: calc.py

Practice Problem CS.49

Complete the implementation of the `Calc` class. You will need to implement the code that handles buttons `C`, `MC`, `M+`, `M-`, and `MR` as well as the square root and square buttons.

Use the instance variable `self.memory` in the code handling the four memory buttons. Implement the square root and the square button so that the appropriate operation is applied to the value in the entry and the result is displayed in the entry.

Solution to the Practice Problem

CS.49 Here is the code fragment that is missing:

Module: `calc.py`

```

1  elif key == '\u221a':
2      # compute and display square root of entry
3      result = sqrt(eval(self.entry.get()))
4      self.entry.delete(0, END)
5      self.entry.insert(END, result)
6
7  elif key == 'x\u00b2':
8      # compute and display the square of entry
9      result = eval(self.entry.get())**2
10     self.entry.delete(0, END)
11     self.entry.insert(END, result)
12
13     elif key == 'C':                                # clear entry
14         self.entry.delete(0, END)
15
16     elif key in {'M+', 'M-'}:
17         # add or subtract entry value from memory
18         self.memory = str(eval(self.memory+key[1]+self.entry.get()))
19
20     elif key == 'MR':
21         # replace value in entry with value stored in memory
22         self.entry.delete(0, END)
23         self.entry.insert(END, self.memory)
24
25     elif key == 'MC':                                # clear memory
26         self.memory = ''

```

Problems

CS.50 Develop a widget class `Finances` that incorporates a calculator and a tool to compute the monthly mortgage. In your implementation, you should use the `Calc` class developed in the case study and a `Mortgage` widget from Problem 9.17.

CS.51 Augment calculator widget `Calc` so that the user can type keyboard keys instead of clicking buttons corresponding to the 10 digits, the dot `.`, and the operators `+`, `-`, `*`, and `/`.

Also allow the user to type the Enter/Return key instead of clicking button labeled =.

CS.52 Most calculators clear to 0 and not an empty display. Modify the calculator `Calc` implementation so the default display is 0.

CS.53 Make the `Calc` class a scientific calculator by adding 2 rows of buttons corresponding to the following math functions and constants: $\sin(x)$, $\cos(x)$, $\tan(x)$, π , 2^x , e^x , 10^x , e .

CS.54 Add one more row of buttons to the calculator from the previous problem. This row should contain buttons corresponding to math functions x^3 , x^y , $\frac{1}{x}$ and a button labeled INV whose behavior is as follows. When a function button (e.g., 2^x) is clicked right after clicking button INV, the inverse of the function (e.g., $\log_2 x$) is actually computed.

CS.10 Tower of Hanoi

In this case study, we consider the Tower of Hanoi problem, the classic example of a problem easily solved using recursion. We also use the opportunity to develop a visual application by developing new classes and using object-oriented programming techniques.

Here is the problem. There are three pegs—which we call, from left to right, pegs 1, 2, and 3—and $n \geq 0$ disks of different diameters. In the initial configuration, the n disks are placed around peg 1 in increasing order of diameter, from top to bottom. Figure CS.11 shows the initial configuration for $n = 5$ disks.

Figure CS.11 Tower of Hanoi with five disks.
The initial configuration.



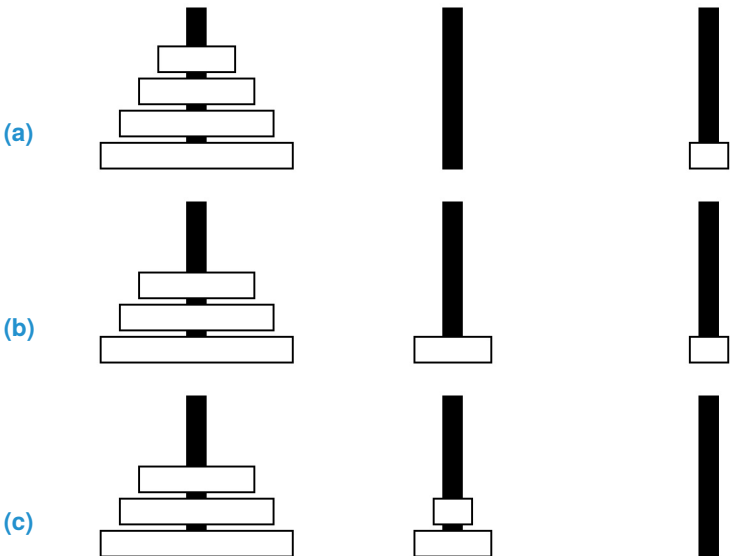
The Tower of Hanoi problem asks to move the disks one at a time and achieve the final configuration shown in Figure CS.12.

Figure CS.12 Tower of Hanoi with five disks.
The final configuration.



There are restrictions on how disks can be moved: (1) Disks can be moved only one at a time, (2) A disk must be released around a peg before another disk is picked up, and (3) A disk cannot be placed on top of a disk with smaller diameter. We illustrate these rules in Figure CS.13, which shows three successive legal moves, starting from the initial configuration from Figure CS.11.

Figure CS.13 Tower of Hanoi with five disks: first three moves. Configuration (a) is the result of moving the topmost, smallest disk from peg 1 to peg 3. Configuration (b) is the result of moving the next smallest disk from peg 1 to peg 2. Note that moving the second smallest disk to peg 3 would have been an illegal move. Configuration (c) is the result of moving the disk around peg 3 to peg 2.



The Recursive Solution

We would like to develop a function `hanoi()` that takes a nonnegative integer n as input and moves n disks from peg 1 to peg 3 using legal single-disk moves. To implement `hanoi()` recursively, we need to find a recursive way to describe the solution (i.e., the moves of the disks). To help us discover it, we start by looking at the simplest cases.

The easiest case is when $n = 0$: There is no disk to move! The next easiest case is when $n = 1$: A move of the disk from peg 1 to peg 3 will solve the problem.

With $n = 2$ disks, the starting configuration is shown in Figure CS.14.



Figure CS.14 Tower of Hanoi with two disks. The initial configuration.

In order to move the two disks from peg 1 to peg 3, it is clear that we need to move the top disk from peg 1 out of the way (i.e., to peg 2) so the larger disk can be moved to peg 3. This is illustrated in Figure CS.15.

