CHAPTER 8

Science and Visualization

Numerical Analysis and Signal Processing

I've covered many topics associated with data analysis and visualization: reading and writing files, text processing and converting text to numerical data, plotting and graphing, writing scripts, and implementing algorithms. It's time to take a deeper dive and analyze numerical data.

This chapter deals with two important topics: numerical analysis and signal processing. These two topics appear in many sciences: mathematics, computing, engineering, and more. From a simplistic point of view, *numerical analysis* is concerned with algorithms that yield numerical values: a solution to a nonlinear equation, the decimal representation of π, and so on. *Signal processing* deals with processing signals—that is, values that change over time. Signal processing includes such topics as detection and filtering.

Most universities and colleges offer undergraduate courses that teach these topics. But you don't have to be an engineer or a computer scientist to use the methods and ideas discussed in the chapter; most of the topics are easy to follow. If you have a strong numerical analysis and signal processing background, this chapter should prove a good starting point for these topics in Python. If you're new to the ideas of numerical analysis and signal processing, I hope to shed some light on them, so that you can pick things up from here with relevant scientific literature. In particular, I'd like to point out one of the books that made a great impact on me (and many others): *Numerical Recipes: The Art of Scientific Computing,* Cambridge University Press, 2007 (see the "Final Notes and References" section at the end of the chapter). Although the book implements algorithms using C/C++ (my original copy was in the Pascal programming language), it provides a wealth of information on numerical algorithms and should prove easy enough to port to Python.

In my view, the field of numerical analysis is a cookbook of algorithms to numerically solve mathematical problems. And so in a sense, that's how the chapter is organized as well: as a list of problems and solutions. Each topic will be explored with examples in hopes that you'll modify the examples to fit your needs. And that's also how I suggest you refer to the chapter: as a cookbook of algorithms. While it's quite possible to read through and learn the algorithms one at a time, it's probably easier to read specific sections as you engage problems associated with them in real-life. So my suggestion is this: skim through the table of contents to acquaint yourself with what's available, and then try to solve a specific problem by reading the relevant section.

In this chapter, I've used *SciPy*, *matplotlib*, and *NumPy* extensively. These three packages are rich and complex; and as a result, I was only able to cover some of their functionality, not all of it. I therefore chose to cover topics and show examples of problems I've personally encountered. I hope you'll find the examples of value.

Finding Your Way: Variables and Functions

The *NumPy* package provides us with two useful helper functions. I call them helper functions because they don't fall into any specific numerical analysis or signal processing category.

When one works in an interactive environment, one constantly defines variables. It's hard to remember what variables are defined and what they mean. The function `who()` prints a list of current *NumPy* arrays:

```
>>> from pylab import *
>>> who()
```

```
Upper bound on total bytes  =       0
```

```
>>> up, down = arange(10), arange(10, 0, -1)
>>> who()
```

```
Name            Shape           Bytes           Type
============================================================

up              10              40              int32
down            10              40              int32

Upper bound on total bytes  =       80
```

The function `lookfor()` is great for searching inside docstrings. So, to look for functions that perform numerical integration, issue the following:

```
>>> lookfor('integrate')
```

```
Search results for 'integrate'
------------------------------
numpy.trapz
    Integrate along the given axis using the composite trapezoidal rule.
```

(I've left out some of the output, but there are several functions that have `'integrate'` in their docstring.)

SciPy

SciPy (http://www.scipy.org/) is an open source scientific library for Python. The idea of *SciPy* is similar to that of Octave-Forge (http://octave.sourceforge.net/), which provides extra packages for GNU-Octave (http://www.octave.org) and toolboxes that enhance MATLAB (http://www.mathworks.com). *SciPy* is built on top of *NumPy*, so it requires *NumPy* to work properly.

SciPy is organized into several modules, some of which are detailed in Table 8-1.

Table 8-1. *SciPy Packages*

Package	Description
Fftpack	Fast Fourier Transform
Integrate	Integration functions, including ordinary differential equations
Interpolate	Interpolation of functions
Linalg	Linear algebra
Optimize	Optimization functions, including root-solving algorithms
Signal	Signal processing
Special	Special functions (Airy, Bessel, etc.)

We'll be exploring *SciPy* modules that deal with numerical analysis and signal processing. Additional *SciPy* modules include sparse matrices (module *scipy.sparse*), statistics (module *scipy.stats*), and more. And the list continues to grow; however, they will not be covered in this book. For a full account of the available modules, issue `import scipy`, followed by `help(scipy)`.

To import a specific *SciPy* module, issue `import scipy.modulename`. For example, to import *scipy.linalg*, issue:

```
>>> import scipy.linalg
```

You can also accomplish the task this way:

```
>>> from scipy import linalg
```

Personally, I prefer the latter option: `linalg.eig()` is shorter to code than `scipy.linalg.eig()` (plus I think it's easier to read).

Linear Algebra

Linear algebra is a branch in mathematics that deals with matrices, vectors, and solving systems of linear equations. *SciPy* and *NumPy* provide us with many functions to deal with these topics: solving systems of linear equations, matrix and vector operations, and matrix decompositions.

Solving a System of Linear Equations

To solve a system of linear equations, we first write the problem in matrix notation.

```
2 * x + 3 * y = 10
3 * x -     y = -1.5
```

We start by defining a matrix, `M`, and a vector, `V`. The matrix is composed of the coefficients of `x` and `y`, which are 2 and 3 on the first row, or `[2, 3]`. They are 3 and –1 on the second row, or `[3, -1]`:

```
>>> from pylab import *
>>> M = array([[ 2, 3], [3, -1]])
```

Next, we define the vector of the results, `[10, -1.5]`:

```
>>> V = array([10, -1.5])
```

Now all that's required is to use the function `solve()`:

```
>>> solve(M, V)
```

```
array([ 0.5,  3. ])
```

The result translates into: `x` is equal to 0.5, and `y` is equal to 3.
It's also possible to reach the solution by calculating the inverse of the matrix `M` and multiplying it by vector `V`:

```
>>> dot(inv(M), V)
```

```
array([ 0.5,  3. ])
```

I've introduced two functions here: `inv()` and `dot()`. The function `inv()` calculates the inverse of a matrix, and the function `dot()` performs a dot product. Had I multiplied `inv(M)` with `V`, I would've received an *element-by-element* multiplication, instead:

```
>>> inv(M)*V
```

```
array([[ 0.90909091, -0.40909091],
       [ 2.72727273,  0.27272727]])
```

Generally speaking, you should use `solve()` instead of `inv()`. The function `solve()` can handle what mathematicians call "less-behaved" matrices—matrices where the determinant is very close to zero. Using `inv()` on these matrices might produce inaccurate results (i.e., dividing by a number close to zero).

Vector and Matrix Operations

Much like `dot()`, the function `vdot()` returns the dot product of two vectors. So if you're only interested in the value of `x` in the previous example, you can write

```
>>> dot(inv(M)[0], V)
```

```
0.50000000000000022
```

The function `inner(v1, v2)` will perform an inner product; that is, it will multiply every element in `v1` with the corresponding element in `v2`, and then sum them together:

```
>>> V1 = array([10, -1.5])
>>> V2 = array([1, 2])
>>> sum = 0
>>> for i in range(len(V1)):
...     sum += V1[i]*V2[i]
```

```
...
>>> sum
```

```
7.0
```

```
>>> inner(V1, V2)
```

```
7.0
```

I've implemented an inner product operation with a `for` loop and compared the results with the results of the function `inner()`. As you would expect, the results are the same. Note that the function `inner()` does not multiply an element with its conjugate (negative imaginary part).

The function `inner()` works on matrices, as well:

```
>>> M = array([[2, 3], [3, -1]])
>>> M
```

```
array([[ 2,  3],
       [ 3, -1]])
```

```
>>> inner(M, inv(M))
```

```
array([[  1.00000000e+00,   1.11022302e-16],
       [  5.55111512e-17,   1.00000000e+00]])
```

Similarly, `outer()` performs an outer product of two vectors or matrices:

```
>>> V1 = array([10, -1.5])
>>> V2 = array([1, 2])
>>> outer(V1, V2)
```

```
array([[ 10. ,  20. ],
       [ -1.5,  -3. ]])
```

The function `transpose()` will permute axes, and `conjugate()` will permute axes and negate the imaginary part of a matrix or vector:

```
>>> V1 = array([10, -1.5])
>>> V2 = array([1, 2])
>>> outer(V1, V2)
```

```
array([[ 10. ,  20. ],
       [ -1.5,  -3. ]])
```

```
>>> outer(V2, V1)
```

```
array([[ 10. ,  -1.5],
       [ 20. ,  -3. ]])
```

```
>>> all(outer(V1, V2) == transpose(outer(V2, V1)))
```

```
True
```

```
>>> conjugate(V1+1j*V2)
```

```
array([ 10.0-1.j,  -1.5-2.j])
```

The function `det(M)` will return the determinant of matrix `M`:

```
>>> det(array([[2, 3], [3, -1]]))
```

```
-11.000000000000002
```

Matrix Decomposition

Matrix decomposition is the rewriting of a matrix to a specific form. There are many decompositions including LU decomposition, singular value decomposition, and QR decomposition. *NumPy*'s linear algebra module supports some matrix decompositions via the functions shown in Table 8-2.

Table 8-2. *Some Matrix Decomposition Functions*

Function	Description
`cholesky(m)`	Cholesky decomposition
`eig(m)`	Eigenvalue decomposition
`qr(m)`	QR decomposition
`svd(m)`	Singular value decomposition

The following code performs eigenvalue decomposition and verifies the results:

```
>>> A = array([[1, 2], [0, 1]])
>>> L, v = eig(A)  # calculate eigenvalues and eigenvectors
>>> det(A - eye(2)*L)   # verify eigenvalues (should be zero)
```

```
0.0
```

```
>>> dot(A, v[:, 0]) - L[0]*v[:, 0]  # verify 1st eigenvector (should be 0)
```

```
array([ 0.,  0.])
```

```
>>> dot(A, v[:, 1]) - L[1]*v[:, 1]  # verify 2nd eigenvector (should be 0)
```

```
array([  2.22044605e-16,   0.00000000e+00])
```

I've created a matrix (`A`) and calculated its eigenvalues $\lambda_{1,2}$ (stored in vector `L`) and eigenvectors $v_{1,2}$ (stored in matrix `v`); this is done in the line `L, v = eig(A)`. Once the eigenvalues are evaluated, they can be verified by calculating det(A - λ·I), which should be zero; this is done in the line: `det(A - eye(2)*L)`. Also, for every eigenvector λ, λ·v should be equal to A·v; this is verified in the last two lines: `dot(A, v[:, 0]) - L[0]*v[:, 0]` and `dot(A, v[:, 1]) - L[1]*v[:, 1]`.

We will not be covering other matrix decompositions here; if you require additional information, `help()` is quite informative.

Additional Linear Algebra Functionality

Additional linear algebra functionality is available with the *scipy.linalg* module. To access *SciPy*'s linear algebra functions, issue `import scipy.linalg` or `from scipy import linalg`. *SciPy*'s added functionality includes the following:

- Matrix decomposition functions: `lu()` for LU decomposition and `qr()` for QR matrix decomposition, as well as functions for other decompositions.
- Matrix and vector operators such as `norm()` to calculate a matrix or vector norm, `det()` to calculate the determinant of a matrix, and `inv()`to invert a matrix.
- Matrix functions like `expm()` and `tanm()`. Matrix function names are similar to regular function names, but with an added character, `m`.
- Special matrices, such as the Hadamard matrix (`hadamard()`) used in some error correction codes (see http://en.wikipedia.org/wiki/Hadamard_matrix) and the Hilbert matrix (`hilbert()`) (see http://en.wikipedia.org/wiki/Hilbert_matrix).

Numerical Integration

Numerical integration is the process of numerically computing a definite integral. There are many occasions where numerical integration is important. Examples include calculating the area of a shape or the area under a graph, as well as solving differential equations.

For this discussion, we'll calculate the area of half a circle of radius 1. We already know this area to be $\pi/2$. So in a sense, calculating the area of half a circle is equivalent to calculating the numerical value of π.

First, we create two vectors: `x` and `y`. These two vectors satisfy the circle equation $x^2 + y^2 = 1$. I assume you've already imported *PyLab*:

```
>>> N = 7
>>> x = linspace(-1, 1, N)
>>> y = sqrt(1-x**2)
>>> x**2 + y**2
```

```
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.])
```

I chose the variable N arbitrarily; N is the number of points in the vectors x and y. The last result shows that all the points in vectors x and y that satisfy the circle equation, $x^2+y^2=1$.

To visualize the numerical integration, I plot rectangles that approximate the area of the circle:

```
>>> figure()
>>> dx = x[1]-x[0]
>>> for i in range(len(x)-1):
...     rect = Rectangle((x[i], 0), dx, 0.5*(y[i]+y[i+1]))
...     gca().add_patch(rect)
...
>>> title('Approximating the area of half a circle')
>>> axis('equal')
>>> show()
```

The area under the curve—that is, the integral—is approximately the sum of these squares. Each square's area is (y[i] + y[i + 1])·dx/2, so the total sum can be written as follows:

```
>>> dx*(sum(y[:-1]+y[1:]))
```

```
2.9175533787759904
```

I've multiplied the result by 2, so we can compare with π instead of π /2. Obviously, the bigger N is, the closer this number will be to π:

```
>>> for N in [5, 10, 20, 100]:
...     x = linspace(-1, 1, N)
...     dx = x[1]-x[0]
...     y = sqrt(1-x**2)
...     est_pi = dx*sum(y[:-1]+y[1:])
...     print("N=%d, estimated pi is %f" % (N, est_pi))
...
```

```
N=5, estimated pi is 2.732051
N=10, estimated pi is 3.019232
N=20, estimated pi is 3.101560
N=100, estimated pi is 3.138218
```

As you can see, for N = 100, the accuracy is about 1 percent. Figure 8-1 captures this visually (see Appendix A for the full listing.)

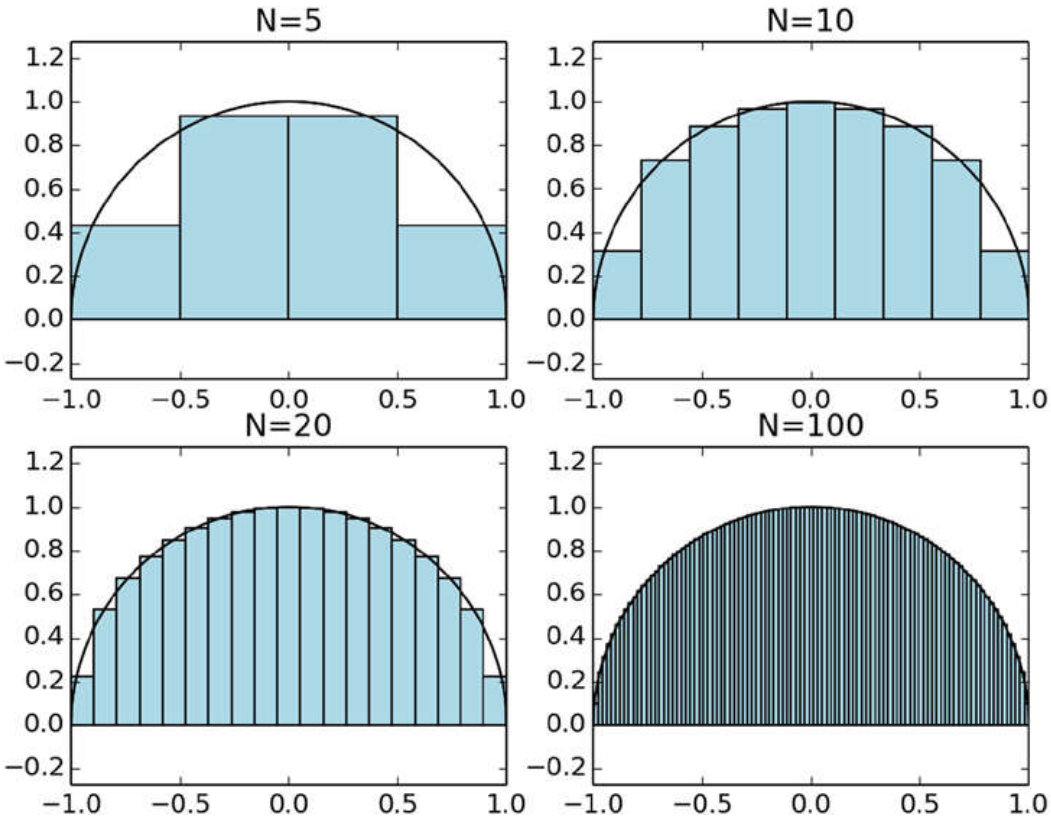

Figure 8-1. *Calculating the area of a circle*

In calculating the area of the circle, I chose values that are evenly spaced. If you'd like to use unevenly spaced values, the implementation is more complex. Also, the method uses rectangles to approximate the area under the curve, but in this particular example (and many others), trapezoidals are probably better suited, which brings us to the function, `trapz(y, x)`. The function accepts vectors `y` and `x` and returns the numerical integral. The following performs numerical integration of unevenly spaced `x` values using the function, `trapz()`:

```
>>> x = array([-1, -0.9, -0.4, 0.0, 0.4, 0.9, 1])
>>> y = sqrt(1-x**2)
>>> trapz(y, x)*2
```

```
2.9727951234089831
```

Figure 8-2 shows a visual representation of the trapezoidal integration (see Appendix A for the full listing).