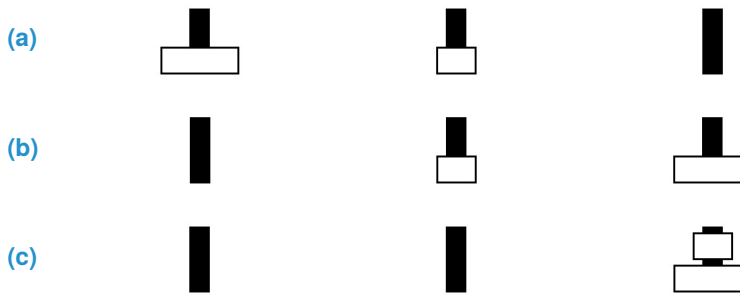


Figure CS.15 Tower of Hanoi with two disks: the solution. Disks are moved in this order: (a) the small disk from peg 1 to peg 2, (b) the large disk from peg 1 to peg 3, and (c) the small disk from peg 2 to peg 3.

In order to implement function `hanoi()` in a clear and intuitive way (i.e., in terms of moving disks from peg to peg), we need to develop classes that represent peg and disk objects. We discuss the implementation of these classes later; at this point, we only need to know how to use them, which we show using the `help()` tool. The documentation for classes `Peg` and `Disk` is:

```
>>> help(Peg)
...
class Peg(turtle.Turtle, builtins.list)
| a Tower of Hanoi peg class
...
| __init__(self, n)
|     initializes a peg for n disks
|
| pop(self)
|     removes top disk from peg and returns it
|
| push(self, disk)
|     pushes disk around peg
...
>>> help(Disk)
...
class Disk(turtle.Turtle)
| a Tower of Hanoi disk class
```

```

...
|   Methods defined here:
|
|   __init__(self, n)
|       initializes disk n

```

We also need to develop function `move()` that takes two pegs as input and moves the topmost disk from the first peg to the second peg:

Module: `turtleHanoi.py`

```

1 def move_disk(from_peg, to_peg):
2     'moves top disk from from_peg to to_peg'
3     disk = from_peg.pop()
4     to_peg.push(disk)

```

Using these classes and function, we can describe the solution of the Tower of Hanoi problem with two disks, illustrated in Figures CS.14 and CS.15 as shown:

```

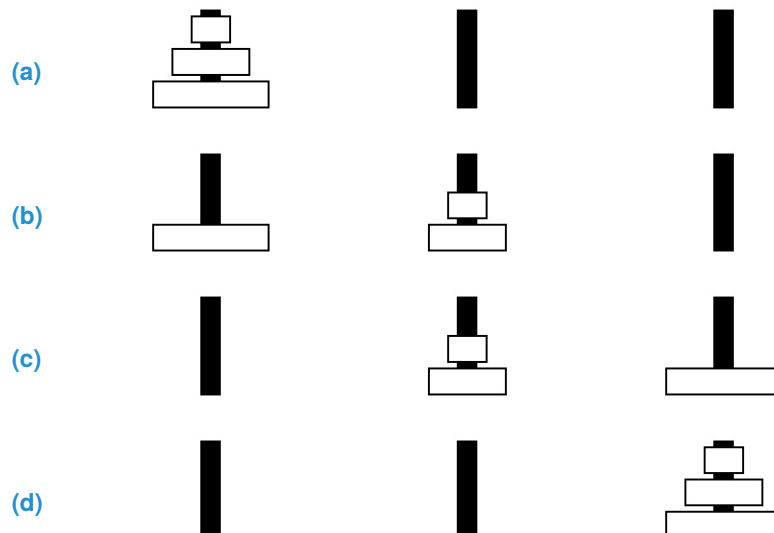
>>> p1 = Peg(2)           # create peg 1
>>> p2 = Peg(2)           # create peg 2
>>> p3 = Peg(2)           # create peg 3
>>> p1.push(Disk(2))      # push larger disk onto peg 1
>>> p1.push(Disk(1))      # push smaller disk onto peg 1
>>> move_disk(p1, p2)     # move top disk from peg 1 to peg 2
>>> move_disk(p1, p3)     # move remaining disk from peg 1 to peg 3
>>> move_disk(p2, p3)     # move disk from peg 2 to peg 3

```

Now let's consider the case when $n = 3$ and try to describe the sequence of disk moves for it recursively. We do this using the same approach we took for $n = 2$: We would like to take the two top disks from peg 1 out of the way (i.e., put them around peg 2) so that we can move the largest disk from peg 1 to peg 3. Once we get the largest disk around peg 3, we again need to move two disks, but this time from peg 2 to peg 3. This idea is illustrated in Figure CS.16.

Figure CS.16 Tower of Hanoi problem with three disks.

Configuration (a) is the initial one. The next one, (b), is the result of recursively moving two disks from peg 1 to peg 2. Configuration (c) is the result of moving the last disk around peg 1 to peg 3. Configuration (d) is the result of recursively moving 2 disks from peg 2 to peg 3.



The question is: How do we move two disks (once from peg 1 to peg 2 and once from peg 2 to peg 3)? Using recursion, of course! We already have a solution for moving two disks, and we can use it. So, if function `hanoi(n, peg1, peg2, peg3)` moves n disks from peg $p1$ to peg $p3$ using intermediate peg $p2$, the next code should solve the Tower of Hanoi problem with three disks.

```
>>> p1 = Peg(3)
>>> p2 = Peg(3)
>>> p3 = Peg(3)
>>> p1.push(Disk(3))
>>> p1.push(Disk(2))
>>> p1.push(Disk(1))
>>> hanoi(2, p1, p3, p2)
>>> move_disk(p1, p3)
>>> hanoi(2, p2, p1, p3)
```

We can now implement the recursive function `hanoi()`. Note that the base case is when $n = 0$, when there is nothing to do.

```
1 def hanoi(n, peg1, peg2, peg3):
2     'move n disks from peg1 to peg3 using peg2'
3
4     # base case: n == 0. Do nothing
5
6     if n > 0: # recursive step
7         hanoi(n-1, peg1, peg3, peg2) # move top n-1 disks
8                                     # from peg1 to peg2
9         move_disk(peg1, peg3)        # move largest disk
10                                    # from peg1 to peg2
11         hanoi(n-1, peg2, peg1, peg3) # move n-1 disks
12                                    # from peg2 to peg3
```

Module: `turtleHanoi.py`

Classes Peg and Disk

We can now discuss the implementation of classes `Peg` and `Disk`. The `Disk` class is a subclass of class `Turtle`. This means that all the attributes of `Turtle` are available to make our `Disk` objects look right.

```
1 from turtle import Turtle, Screen
2 class Disk(Turtle):
3     'a Tower of Hanoi disk class'
4
5     def __init__(self, n):
6         'initializes disk n'
7         Turtle.__init__(self, shape='square', visible=False)
8         self.penup()           # moves should not be traced
9         self.sety(300)         # moves are above the pegs
10        self.shapesize(1, 1.5*n, 2) # set disk diameter
11        self.fillcolor(1, 1, 1)   # disk is white
12        self.showturtle()        # disk is made visible
```

Module: `turtleHanoi.py`

The class `Peg` is a subclass of two classes: `Turtle`, for the visual aspects, and `list`, because a peg is a container of disks. Each `Peg` will have an x -coordinate determined by class variable `pos`. In addition to the constructor, the class `Peg` supports stack methods `push()` and `pop()` to put a disk around a peg or remove a disk from the peg.

Module: `turtleHanoi.py`

```

1  class Peg(Turtle, list):
2      'a Tower of Hanoi peg class, inherits from Turtle and list'
3      pos = -200                                # x-coordinate of next peg
4
5      def __init__(self, n):
6          'initializes a peg for n disks'
7
8          Turtle.__init__(self, shape='square', visible=False)
9          self.penup()                          # peg moves should not be traced
10         self.shapesize(n*1.25,.75,1) # height of peg is function
11                                     # of the number of disks
12         self.sety(12.5*n)                # bottom of peg is y=0
13         self.x = Peg.pos                 # x-coord of peg
14         self.setx(self.x)               # peg moved to its x-coord
15         self.showturtle()               # peg made visible
16         Peg.pos += 200                  # position of next peg
17
18     def push(self, disk):
19         'pushes disk around peg'
20
21         disk.setx(self.x)                # moves disk to x-coord of peg
22         disk.sety(10+len(self)*25) # moves disk vertically to just
23                                     # above the topmost disk of peg
24         self.append(disk)                # adds disk to peg
25
26     def pop(self):
27         'removes top disk from peg and returns it'
28
29         disk = self.pop()                # removes disk from peg
30         disk.sety(300)                   # lifts disk above peg
31         return disk

```

Finally, here is the code that starts the application for up to seven disks.

Module: `turtleHanoi.py`

```

1  def play(n):
2      'shows the solution of a Tower of Hanoi problem with n disks'
3      screen = Screen()
4      Peg.pos = -200
5      p1 = Peg(n)
6      p2 = Peg(n)
7      p3 = Peg(n)
8
9      for i in range(n):                # disks are pushed around peg 1
10         p1.push(Disk(n-i))           # in decreasing order of diameter
11

```



```

12     hanoi(n, p1, p2, p3)
13
14     screen.bye()

```

Problems

CS.55 Suppose someone started solving the Tower of Hanoi problem with five disks and stopped at the configuration illustrated in Figure CS.13(c). Describe a sequence of `move()` and `hanoi()` function calls that will complete the move of the five disks from peg 1 to peg 3. *Note:* You can obtain the starting configuration by executing these statements in the interactive shell:

```

>>> peg1 = Peg(5)
>>> peg2 = Peg(5)
>>> peg3 = Peg(5)
>>> for i in range(5,0,-1):
>>>     peg1.push(Disk(i))

>>> hanoi(2, peg1, peg3, peg2)

```

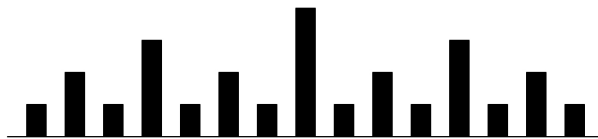
CS.56 Implement function `patternGUI()`, a GUI version of recursive function `pattern()` from Section 10.2. Instead of displaying a number sequence pattern as in

```

>>> pattern(4)
0 1 0 2 0 1 0 3 0 1 0 2 0 1 0 4 0 1 0 2 0 1 0 3 0 1 0 2 0 1 0

```

your function should display a sequence pattern of vertical bars:



The height of each bar should be proportional to the height of the corresponding number in the number sequence pattern.

CS.57 Implement function `toughGUI()`, a GUI version of recursive function `tough()` from Practice Problem 10.23.

CS.11 Web Crawlers

In this case study, we develop a basic *web crawler*, that is, a program that systematically visits web pages by following hyperlinks. (Web crawlers are also referred to as *automatic indexers*, *web robots*, or simply *bots*.) Every time it visits a web page, our web crawler will analyze its content and print out its analysis. The ultimate goal, which we will realize in the next case study, is to use this analysis to build a *search engine*.

Recursive Crawler, Version 0.1

A basic approach to implementing a web crawler is this: After completing the analysis of the current web page, the web crawler will recursively analyze every web page reachable from the current one with a hyperlink. This approach is very similar to the one we used when implementing the virus scanner function `scan()` in Section 10.2. Function `scan()` took as input a folder, put the content of the folder in a list, and then recursively called itself on every item in the list. Our web crawler should take as input a URL, put the hyperlink HTTP URLs contained in the associated web page into a list, and then recursively call itself on every item in the list:

Module: crawler.py

```

1  def crawl1(url):
2      'recursive web crawler that calls analyze() on every web page'
3
4      # analyze() returns a list of hyperlink URLs in web page url
5      links = analyze(url)
6
7      # recursively continue crawl from every link in links
8      for link in links:
9          try: # try block because link may not be valid HTML file
10             crawl1(link)
11         except: # if an exception is thrown,
12             pass # ignore and move on.
```

Since function `crawl1()` is recursive, normally we would need to define a base case for it. Without the base case, the crawler may just continue crawling forever. That is not necessarily wrong in this case, as a crawler should continuously crawl the web. There is an issue with this, however. A continuously running program may exhaust the computer's resources (such as memory), but that is outside of the scope of this text. So, for simplicity's sake, we choose to leave the base case out and let our crawler run free.

The function `analyze()` used in function `crawl1()` encapsulates the analysis of the content of the web page with URL `url`. We will implement this aspect of `analyze()` later. Function `analyze()` also returns the list of links in the web page. We need to implement this part if we want to test our basic web crawler `crawl1()`. We do this using the Collector parser we developed in Section 11.2:

Module: crawler.py

```

1  def analyze(url):
2      '''returns the list of http links, in absolute format, in
3         the web page with URL url'''
4      print('Visiting', url) # for testing
5
6      # obtain links in the web page
```

```

7     content = urlopen(url).read().decode()
8     collector = Collector(url)
9     collector.feed(content)
10    urls = collector.getLinks()      # urls is the list of links
11
12    # analysis of web page content to be done
13    return urls

```

Now let's test our crawler. We do so on a set of linked web pages represented in Figure CS.17. Each page contains a few words (world cities, actually) and links to some of the other pages. For example, the HTML file `five.html` is:

```

1 <html>
2 <body>
3 <a href="four.html">Nairobi Nairobi Nairobi Nairobi Nairobi
4   Nairobi Nairobi</a>
5 <a href="one.html">Bogota</a>
6 <a href="two.html">Bogota</a>
7 </body>
8 </html>

```

File: `five.html`

When we run `crawl1()` starting from web page `one.html`, we get this output:

```

>>> crawl1('http://reed.cs.depaul.edu/lperkovic/one.html')
Visiting http://reed.cs.depaul.edu/lperkovic/one.html
Visiting http://reed.cs.depaul.edu/lperkovic/two.html
Visiting http://reed.cs.depaul.edu/lperkovic/four.html
Visiting http://reed.cs.depaul.edu/lperkovic/five.html
Visiting http://reed.cs.depaul.edu/lperkovic/four.html
Visiting http://reed.cs.depaul.edu/lperkovic/five.html
...