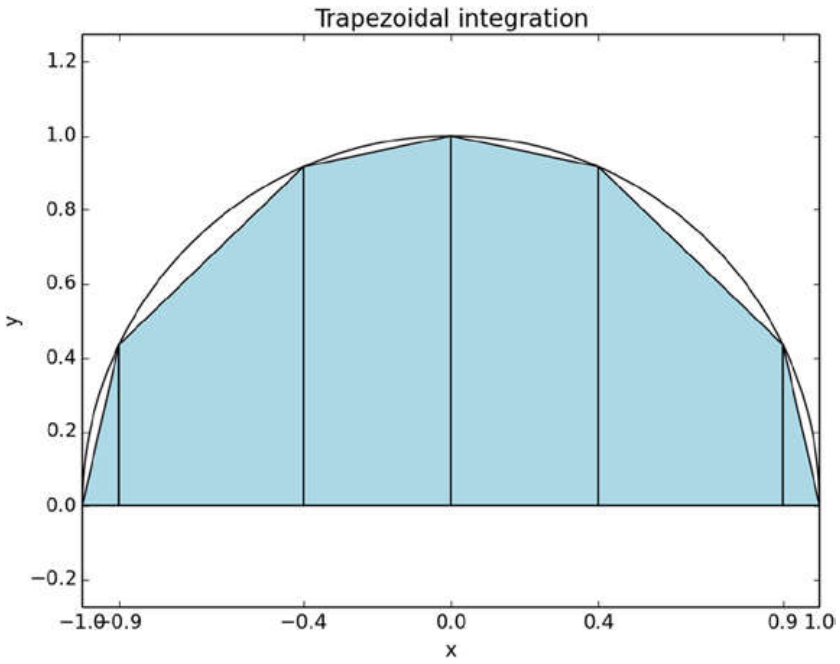

Figure 8-2. *Calculating the area of a circle using the trapezoidal method and unevenly spaced values*

More Integration Methods

Additional integration algorithms are available with the module, *scipy.integrate*. To use this module, issue `from scipy import integrate`.

We'll limit our discussion to the algorithm `quad()`, which uses a Gaussian quadrature to numerically integrate a *mathematical function*. Unlike previous methods such as `trapz()`, using `quad()` requires supplying a mathematical function and not the `x` and `y` vectors.

■ **Note** I've used the term "mathematical function" to differentiate this type of function from a general-purpose Python function. A mathematical function is one that returns a numerical value given an input numerical value, such as y = f(x). In reality, we implement a mathematical function as a Python function.

```
>>> from scipy.integrate import quad
>>> def half_circle(x):
...     return sqrt(1-x**2)
...
>>> pi_half, err = quad(half_circle, -1, 1)
>>>  (pi_half*2, err)
```

```
(3.1415926535897967, 1.0002354500215915e-009)
```

I defined a mathematical function `half_circle()` that returns the y-coordinate value of the upper half circle of radius 1, given an x-coordinate value. I then called `quad()` with the arguments `half_circle()`; the function to integrate; and `-1` and `1`, the range of values to integrate. The function `quad()` returns a value and an error.

The module *scipy.integrate* also supports the solving of ordinary differential equations using functions `ode()` and `odeint()`. We will not be discussing these functions. If you're interested in solving differential equations, refer to the *SciPy* home page: http://www.scipy.org/.

Interpolation and Curve Fitting

Interpolation and curve fitting deal with fitting functions to discrete known values. There are several reasons you would want to fit functions to points of data, including the following:

- Fitting a known function to gathered experimental data. This can be helpful in determining other parameters of the experiment.
- Evaluating the numerical values of functions at additional points (other than the given ones).

Interpolation allows efficient implementations that are tailor-made to a specific problem. Instead of writing a lookup table for all the possible values, you could come up with an interpolation polynomial that is more efficient, albeit with a possible loss of accuracy. At other times, you might choose to implement a known function like `sqrt()` instead of using a library-supplied algorithm to *increase* performance (at the possible cost of accuracy).

INVERSE SQUARE ROOT AND QUAKE III

If you're interested in efficient algorithms to calculate numerical functions, you may find this article of value: "Fast Inverse Square Root" by Chris Lomont at http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf. The article describes a very efficient algorithm to implement the inverse square root of a number that appeared in the source code of the computer game Quake III. The implementation uses the Newton-Raphson method (and not interpolation). The article assumes knowledge of C.

Piecewise Linear Interpolation

Let's turn back to our half-a-circle example. This time, we'll limit ourselves to a quarter of a circle; that is, to positive values of x and y. We start by calculating the y values for x equal to 0, 0.2, . . . , 1. We'll store the results in vectors `xp` and `yp`:

```
>>> xp = linspace(0, 1, 6)
>>> xp
```

```
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
```

```
>>> yp = sqrt(1-xp**2)
```

We'd like to calculate the values of y for x values equal to 0.1, 0.3, . . . , 0.9, given `xp` and `yp`. We'll use the function `interp(x, xp, yp)` for this. The function returns the value of the piecewise-linear function defined by `xp`, `yp` at a requested point, `x`. What this means is `interp()` returns the value of a point on a line connecting two adjacent (`xp`, `yp`) points. This is known as piecewise linear interpolation:

```
>>> xi = arange(0.1, 1.0, 0.2)
>>> yi = interp(xi, xp, yp)
```

The vector `yi` holds the interpolated values at points 0.1, 0.3, . . ., 0.9.
The following visualizes a piecewise linear interpolation for the quarter of a circle:

```
>>> from pylab import *
>>> figure()
>>> hold(True)
>>> x = linspace(0, 1, 500)
>>> y = sqrt(1-x**2)
>>> xp = linspace(0, 1, 6)
>>> yp = sqrt(1-xp**2)
>>> xi = arange(0.1, 1.0, 0.2)
>>> yi = interp(xi, xp, yp)
>>> plot(x, y, 'b', label='ideal')
>>> plot(xp, yp, 'or', label='interpolation points')
>>> plot(xp, yp, '--r', label='piecewise linear function')
>>> plot(xi, yi, 'sg', label='interpolated values')
>>> legend(loc='best')
>>> grid()
>>> axis('scaled')
>>> axis([0, 1.1, 0, 1.1])
>>> title('Piecewise linear interpolation')
>>> show()
```

Figure 8-3 shows the results of this visualization.

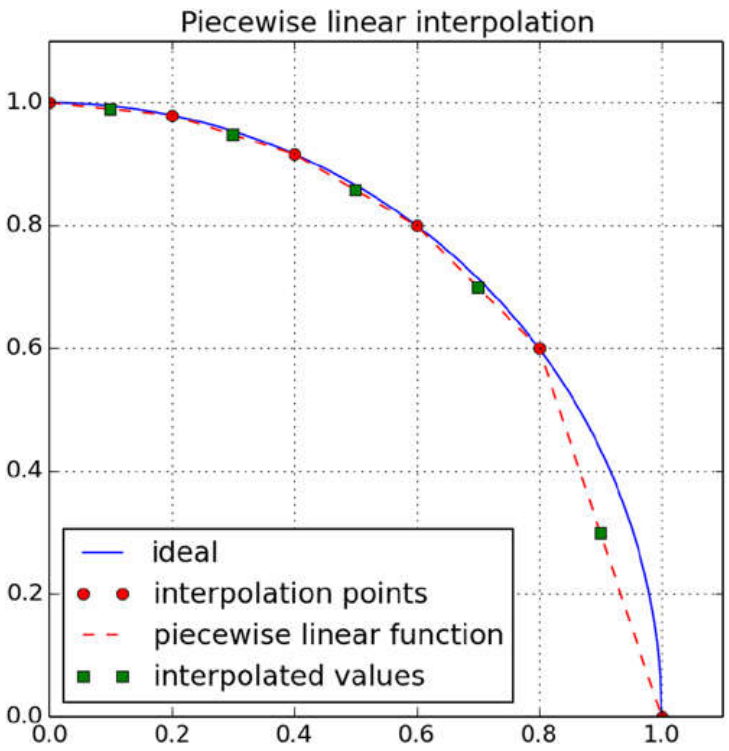

Figure 8-3. *Piecewise linear interpolation*

The values xp and yp are computed in this example; but in reality, these values can originate from sampled data. As you can see from the graph, the interpolated value at 0.9 is considerably less accurate than other interpolated values. Typically, the more points you add, the more accurate the result.

Polynomials

Polynomials are mathematical expressions that involve a sum of integer powers of a variable multiplied by a coefficient. Examples include $2x^2 + x - 1$, as well as x. However, sin(x) is not a polynomial. The reason polynomials are so important is that they involve only basic operations: addition, subtraction, and multiplication (integer power can be implemented with several multiplications). This property makes them very easy to implement in computing. Taylor series expansion (http://en.wikipedia.org/wiki/Taylor_series) is a prime example of transforming a function to a polynomial that is easily computed.