```

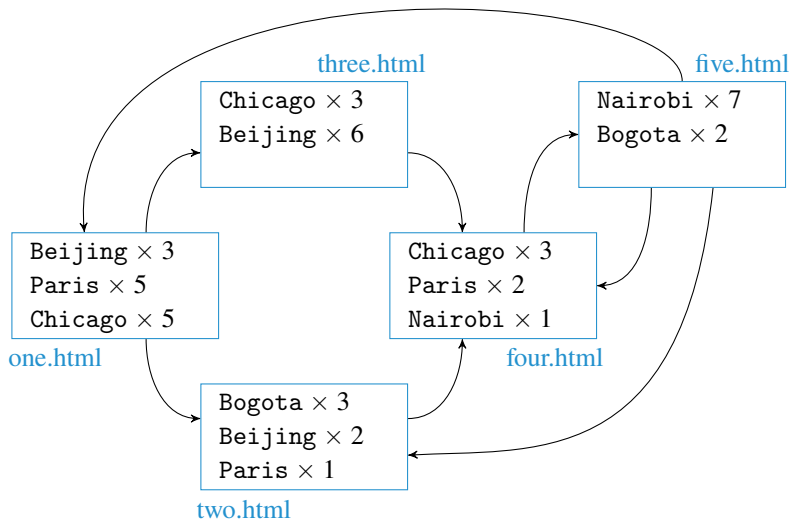
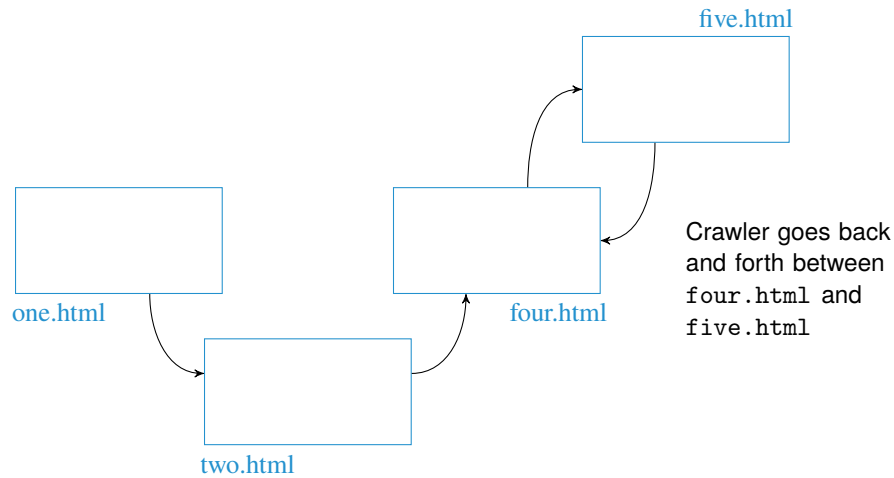


Figure CS.17 Five linked web pages. Each page contains a few occurrences of some of the world's major cities. Page `one.html`, for example, contains 3 occurrences of 'Beijing', 5 of 'Paris', and 5 of 'Chicago'. It also contains hyperlinks to web pages `two.html` and `three.html`.

Figure CS.18 An execution

of `crawl1()`. We start the crawl by calling function `crawl1()` on `one.html`. The first link in `one.html` is to `two.html`, and so a recursive call is made on `two.html`. From there, recursive calls are made on `four.html` and then on `five.html`. There are three links out of `five.html`. Since the first link out of `five.html` is to page `four.html`, a recursive call is made on `four.html`. From there, a recursive call is made on `five.html`...



(The execution did not stop and had to be interrupted by typing `Ctrl-C`.)

Let's try to understand what happened. The crawler started at page `one.html`. There are two links out of `one.html`. The first one is a link to `two.html`, and the crawler followed it (more precisely, made a recursive call on it). The crawler then followed the only link out of `two.html` to page `four.html`, and then, again, the only link from `four.html` to `five.html`. The page `five.html` has three outgoing links. The first one happens to be the link to page `four.html`, and the crawler follows it. From then on, the crawler will visit pages `four.html` and `five.html` back and forth, until it crashes because it reaches the maximum recursion depth or until it is interrupted. (See Figure CS.18 for an illustration.)

Clearly, something went very wrong with this execution. Page `three.html` was never visited, and the crawler got stuck going between pages `four.html` and `five.html`. We can fix the second problem by having the crawler ignore the links to pages it has already visited. To do this, we need to somehow keep track of visited pages.

Recursive Crawler, Version 0.2

In our second crawler implementation, we use a set object to store the URLs of visited web pages. Because this set should be accessible from the namespace of every recursive call, we define the set in the global namespace:

Module: `crawler.py`

```

1  visited = set()                # initialize visited to an empty set
2
3  def crawl2(url):
4      '''a recursive web crawler that calls analyze()
5         on every visited web page'''
6
7      # add url to set of visited pages
8      global visited             # while not necessary, warns the programmer
9      visited.add(url)
10
11     # analyze() returns a list of hyperlink URLs in web page url
12     links = analyze(url)
13

```

```

14     # recursively continue crawl from every link in links
15     for link in links:
16         # follow link only if not visited
17         if link not in visited:
18             try:
19                 crawl2(link)
20             except:
21                 pass

```

Lines 8 and 16 are the difference between `crawl2()` and `crawl1()`. By adding URLs of visited web pages to set `visited` and avoiding links to web pages with URLs in `visited`, we ensure that the crawler does not revisit a page. Let's test this crawler on the same test bed of web pages:

```

>>> crawl2('http://reed.cs.depaul.edu/lperkovic/one.html')
Visiting http://reed.cs.depaul.edu/lperkovic/one.html
Visiting http://reed.cs.depaul.edu/lperkovic/two.html
Visiting http://reed.cs.depaul.edu/lperkovic/four.html
Visiting http://reed.cs.depaul.edu/lperkovic/five.html
Visiting http://reed.cs.depaul.edu/lperkovic/three.html

```

(The execution now stops, unlike the the execution of `crawl1()`.)

The first four pages visited by the crawler are the same as the first four pages visited when testing `crawler1()`. The difference now is that each visited page is added to set `visited`. When the crawler reaches page `five.html`, it finds links to `one.html`, `two.html`, and `four.html`, all of which have been visited. Therefore, the recursive call of `crawl2()` on page `five.html` terminates, and so do recursive calls on pages `four.html` and `two.html` as well. The execution returns to the original function call of `crawl2()` on page `one.html`. The second link in that page is to `three.html`. Since `three.html` has not been visited, the crawler will go ahead and visit it next. Figure CS.19 illustrates this execution of function `crawl2()`.

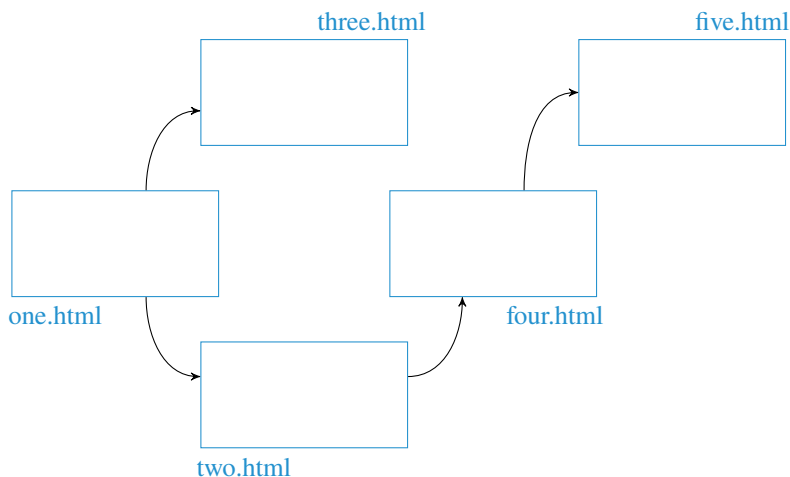


Figure CS.19 Execution of `crawl2()`. Starting from page `one.html`, the crawler visits the same sequence of pages as in Figure CS.18. When the crawler reaches `five.html`, it finds no link to an unvisited page. It then has to backtrack to page `four.html`, then to page `two.html`, then finally `one.html`. The crawler then follows the link out of page `one.html` to unvisited page `three.html`.

Practice Problem
CS.58

Redevelop the second crawler as a class `Crawler2`. The set visited should be encapsulated as an instance variable of the `Crawler2` object rather than as a global variable.

```
>>> crawler2 = Crawler2()
>>> crawler2.crawl('http://reed.cs.depaul.edu/lperkovic/one.html')
Visiting http://reed.cs.depaul.edu/lperkovic/one.html
Visiting http://reed.cs.depaul.edu/lperkovic/two.html
Visiting http://reed.cs.depaul.edu/lperkovic/four.html
Visiting http://reed.cs.depaul.edu/lperkovic/five.html
Visiting http://reed.cs.depaul.edu/lperkovic/three.html
```

The Web Page Content Analysis

The current implementation of function `analyze()` analyzes the content of a web page for the sole purpose of finding hyperlink URLs in it. Our original goal was to do more than that: The function `analyze()` was supposed to analyze the content of each web page and print this analysis out. We now add this additional functionality to function `analyze()` and complete its implementation.

We choose that the web page analysis consists of computing (1) the frequency of every word in the web page content (i.e., in the text data) and (2) the list of links contained in the web page. We have already computed the list of links. To compute the word frequencies, we can use function `frequency()` we developed in Practice Problem 13.6. Here is then our final implementation:

Module: crawler.py

```
1 def analyze(url):
2     '''prints the frequency of every word in web page url and
3     prints and returns the list of http links, in absolute
4     format, in it'''
5
6     print('Visiting', url)          # for testing
7
8     # obtain links in the web page
9     content = urlopen(url).read().decode()
10    collector = Collector(url)
11    collector.feed(content)
12    urls = collector.getLinks()      # get list of links
13
14    # compute word frequencies
15    content = collector.getData()    # get text data as a string
16    freq = frequency(content)
17
18    # print the frequency of every text data word in web page
19    print('\n{:50} {:10} {:5}'.format('URL', 'word', 'count'))
20    for word in freq:
21        print('{:50} {:10} {:5}'.format(url, word, freq[word]))
22
23    # print the http links found in web page
24    print('\n{:50} {:10}'.format('URL', 'link'))
```

```

25     for link in urls:
26         print('{:50} {:10}'.format(url, link))
27
28     return urls

```

Using this version of `analyze()`, let's test our crawler again. We start the crawl with:

```
>>> crawl2('http://reed.cs.depaul.edu/lperkovic/one.html')
```

The output that is printed in the interactive shell is shown on the next page. *Note:* In order to get the output to fit the width of the page and also to have a cleaner view of it, we edited out from some of the URLs the substring

```
http://reed.cs.depaul.edu/lperkovic/
```

```
Visiting http://reed.cs.depaul.edu/lperkovic/one.html
```

URL	word	count
one.html	Paris	5
one.html	Beijing	3
one.html	Chicago	5

URL	link
one.html	two.html
one.html	three.html

```
Visiting http://reed.cs.depaul.edu/lperkovic/two.html
```

URL	word	count
two.html	Bogota	3
two.html	Paris	1
two.html	Beijing	2

URL	link
two.html	four.html

```
Visiting http://reed.cs.depaul.edu/lperkovic/four.html
```

URL	word	count
four.html	Paris	2
four.html	Nairobi	1
four.html	Chicago	3

URL	link
four.html	five.html

```
Visiting http://reed.cs.depaul.edu/lperkovic/five.html
```

URL	word	count
five.html	Bogota	2
five.html	Nairobi	7

URL	link
five.html	four.html
five.html	one.html
five.html	two.html

Visiting `http://reed.cs.depaul.edu/lperkovic/three.html`

URL	word	count
three.html	Beijing	6
three.html	Chicago	3

URL	link
three.html	four.html

DETOUR



Depth-First and Breadth-First Traversals

The approach that the crawler version 0.2 uses to visit pages on the web is called *depth-first traversal*. *Traversal* is synonymous with *crawl* for our purposes. The *depth-first* term refers to the fact that, in this approach, the crawler can quickly move away from the start of the crawl. To see this, look at Figure CS.19. It shows that the crawler visits faraway pages `four.html` and `five.html` before it visits neighboring page `three.html`.