To be able to operate on polynomials with *NumPy* and *SciPy*, we represent a polynomial as a vector. The first element in the vector is the coefficient of the highest power, and the last element in the array is the coefficient of the lowest power, 0. So to express the polynomial $x^2 + 3x + 2$, issue the following:

```
>>> p = array([1, 3, 2])
```

To solve the equation $x^2 + 3x + 2 = 0$, use the function `roots(p)`:

```
>>> roots(p)
```

```
array([-2., -1.])
```

If you'd like to construct a polynomial from its roots instead of its coefficients, use the function poly():

```
>>> p = poly([-2, -1])
>>> p
```

```
array([1, 3, 2])
```

You add and subtract polynomials using `polyadd()` and `polysub()`:

```
>>> p1 = poly([-2, -1])
>>> p2 = array([1, 0, 0, 0])
>>> polyadd(p1, p2)
```

```
array([1, 1, 3, 2])
```

I've added $x^2 + 3x + 2$ to x^3 and got $x^3 + x^2 + 3x + 2$ as a result.

Multiplying and dividing polynomials is done using `polymul()` and `polydiv()`. The return value from `polydiv()` is a quotient and a remainder:

```
>>> p = polymul(array([1, 2]), array([1, 3]))
>>> p
```

```
array([1, 5, 6])
```

```
>>> polydiv(p, array([1, 3]))
```

```
(array([ 1.,  2.]), array([0]))
```

You perform integration and differentiation on polynomials using the functions `polyint()` and `polyder()`, respectively:

```
>>> p = poly([-1j, 1j])
>>> p
```

```
array([ 1.,  0.,  1.])
```

```
>>> polyder(p)
```

```
array([ 2.,  0.])
```

```
>>> polyint(p)
```

```
array([ 0.33333333,  0. ,  1.  ,  0.  ])
```

In the first line, I created a polynomial from complex numbers; the polynomial created is stored in `p` and is $x^2 + 1$. Using `polyder()`, I calculated the derivative of `p` and got 2x. Using `polyint()`, I calculated the integral of `p` and got $\frac{1}{3}x^3 + x$.

Uses of Polynomials

So why is all this polystuff important? The main reason is that you can use polynomials to approximate functions both from gathered data and from analytical functions. And since polynomials only require multiplications and additions, implementing polynomials in an embedded system, for example, is straightforward.

You fit polynomials to data using the function, `polyfit(x, y, n)`. Given a vector of x points and a vector of y points, `polyfit()` will return a polynomial of degree n (highest power of x) that best fits the set of data points. Another useful function is `polyval(p, x)`; this function returns the value of the polynomial at `x` (`x` can be a vector).

Example: Linear Regression

A known curve-fitting algorithm is linear regression. The idea is to draw a straight line in such a way that the total distance of all the points from the line is minimal.

For this example, we'll create a straight line and then add "measurement noise" to the values. Confronted with the new, "noisy" data, we'll try to evaluate the first order polynomial that fits the data. We'll compare the results with the known true values (see Listing 8-1).

Listing 8-1. Linear Regression with `polyfit()`

```
from pylab import *

# number of data points
N     = 100
```

```
start = 0
end   = 1

A = rand()+1
B = rand()

# our linear line will be y = A*x + B

x = linspace(start, end, N)
y = A*x + B
y += randn(N)/10

# linear regression
p = polyfit(x, y, 1)

figure()
title('Linear regression with polyfit()')
plot(x, y, 'o',
    label='Measured data; A=%.2f, B=%.2f' % (A, B))
plot(x, polyval(p, x), '-',
    label='Linear regression; A=%.2f, B=%.2f' % tuple(p))
legend(loc='best')
show()
```

I've randomly selected values for A (from 1 to 2) and B (from 0 to 1), and then constructed a linear line with noise using `randn()`. Next, I used `polyfit()` to fit the data to a first degree polynomial, a straight line. Lastly, I plotted the data along with the newly constructed linear line. Figure 8-4 shows the results of this linear regression.

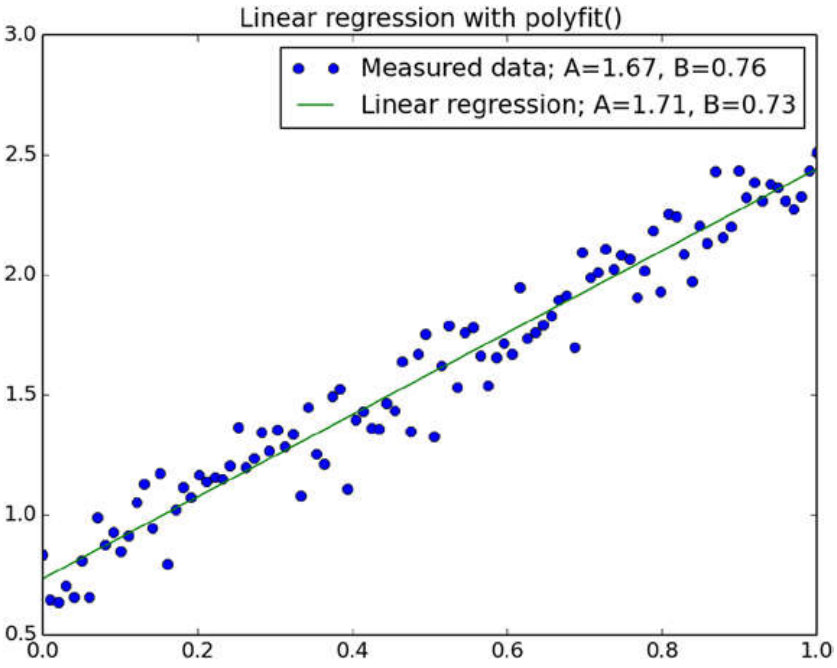

Figure 8-4. *A linear regression with* `polyfit()`

Example: Linear Regression of Nonlinear Functions

In cases where the function you're trying to fit isn't linear, sometimes it's still possible to perform linear regression, as shown in Listing 8-2.

Listing 8-2. Fitting Exponential Data

```
from pylab import *

# number of data points
N     = 100
start = 0
end   = 2

A = rand()+0.5
B = rand()

# our linear line will be:
# y = B*exp(A*x) = exp(A*x + log(B))

x = linspace(start, end, N)
y = exp(A*x+B)
y += randn(N)/5

# linear regression
p = polyfit(x, log(y), 1)

figure()
title(r'Linear regression with polyfit(), $y=Be^{Ax}$')
plot(x, y, 'o',
    label='Measured data; A=%.2f, B=%.2f' % (A, exp(B)))
plot(x, exp(polyval(p, x)), '-',
    label='Linear regression; A=%.2f, B=%.2f' % (p[0], exp(p[1])))
legend(loc='best')
show()
```

The regression is performed in the call to the function, `polyfit()`. This time, I've passed `x` and `log(y)` as values allowing a linear regression on `log(y)` or an exponential regression on `y`. You can see the results of this regression in Figure 8-5.

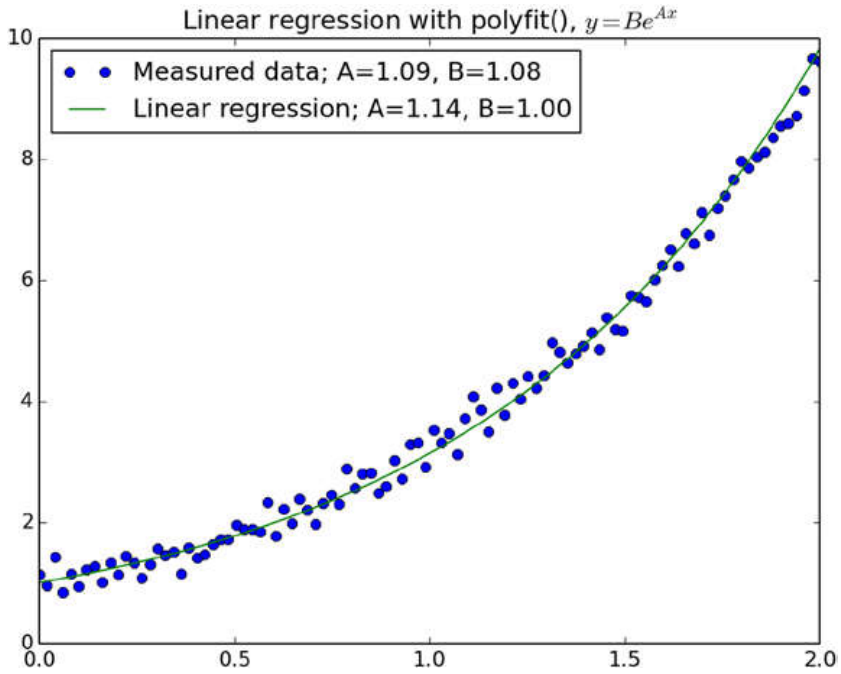

Figure 8-5. *Fitting exponential data*

Example: Approximating Functions with Polynomials

Another set of problems solvable with `polyfit()` is the approximation of functions using interpolation. The motivation behind this is a simple implementation of known functions. For the purpose of this example, we'll approximate the function sin(x).