The problem with depth-first traversal is that it may take a very long time to visit a neighboring page. For example, if page `five.html` had a link to `www.yahoo.com` or `www.google.com`, it is unlikely that the crawler would ever visit page `three.html`.

For this reason, crawlers used by Google and other search providers use a *breadth-first traversal* that ensures that pages are visited in the order of *distance* (the number of links) from the starting web page. Problem 11.64 asks you to implement this approach.

Solution to the Practice Problem

CS.58 The set `visited` should be initialized in the constructor. The method `crawl()` is a slight modification of function `crawl2()`:

```
class Crawler2:
    'a web crawler'

    def __init__(self):
        'initializes set visited to an empty set'
        self.visited = set()

    def crawl(self, url):
        '''calls analyze() on web page url and calls itself
           on every link to an unvisited web page'''
```



```

links = analyze(url)
self.visited.add(url)
for link in links:
    if link not in self.visited:
        try:
            self.crawl(link)
        except:
            pass

```

Problems

CS.59 Modify the crawler function `crawl1()` so that the crawler does not visit web pages that are more than n click (hyperlinks) away. To do this, the function should take an additional input, a nonnegative integer n . If n is 0, then no recursive calls should be made. Otherwise, the recursive calls should pass $n - 1$ as the argument to the `crawl1()` function.

CS.60 Using Figure 10.1 as a model, draw all the steps that occur during the execution of `crawl2('one.html')`, including the state of the program stack at the beginning and end of every recursive call.

CS.61 Modify the crawler function `crawl2()` so that the crawler only follows links hosted on the same host as the starting web page.

CS.62 Modify the crawler function `crawl2()` so that the crawler only follows links to resources that are contained, directly or indirectly, in the web server filesystem folder containing the starting web page.

CS.63 Develop a crawler that collects the email addresses in the visited web pages. You can use function `emails()` from Problem 11.19 to find email addresses in a web page. To get your program to terminate, you may use the approach from Problem CS.59 or Problem CS.61.

CS.64 Implement a web crawler that uses breadth-first traversal rather than depth-first. Unlike depth-first traversal, breadth-first traversal is not naturally implemented using recursion. Instead, iteration and a queue (of the kind we developed in Section 8.3) are used. The purpose of the queue is to store URLs that have been *discovered* but not visited yet. Initially, the queue will contain the starting web page only, the only discovered URL at that point. In every iteration of a `while` loop, the queue is dequeued to obtain a URL, and *then* the associated web page is visited. Any link in the visited page with a URL that has not been visited or discovered is then added to the queue. The `while` loop should iterate as long as there are discovered but unvisited URLs (i.e., as long as the queue is not empty).

CS.12 Data Interchange

In Case Study CS.11, we have developed a simple web crawler that collects information about web pages it visits and then prints that information on the screen. More useful than printing on the screen, however, would have been to output the information to a file. As we saw in Practice Problem 12.3 and Practice Problem 12.4, the information collected by a crawler can be used to build a search engine. By saving the crawl data into a file, we can make that data available to other programs. In this Case Study, we look at *data interchange* or how to format and save data so that it is accessible, easily and efficiently, to any program that requires it.

Serialization and Data Interchange Formats

A good software engineering principle is for a program to do only one task and do it well (e.g., efficiently). Many applications, however, consist of several tasks and data that is produced by one task (the producer program) may need to be consumed by another (the consumer program). Because the consumer program is not necessarily going to handle the data as soon as it is produced by the producer, the data will need to be stored in a file. Files can be either text or binary; text files have the advantage of being readable by humans, and for that reason they are typically used for data interchange. This means that the data from the producer, typically consisting of objects such as lists and dictionaries containing various types of objects, must be translated to a string, a process called *serialization*. The string representation of the data will need to be read by the consumer who will need to re-create the original lists and/or dictionaries from it, a process called *deserialization*.

The format of the string representation of the serialized object must be well defined so the data serialized by the producer can be properly and independently deserialized by the consumer. This is particularly important because the producer program and consumer program are likely to be developed independently, by different developers, and perhaps even using different programming languages. The format should also be efficient to serialize and deserialize and, preferably, easy to read by humans.

In the last 20 years, a number of data interchange formats have been defined: XML (Extensible Markup Language) and JSON (JavaScript Object Notation) are two that are currently seeing widespread use. In this case study, we will learn about JSON.

JSON (JavaScript Object Notation)

JSON (JavaScript Object Notation) defines a standard format for describing, using strings, typical built-in objects such as dictionaries, lists, numbers, and strings. While originally a subset of the JavaScript programming language, it is now a language-independent data format and all major programming languages have libraries that produce and consume JSON data.

DETOUR



JavaScript

JavaScript is a programming language that is typically used within web pages and executed by web browsers. Web pages contain JavaScript programs for a variety of reasons: to interact with the user, to change the displayed content, and to communicate with the web server, among others. JavaScript was famously developed

in 10 days by Brendan Eich in order for it to be incorporated into the beta version of the Netscape Navigator 2.0 web browser, way back in 1995. Despite its name, JavaScript is not related to the Java programming language; the name was simply a marketing ploy designed to capitalize on the popularity, at the time, of the Java programming language.

The Python Standard Library module `json` contains functions to encode a Python object into JSON format (or simply JSON) and to decode JSON back to a Python object (see Table CS.8). We use this dictionary object to illustrate its usage:

```
>>> record = {'url': 'one.html', 'links': ['two.html', 'three.html']}
```

(As motivation, you can assume that this dictionary is the summary of the analysis of the web page with URL `one.html` and containing links to web pages with URL `two.html` and `three.html`.)

Function `dumps()` in module `json` is used to serialize a Python object into JSON. It takes a Python object as input and returns a string that is the JSON representation of the Python object:

```
>>> import json
>>> jsonRec = json.dumps(record)
>>> jsonRec
'{"links": ["two.html", "three.html"], "url": "one.html"}'
```

Note that the JSON representation of a dictionary, a list, and a string is essentially the same as Python's string representation of these same objects. While this is very convenient, it may also be confusing, so we go over the returned string more carefully. The Python string

```
'{"links": ["two.html", "three.html"], "url": "one.html"}'
```

returned by function `dumps()` is a representation of dictionary `record` in JSON format. The JSON consists of a *JSON object* containing two key:value pairs. The first one maps key `"links"`, a *JSON string*, to a value that is a *JSON array* containing two (JSON) strings `"two.html"` and `"three.html"`. The second key:value pair maps key `"url"`, a (JSON) string, to a value that is the (JSON) string `"one.html"`.