The idea is to create a polynomial that passes through known interpolation points—that is, to calculate the value of sin(x) for known n values of x, and then to create a polynomial of degree n–1 that passes through all these points.

We start by selecting a set of points from 0 to $\pi/2$; these will be our interpolation points. Values outside this range can be computed using trigonometry identities and the interpolation function. We select five points for interpolation, thus deciding the degree of the interpolation polynomial to be 4. Once the points are selected, we calculate the sine of these points.

For the purpose of this example, I've chosen sine values that can be easily computed using the `sqrt()` function. You might argue that I'm cheating here because I'm using a nonlinear function (square root) to calculate sin(x) and not pure polynomials. However, you've already seen how to calculate the square root of a number using Newton's method in Chapter 7.

■ **Note** The selection of interpolation points is an interesting topic, and work by the mathematician Pafnuty Chebyshev has contributed much to the topic. See `http://en.wikipedia.org/wiki/Pafnuty_Chebyshev` and `http://en.wikipedia.org/wiki/Chebyshev_nodes`.

The values I'll select for interpolation are 0, 30, 45, 60, and 90 degrees. The reason I chose these values is that I know their exact sine values: 0, ½, √2/2, √3/2, and 1, respectively. In vector form, that looks like this:

```
>>> values = [0, pi/6, pi/4, pi/3, pi/2]
>>> sines = sqrt(arange(5))/2
>>> sines
```

```
array([ 0. , 0.5, 0.70710678, 0.8660254, 1. ])
```

Given these, interpolation is straightforward:

```
>>> p = polyfit(values, sines, len(values)-1)
>>> p
```

```
array([  2.87971125e-02,  -2.04340696e-01,   2.13730075e-02,
         9.95626184e-01,   1.52055217e-16])
```

So if you were to implement sin(x), all you need is to store the values of `p` given previously and then write a simple routine to calculate the value of sin(x) using the polynomial. If you're using *NumPy*, simply call `polyval()`.

Let's plot the difference between our implementation of sin(x) and Python's built-in sin(x) function:

```
>>> figure()
>>> x = linspace(0, pi/2, 100)
>>> plot(x, polyval(p, x)-sin(x), label='error', lw=3)
>>> grid()
>>> ylabel('polyval(p, x)-sin(x)')
>>> xlabel('x')
>>> title('Error approximating sin(x) using polyfit()')
>>> xlim(0, pi/2)
>>> show()
```

Figure 8-6 illustrates this difference.

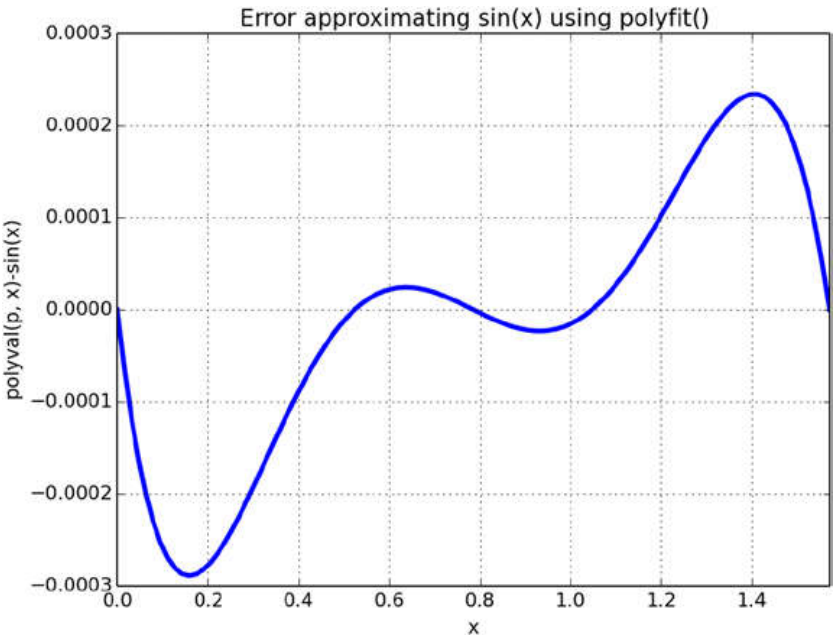

Figure 8-6. *Interpolation accuracy*

The results are quite accurate; the absolute error is less than 0.003.

Spline Interpolation

The *scipy.interpolate* module adds additional interpolation functions. One of these is the `spline(xp, yp, x)` interpolation function. Notice that the arguments to the function `spline()` are ordered differently from those of the function `interp()`. Spline interpolation is a piecewise polynomial interpolation that adheres to specific rules to yield smooth results.

Let's turn to the previous circle example:

```
from scipy.interpolate import spline
from pylab import *

xp = linspace(0, 1, 6)
yp = sqrt(1-xp**2)
xi = linspace(0, 1, 100)
yi = interp(xi, xp, yp)
ys = spline(xp, yp, xi)
figure()
hold(True)
plot(xi, yi, '--', label='piecewise linear', lw=2)
plot(xi, ys, '-', label='spline', lw=2)
legend(loc='best')
grid()
title(r'Spline interpolation of $y=\sqrt{1-x^2}$')
xlabel('x')
ylabel('y')
axis('scaled')
axis([0, 1.2, 0, 1.2])
show()
```

In Figure 8-7, I've compared a piecewise linear interpolation with a spline interpolation. The spline interpolation appears "smoother."

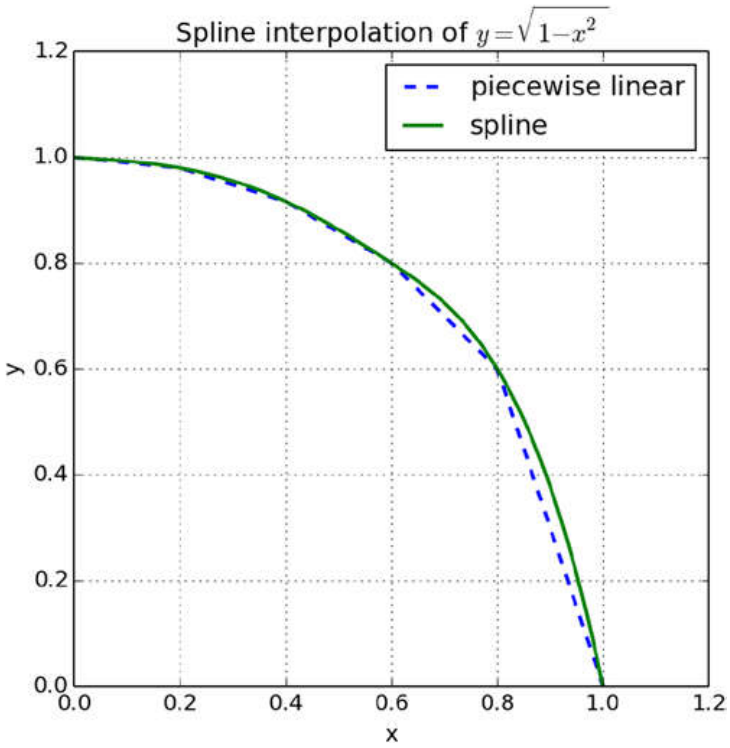

Figure 8-7. *A spline interpolation*

Solving Nonlinear Equations

In Chapter 7 we've talked about Newton's method and used it to draw fractals. Newton's method was used to solve a nonlinear equation.

The module *scipy.optimize* provides us with additional tools to solve nonlinear equations, as well as other optimization routines that will not be discussed here. Of those routines, I'd like to highlight three: `fsolve(f, x0)`, `bisect(f, a, b)`, and `newton(f, x0)`. All these functions try to solve the equation f = 0, where f is a mathematical function implemented in Python.

Suppose we'd like to calculate √3 for the previous example. The idea is to construct a function such that the solution will be √3. This is easily done by setting $f = x^2 - 3$:

```
>>> def f(x):
...     """Returns x**2-3"""
...     return x**2-3
...
>>> f(10)
```

Let's use the functions `fsolve()`, `bisect()`, and `newton()` to calculate the roots. For `fsolve()` and `newton()`, we'll use x0 = 1, which is called the initial guess. The initial guess is a value that is close to the desired result. For `bisect()`, we need to provide a region for the search. We'll set the region to (1, 2) because we know the square root of 3 is less than 2 but greater than 1:

```
>>> from scipy import optimize
>>> optimize.fsolve(f, 1)
```

```
array([ 1.73205081])
```

```
>>> optimize.newton(f, 1)
```

```
1.7320508075688772
```

```
>>> optimize.bisect(f, 1, 2)
```

```
1.7320508075690668
```

```
>>> _**2
```

```
3.0000000000006564
```

Although in the simple case of square root of 3, all these functions provide accurate results, the algorithms are computationally intensive. In most these functions you can control how accurate you'd like your result to be by passing proper arguments to the functions. Of course, for a question as simple as the one presented here, it's best to use `sqrt(3)`.

Special Functions

The *scipy.special* module provides a host of special functions that surface in higher mathematics and physics. These include the following (and many more):

- Bessel functions, integrals, derivatives, and zeros of Bessel functions
- Airy functions
- Gamma functions and error functions
- Special polynomials: Legendre, Chebyshev

To use the functions, issue the following:

```
>>> from scipy import special
>>> special.chebyt(2)
```

```
poly1d([  2.00000000e+00,  -4.44089210e-16,  -1.00000000e+00])
```

For a full account, issue `help(special)`.

Signal Processing

Up to this point in the chapter, we've dealt with numerical analysis. Going forward, the topics are related to signal processing. Signal processing is a vast field that deals with signals: values that change over time. Popular signal processing algorithms include the processing of sound, such as an equalizer; others include algorithms for radars, CAT scanning, and many more.

This part of the chapter will cover some of the functionality available with the module *scipy.signal* and complement the discussion with examples. You'll learn about some basic algorithms to detect signals in the presence of noise, as well as some functions to design filters. However, this section is but a taste of the topic, and I encourage you to consult with the references at the end of the chapter and professional literature for efficient signal processing algorithms.

Functions find, nonzero, where and select

The first set of functions we'll cover is `find()`, `nonzero()`, `where()`, and `select()`.

The function `find(cond)` finds the indices to an array for which a condition is met:

```
>>> from pylab import *
>>> squares = arange(10)**2
>>> squares
```

```
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

```
>>> I = find(squares<50)
>>> I
```

```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

```
>>> squares[I]
```

```
array([ 0,  1,  4,  9, 16, 25, 36, 49])
```

We created a vector holding the squares of the numbers 0-9, and then found all the indices to the vector that satisfy the condition that the values are less than 50. Notice that the return value is a vector of indices; if you require the values and not the indices, you have to access the original array, which is `squares[I]` in the preceding example.

The function `nonzero(seq)` returns nonzero elements of seq. This is similar to `find(seq!=0)`. However, the function `nonzero()` returns a 2-D matrix, whereas `find()` returns indices to the flattened sequence, `seq`. This makes it hard to use `find()` in the kind of 2-D matrices used in images; in that case, `nonzero()` is the better option:

```
>>> A = eye(3)
>>> A
```

```
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

```
>>> I1 = find(A!=0)
>>> I1
```

```
array([0, 4, 8])
```

```
>>> A[I1]
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index 4 is out of bounds for axis 0 with size 3
```

```
>>> A.ravel()[I1]
```

```
array([ 1.,  1.,  1.])
```

```
>>> I2 = nonzero(A)
>>> A[I2] = 3
>>> A
```

```
array([[ 3.,  0.,  0.],
       [ 0.,  3.,  0.],
       [ 0.,  0.,  3.]])
```

I've created a 2-D matrix, A, and used `find(A!=0)` to find nonzero elements in A. Notice that the return value, `I1`, is a 1-D array; this means that indexing a 2-D array. Therefore, issuing `A[I1]` raises and exception. To access the values of A, I must first flatten the matrix A using a call to `ravel()`. However, if I use the function `nonzero()`, I can access 2-D elements seamlessly, as shown in the code of line `A[I2]=3`.

The function `where(cond, x, y)` accepts three arrays of the same size: `cond`, `x`, and `y`, and then evaluates every element in `cond`. If the element evaluates to `True`, the return value is the corresponding element from `x`. If the return element evaluates to `False`, the return value is the corresponding element from `y`:

```
>>> up = arange(10)
>>> up
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
>>> down = arange(10, 0, -1)
>>> down
```

```
array([10,  9,  8,  7,  6,  5,  4,  3,  2,  1])
```

```
>>> highest = where(up > down, up, down)
>>> highest
```

```
array([10,  9,  8,  7,  6,  5,  6,  7,  8,  9])
```

The function `select(cond, vals, default=0)` adds functionality to the function `where()` by allowing for several conditions. The function accepts a list of conditions specified in `cond` and returns the corresponding element associated with `vals` if a condition is met; if none of the conditions is met, the `default` value is selected:

```
>>> up = arange(10)
>>> ramp = select([up < 4, up > 7], [4, 7], up)
>>> ramp
```

```
array([4, 4, 4, 4, 4, 5, 6, 7, 7, 7])
```

The first three elements of `up` are less than 4, so the condition `up < 4` is met, causing the selection of value 4. The last three elements are greater than 7, causing the selection of the value 7. Values greater than or equal to 4 yet less than 7 are retained as-is because the default is set to be equal to `up`. As a matter of fact, this functionality is called *clipping* and is available as both a method of the *NumPy* ndarray object and as a stand-alone function, `clip()`.

So now that you know about the functions `find()`, `nonzero()`, `where()`, and `select()`, what can you do with them? The answer is simple: they're great for picking up values, what we call *detection* in signal processing.

Example: Simple Detection of Signal in Noise, Part 1

The detection of signals in the presence of noise plays an integral role in a great number of applications. For example, it is used in communication systems in the detection of signals such as radio or television broadcasts and to help differentiate them from noise. It is also used in medicine with the detection of an ECG signal, as well as in many other fields.

For this example, we'll first construct a clean signal. By a signal, I mean a one-dimensional array (vector), where values are stored as a function of time. Our purpose will be to detect "events," which will be represented by narrow triangles placed randomly in time. There can be several events in a signal.

To generate a triangular pulse, I'll use the `signal.triang()` function. This is really a window function (you'll learn more about such functions in the "Window Functions" section of this chapter). The function generates a triangular window of a specified size. We randomly place triangular pulses in the signal vector, as shown in Listing 8-3.

Listing 8-3. Randomly Placing Triangular Spikes

```
from pylab import *
from scipy import signal

# parameters controlling the signal
n = 100
t = arange(n)
y = zeros(n)
num_pulses = 3
pw = 11
amp = 20

for i in range(num_pulses):
    loc = floor(rand()*(n-pw+1))
    y[loc:loc+pw] = signal.triang(pw)*amp

# add some noise
y += randn(n)
```

```
figure()
title('Signal and noise')
xlabel('t')
ylabel('y')
plot(t, y)
show()
```

First, I defined some parameters I'll be using. The number of points in the signal is `n` and is equal to 100. The number of triangular pulses I'll place is 3, denoted by `num_pulses`. Each triangular pulse will be generated using `pw=11` points. The maximum value for the triangular spike will be `amp`, which denotes amplitude.

Once I have all the parameters defined, I create two vectors: the time vector, `t`, and the values vector, `y`. The vector `t` is some arbitrary timestamp; in this example, it increments values starting at zero and ending at `n–1`. The vector `y` is initially set at zero.

Next I randomly place triangular spikes. The location, `loc`, where the triangular spike will be placed, is randomly generated with the call to the function `rand()` that generates a value between 0 and 1. So, I randomly pick a value between 0 and n – pw + 1 to ensure spikes aren't placed outside the vector `y`. Once I have all the spikes placed, I add noise with the function `randn()`, which generates normally distributed noise. This is also known as Gaussian distribution, or "white noise." I've chosen to use a normal distribution with variance 1 and mean 0. Notice that `randn()` is different from `rand()`.

Figure 8-8 shows the generated signal.

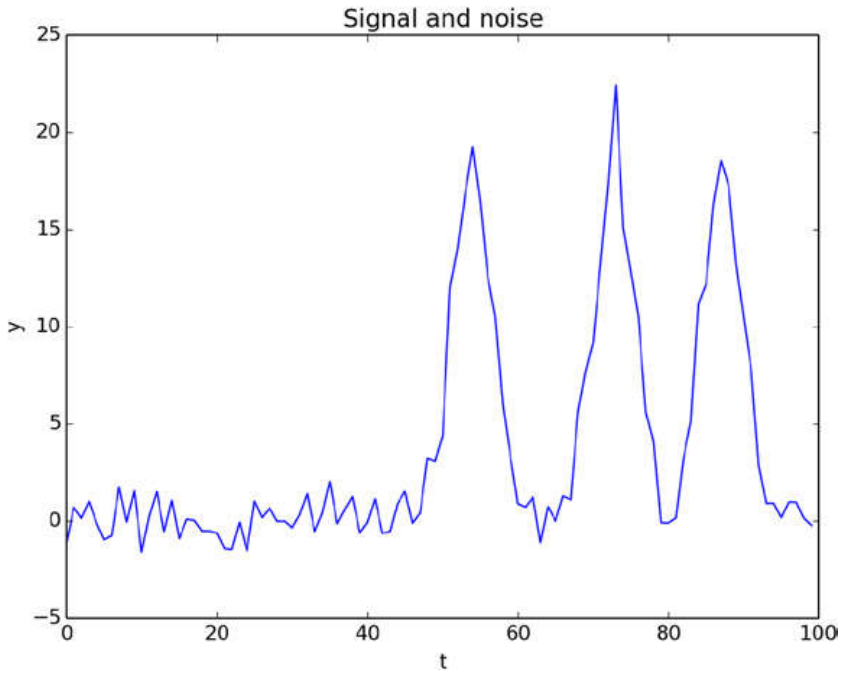

Figure 8-8. *Three triangular spikes with noise*

I did not check to see that spikes do not overlap. So, as you run the script, sometimes you'll view one or two spikes instead of three. This is fine, since we want to add some randomness to the example.

So far we've just created the signal. Now let's detect it. For detection, we'll use a simple algorithm: whenever a value is above a set threshold, we'll declare this as an event, or detection. We'll set the threshold at `amp/2` and use the function `find()`, as shown in Listing 8-4.

Listing 8-4. Detecting Signals (a Continuation of Listing 8-3)