We now define more formally the types of values supported by JSON. There are two types of JSON containers:

- A JSON object, which is a collection of key:value pairs separated by commas and enclosed in curly braces (and that corresponds to a Python dictionary).
- A JSON array, which is an ordered collection of values separated by commas and enclosed in square brackets (and that corresponds to a Python list).

Usage	Explanation
<code>dumps(record)</code>	Return a string that contains the serialization of <code>record</code> in JSON format
<code>loads(serial)</code>	Return the Python object that is the deserialization of <code>serial</code> (a serialization in JSON format)

Table CS.8 Two functions in module `json`. These are used to serialize a Python object to JSON and to deserialize it from JSON back to a Python object.

The key in a key:value pair of a JSON object must be a string (unlike Python which allows the key to be any immutable type). Values in JSON objects and arrays can be JSON objects, arrays, numbers (integers or floating point, just as in Python), strings (unlike Python, always represented with double quotes), and Boolean values `true` and `false` (not quite the same as Python's `True` and `False`).

To deserialize a Python string that contains JSON, we use function `load()` from the `json` module:

```
>>> recordDup = json.loads(jsonRec)
>>> recordDup
{'links': ['two.html', 'three.html'], 'url': 'one.html'}
>>> type(recordDup)
<class 'dict'>
```

The function returns a Python object, referred to as `recordDup`, whose type matches the type of the original dictionary object `record`. It is critical that the deserialized Python object has the same value as the pre-serialized Python object. We check that is the case:

```
>>> recordDup == record
True
```

Practice Problem CS.65

JSON has certain restrictions on the kind of Python objects that can be serialized to JSON data. Give an example of a Python dictionary that violates the restrictions and check that the deserialization of the serialization of the dictionary is not equal in value to the dictionary.

Data Compression

Data files are stored on a hard disk drive (or solid-state drive, magnetic tape, etc.) and can also be sent across a computer network (e.g., from the computer cluster running the producer program to the computer cluster running the consumer program). In today's Big Data world, data files are often huge and can take a lot of disk space to store or time to transmit across the network. In order to save disk space and transmission time, large data files should be compressed.

To illustrate how to create, in a Python program, a compressed file containing JSON data, we turn back to our crawler example from Case Study CS.11. Suppose that we would like to store in a compressed file many records similar to:

```
>>> record = {'links': ['two.html', 'three.html'], 'url': 'one.html'}
```

The Python Standard Library module `gzip` contains functions and classes that are used to read and write compressed files. We will, in particular, make use of the class `GzipFile` defined in the module.

To create (i.e., open) a compressed file, we use the `GzipFile` constructor as follows:

```
>>> import gzip
>>> outfile = gzip.GzipFile('crawl.txt.gz', 'wb')
```

Just like function `open()` that is used to open regular, uncompressed files, the `GzipFile()` constructor takes as input the name of the file to be opened and the mode. The name of the file has a `.gz` suffix, which is the standard suffix for a file compressed using the `gzip` file

compression format. The file mode is 'wb' because we will write into the file and because compressed files are binary, not text, files.

Data compression algorithms and formats

As we saw in Section 6.3, uncompressed string data is stored in memory using a Unicode encoding such as UTF-8. Data compression involves finding, for given string data, a more efficient way to represent the data. One way to achieve this is to replace the UTF-8 encoding with an encoding that is shorter for characters that appear more often in the string. That is (basically) the idea behind the Huffman code, developed by David Huffman in 1952. This idea can be generalized so that repeated occurrences of a substring, not just a character, are replaced with (short) references to a single copy of the substring. This is the approach taken by the LZ77 compression algorithm developed by Abraham Lempel and Jacob Ziv in 1977. Module `gzip` makes use of algorithm DEFLATE that is essentially a hybrid of the LZ77 and Huffman code approaches.

There are numerous data compression applications available today. Module `gzip` is really an interface to the Zlib application. Other commonly used data compression apps include PKZIP, Gzip (which differs from Zlib only in the meta-data format), and PNG (used to compress images). Data compression apps differ in the compression algorithm (though most use DEFLATE in some way) and the meta-data (original file length, timestamp, checksum, etc.) added to the compressed file.

DETOUR



A `GzipFile` object is a file-like object, which means that you can write to it using method `write()` just as you would with any other file. In our case we want to write a line containing the JSON representation of record:

```
>>> import json
>>> line = json.dumps(record) + '\n'
```

Since the file is binary, however, we must first encode the string `line` to bytes before writing:

```
>>> outfile.write(line.encode())
57
```

To make the example a bit more realistic, we add a few more records to the file:

```
>>> records = [{"links": ["four.html"], "url": "two.html"}, \
               {"links": ["four.html"], "url": "three.html"}, \
               {"links": ["five.html"], "url": "four.html"}, \
               {"links": ["one.html", "two.html", "four.html"], \
                "url": "five.html"}]
>>> for record in records:
    outfile.write((json.dumps(record) + '\n').encode())
```

```
44
46
45
```

```
69
>>> outfile.close()
```

After closing the file, we have a compressed file in the current working directory. It effectively contains, in compressed form, a text file containing a JSON object in every line.

We now turn our attention to uncompressing this file and deserializing the JSON objects in it. We first open the compressed file for reading using the `GzipFile()` constructor:

```
>>> infile = gzip.GzipFile('crawl.txt.gz', 'rb')
```

Note the use of the binary mode because the compressed file is binary. After the compressed file has been opened, the compression is transparent and the file is read without regard to compression. For example, we can read the file line by line, decode each line from the bytes type to a string containing JSON, and then deserialize JSON to a Python object:

```
>>> for line in infile:
    json.loads(line.decode())

{'url': 'one.html', 'links': ['two.html', 'three.html']}
{'url': 'two.html', 'links': ['four.html']}
{'url': 'three.html', 'links': ['four.html']}
{'url': 'four.html', 'links': ['five.html']}
{'url': 'five.html', 'links': ['one.html', 'two.html', 'four.html']}
```

We got our dictionary records back.

Practice Problem CS.66

Run, in the interactive shell, the code from the previous example that creates the file `crawl.txt.gz`, and find out the size of the file. Then repeat the process without compression to create an uncompressed text file `craw2.txt` that contains the same JSON data. Compute the ratio of the uncompressed file size and the compressed file size. This ratio is referred to as the *compression ratio*.

I/O Streams

We just saw how to open and read a compressed file when the file is stored in the local computer's filesystem. What if the file was on a remote server instead? After all, it is very likely that the program that produced the data is on a different system than the consumer program. As an example, let's see how to open file `crawl.txt.gz` when it is stored on a remote web server and can be identified with a URL.

We can start by retrieving the file just as we retrieved web pages in Section 11.2:

```
>>> from urllib.request import urlopen
>>> url = 'http://reed.cs.depaul.edu/lperkovic/crawl.txt.gz'
>>> response = urlopen(url)
```

Let's look at the header fields of the HTTP response:

```
>>> for field in response.getheaders():
    print(field)

('Server', 'Apache-Coyote/1.1')
```

```
...
('Content-Type', 'application/x-gzip')
('Content-Length', '111')
...
```

The Content-Type header field shows that the retrieved content is a compressed file in gzip format. We can still read the content as usual:

```
>>> content = response.read()
```

Recall that object returned by method `read()` is of type `bytes`. When we retrieved a web page in Section 11.2, all we needed to do is decode the `bytes` content to interpret the bytes as a string in some Unicode encoding. This time, however, the binary data is the compression of text data, and it needs to be decompressed first.

We saw previously how to read (or decompress) a compressed file. The situation is a bit different now. The compressed data is not in a file on disk: It is in memory. We could, of course, write the binary data to a local file and then read it as we did previously but that would be silly. There is a more elegant approach that uses a generalization of a file called an *I/O stream*.

The file API (opening and closing, reading and writing files) is a powerful abstraction that is useful for processing not just regular files but also other types of inputs and outputs. An I/O stream is a type of object that can represent many type of inputs or outputs (disk, memory, I/O devices, etc.) and that is file-like: It behaves like a file and supports methods such as `read()` and `write()`.

The Python Standard Library module `io` provides classes `StringIO` and `BytesIO` that make it possible to process string (`str`) and `bytes` objects as file-like objects. Before we continue with our working example, we take a small detour to illustrate how `StringIO` is used. Consider the poem by Emily Dickinson that we saw in Section 4.1:

```
>>> poem = '''
To make a prairie it takes a clover and one bee, -
One clover, and a bee,
And revery.
The revery alone will do
If bees are few.
'''
```

With the `StringIO()` constructor, we create a file-like object whose content is the poem:

```
>>> import io
>>> poemStream = io.StringIO(poem)
```

We can now read from the file-like object as we would from a regular file:

```
>>> poemStream.readline()
'\n'
>>> poemStream.readline()
'To make a prairie it takes a clover and one bee, -\n'
>>> poemStream.readline()
'One clover, and a bee,\n'
>>> poemStream.read()
'And revery.\nThe revery alone will do\nIf bees are few.\n'
```

We now return to our working example. Because `content` is of type `bytes`, we use the

BytesIO class instead of StringIO to create a file-like object:

```
>>> import io
>>> compressed = io.BytesIO(content)
```

Now, `compressed` refers to a file-like object that contains compressed data. To read it, we make use of the `GzipFile` class again. This time, however, we do not pass a filename but a reference `compressed` to a file-like object:

```
>>> import gzip
>>> infile = gzip.GzipFile(fileobj=compressed, mode='rb')
```

The `GzipFile()` constructor has several input parameters, all optional. The first one is `filename`, and `fileobj` is another. If a file name is not passed to the constructor, then a reference to a file-like object must be passed to parameter `fileobj`.

Finally, we read the file-like object exactly as we did before:

```
>>> import json
>>> for line in infile:
    json.loads(line.decode())

{'links': ['two.html', 'three.html'], 'url': 'one.html'}
{'links': ['four.html'], 'url': 'two.html'}
{'links': ['four.html'], 'url': 'three.html'}
{'links': ['one.html', 'two.html', 'four.html'], 'url': 'five.html'}
{'links': ['five.html'], 'url': 'four.html'}
```

Solution to the Practice Problems

CS.65 JSON restricts keys of JSON objects to be strings. Therefore a Python dictionary with nonstring keys such as `{2: 'two', 3: 'three', 4: 'four'}` should not be serialized to a JSON object. Let's try it anyways:

```
>>> rec = {2: 'two', 3: 'three', 4: 'four'}
>>> json.dumps(rec)
'{"2": "two", "3": "three", "4": "four"}'
```

Note that the integer keys are transformed into string keys. We check that the deserialized JSON data is a Python dictionary whose value is *not* the same as the value to the original dictionary:

```
>>> json.loads(json.dumps(rec))
{'2': 'two', '4': 'four', '3': 'three'}
>>> rec2 = json.loads(json.dumps(rec))
>>> rec == rec2
False
```

CS.66 We create the uncompressed file as follows:

```
>>> outfile = open('crawl2.txt', 'w')
>>> records = [{ 'links': ['two.html', 'three.html'], \
    'url': 'one.html'}, \
    {"links": ["four.html"], "url": "two.html"}, \
```



```

{"links": ["four.html"], "url": "three.html"}, \
{"links": ["five.html"], "url": "four.html"}, \
{"links": ["one.html", "two.html", "four.html"], \
 "url": "five.html"}]
>>> for record in records:
    outfile.write(json.dumps(record) + '\n')

57
44
46
45
69
>>> outfile.close()

```

The size of `crawl2.txt` on my machine is 261 bytes. The compression ratio is thus $\frac{261}{111} \approx 2.35$.

Problems

CS.67 Modify your solution to Problem CS.63 so that all the emails collected by the crawler are stored in a Python list that is, upon completion of the crawl, serialized into a JSON array and then written in a compressed file saved in the current working directory.

CS.68 Modify the functions `crawl2()` and `analyze()` from Case Study CS.11 so that the data collected from a web page is stored in a Python dictionary that is then serialized into a JSON object and written in one line of a compressed file named `mycrawl.txt.gz` (located in the current working directory). The Python dictionary should have keys `url`, `links`, and `words` mapping to the URL of the analyzed web page, a list of hyperlinks in it, and a dictionary mapping every word appearing in the web page to the number of occurrences of the word. Each line in file `mycrawl.txt.gz` should contain a JSON object corresponding to a visited web page.

CS.69 Reimplement function `webData()` from Practice Problem 12.3 so that it takes just two inputs: the name of a database file and the name of a compressed text file whose content is as described in Problem CS.68.

CS.70 A web search by a crawler of the type you developed in Problem CS.68 generates an enormous amount of data that often cannot be stored in a single file. Instead, many files similar to `mycrawl.txt.gz` must be created and the names of these files are then stored in a compressed text file, one file name per line. A function that processes crawl data such as `webData()` from Problem CS.69 would need to go through this file filename by filename and process each filename just as described in Problem CS.69. Reimplement `webdata()` to do this.