```
# detect signals
thr = amp/2
I = find(y > thr)

# plot signal with noise plus detection
figure()
hold(True)
plot(t, y, 'b', label='signal with noise')
plot(t[I], y[I], 'ro', label='detections')
plot([0, n], [thr, thr], 'g--')

# annotate the threshold
text(2, thr+.2, 'Threshold', va='bottom')

title('Simple signal detection in noise')
legend(loc='best')
show()
```

Figure 8-9 shows the result.

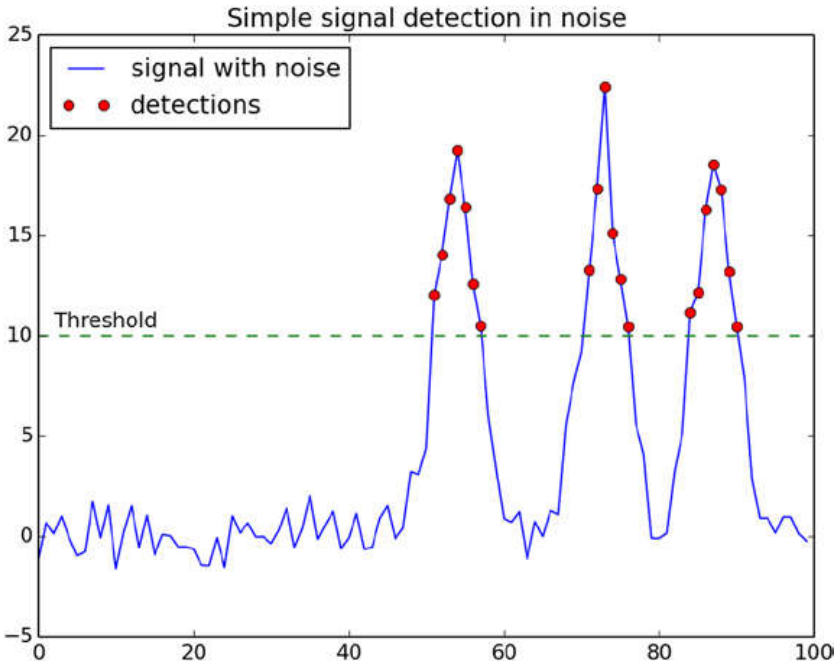

Figure 8-9. *Simple signal detection in the presence of noise*

Functions diff and split

Another set of functions that's useful in signal detection is `diff()` and `split()`. The function `diff(v)`, which was introduced in previous chapters, returns a vector composed of the differences of the elements in `v`. The function `split(v, indices)` splits a vector on `indices`:

```
>>> v = arange(10)
>>> split(v, [4, 8])
```

```
[array([0, 1, 2, 3]), array([4, 5, 6, 7]), array([8, 9])]
```

Example: Simple Detection of Signal in Noise, Part 2

In the previous example, you saw how to perform simple detection using `find()`. We've displayed all points that were above a specific threshold. In many occasions, we're less interested with points above a threshold because the threshold is arbitrarily chosen; we're more interested with the highest points above a threshold.

Here we pick up from the previous example. This time, we'd like to spot the peak in each detection. Listing 8-5 presents the code to do that.

Listing 8-5. Peak Detections

```
# peak detections
J = find(diff(I) > 1)
for K in split(I, J+1):
    ytag = y[K]
    peak = find(ytag==max(ytag))
    plot(peak+K[0], ytag[peak], 'sg', ms=7)
```

The implementation is a bit tricky, so let's walk through it. The idea is this: we split the detections into separate groups; and in each group, we find the peak and plot it.

■ **Note** Because the both noise and signal are randomly selected, your actual numerical values might differ from those presented in this section. Nevertheless, the concepts and ideas are still valid.

The first problem of splitting detections uses the indices of detected values. A group is considered one detection if the indices are consecutive. Whenever there's a jump in indices, it means a new group:

```
>>> I = find(y > thr)
>>> I
```

```
array([51, 52, 53, 54, 55, 56, 57, 71, 72, 73, 74, 75, 76, 84, 85, 86, 87, 88, 89, 90])
```

So the group `[51, 52, 53, 54, 55, 56, 57]` is one group, the group `[71, 72, 73, 74, 75, 76]` is the second group, and the group `[84, 85, 86, 87, 88, 89, 90]` is the last group.

The function `diff(I)` returns values other than 1 whenever there's a new group. Whenever the difference is greater than 1, it means the start of a new group:

```
>>> diff(I)
```

```
array([ 1,  1,  1,  1,  1,  1, 14,  1,  1,  1,  1,  1,  8,  1,
    1,  1,  1,  1,  1])
```

```
>>> J = find(diff(I)>1)
>>> J
```

```
array([ 6, 12])
```

So we'd like to split on the sixth element (denoted by 6) and the twelfth element (denoted by 12). We do this with the `split()` function:

```
>>> split(I, J+1)
```

```
[array([51, 52, 53, 54, 55, 56, 57]),
 array([71, 72, 73, 74, 75, 76]),
 array([84, 85, 86, 87, 88, 89, 90])]
```

All we need to do now is find the peak, which is coded as `find(ytag == max(ytag))`. In Figure 8-10, peak detections are marked by squares.

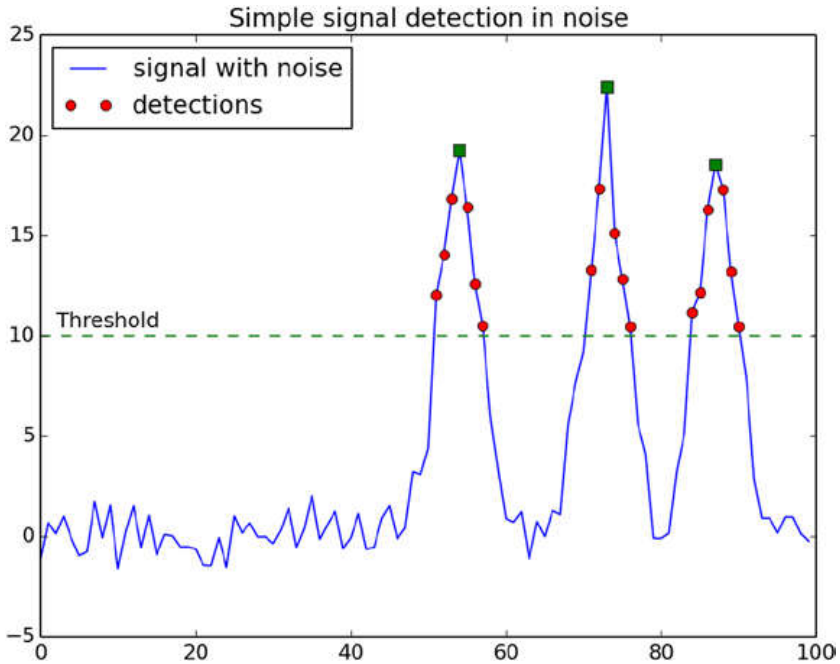

Figure 8-10. *Peak detections*

Waveforms

Additional *SciPy* functionality includes several waveforms that can be used when you're designing a signal processing algorithm or testing it. These include `sawtooth()`, `square()`, `gausspulse()`, and `chirp()`:

```
from pylab import *
from scipy import signal

cycles = 10
t = arange(0, 2*pi*cycles, pi/10)

waveforms = ['sawtooth', 'square']

figure()
for i, waveform in enumerate(waveforms):
    subplot(2, 2, i+1)
    exec('y = signal.' + waveform + '(t)')
    plot(t, y)
    title(waveform)
    axis([0, 2*pi*cycles, -1.1, 1.1])
show()
```

Figure 8-11 shows the resulting waveforms.

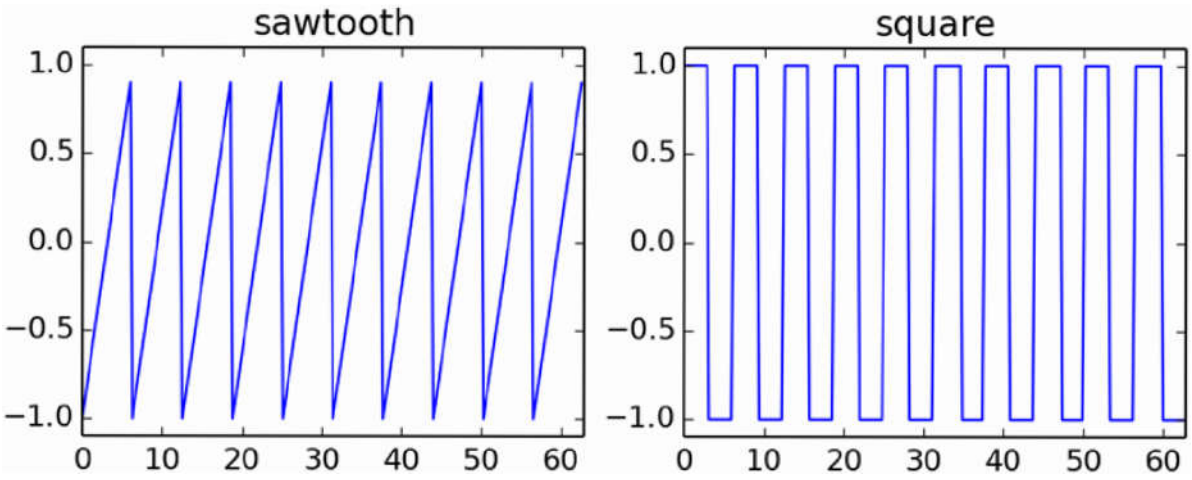

Figure 8-11. *Some waveforms*

The difference between waveforms and the triangular window used earlier is that they're repetitive, whereas `triang()` generates a single window.

The functions `gausspulse()` and `chirp()` are a bit more specialized; refer to the interactive help for information on using them.

Fourier Transform

Fourier transform is a linear operation that transforms a function from the time domain to the frequency domain. Much as the sound you hear can be viewed as an amplitude as a function of time, it can also be viewed by its frequency components: basses are the low frequencies of audio, for example.

The topic of Fourier transforms is quite large and requires some mathematical rigor. I will not be trying to address the topic in depth here; instead, I will show how you can use *PyLab* to perform Fourier transforms on sampled data.

To convert a signal from a time domain to a frequency domain, use `fft(x)`. FFT, which stands for Fast Fourier Transform, is an efficient implementation of the transformation. Generally speaking, if the number of elements in x is a power of 2, the results are quite fast:

```
>>> from time import time as t
>>> t1 = t(); dummy = sum(fft(arange(2**21))); print(t()-t1)
```

```
0.20280003547668457
```

```
>>> t1 = t(); dummy = sum(fft(arange(2**21-1))); print(t()-t1)
```

```
1.0452020168304443
```

(Your actual result might differ due to the processing power of your CPU). The first `fft()` was performed on a vector the size of 2^{21}, which is a power of 2; the second one was performed on a *shorter* vector, but it took longer to compute (more than twice as long, actually) because it is not a power of 2. In the example, I've used the function `time()` from module *time*, to measure the time the calculations take.

To transform data from the frequency domain to the time domain, use `ifft(x)`.

Example: FFT of a Sampled Cosine Wave

A cosine wave is made of one frequency (actually, two frequencies if you include the negative frequency). Let's generate a cosine wave and calculate its frequency using `fft()`, as shown in Listing 8-6.

Listing 8-6. Fourier Transform of a Cosine Wave

```
from pylab import *
from scipy import signal

N = 2**9   # we prefer powers of 2
F = 25     # a wave at 25 Hz
t = arange(N)/float(N)  # sampled over 1 second
x = cos(2*pi*t*F)   # the signal
figure()
p = subplot(2, 1, 1)
plot(t, x)
ylabel('x []')
xlabel('t [seconds]')
title('A cosine wave')
grid()
```

```
p = subplot(2, 1, 2)
f = t*N
xf = fft(x)
plot(f, abs(xf))
title('Fourier transform of a cosine wave')
xlabel('f [Hz]')
ylabel('xf []')
xlim([0, N])
grid()
show()
```

I began by defining a few parameters: `N` is the number of points in the signal, and `f` is the frequency of the cosine wave. I then created a time vector, `t`, which is composed of evenly spaced samples between 0 and 1, representing 1 second. I then calculated the sampled cosine wave and plotted it, along with its Fourier transform. I've chosen to plot the absolute of the transformed signal because Fourier transforms return complex values (albeit in this case those complex values are zero). Figure 8-12 shows the results.

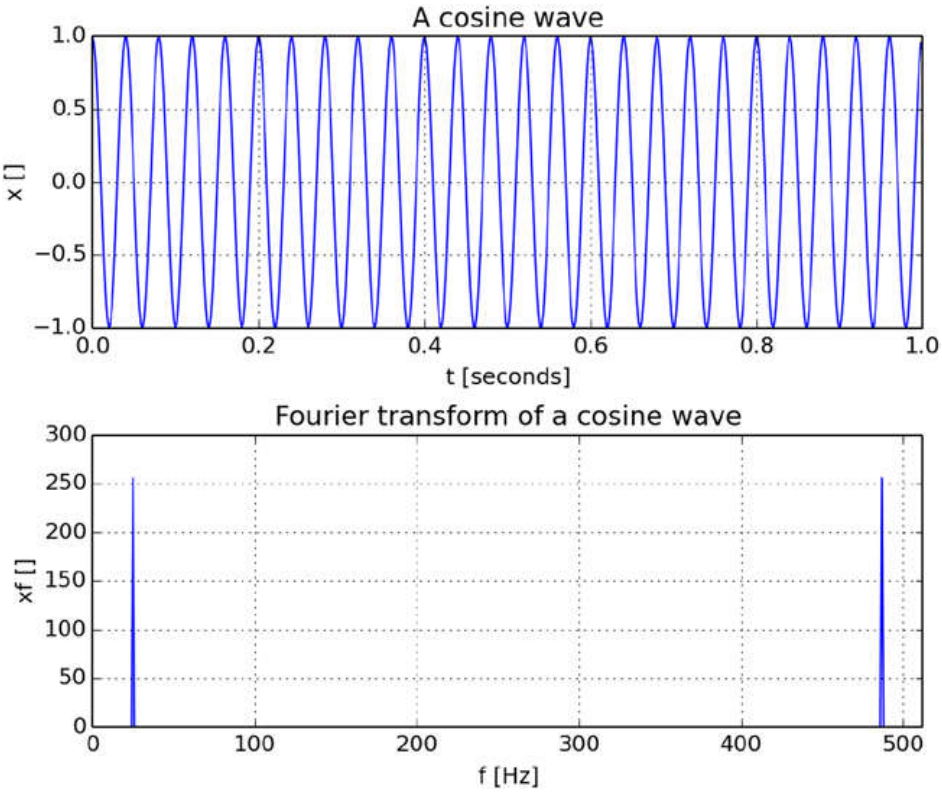

Figure 8-12. *FFT of a signal*

■ **Note** There's a frequency content at 25 Hz (the left spike), but there's also another one at 487 Hz. That's really the value corresponding to (–25) Hz; that is, (512 – 25) Hz. If you'd like to view the frequency domain centered around 0 Hz, use the function `fftshift()`.

Window Functions

In the FFT example, I carefully chose a cosine wave that will have a full number of cycles in 1 second, which is basically any integer number for the frequency value. If I had chosen a noninteger value, I would've ended up with a signal that does not have full wave cycles. The problem with this signal is that, when you perform the FFT of the signal, you'll start seeing other frequencies—and not just the frequency of your original signal. The reason for this is, in essence, that FFT assumes the signal is repetitive; that is, it's not just from 0 to 1 second, but from minus infinity to infinity. And so it treats the signal as if it's copied left and right an infinite number of times. If the signal has an integer number of cycles, it will nicely fit when copied left and right. But in reality, you can't guarantee an integer number of waves in your sampled signal, so you'll start seeing these sampling effects. To minimize the effect, we can use a window function.

Several window functions such as `hamming()`, `hanning()`, `bartlett()`, and `kaiser()` help minimize this effect, but at a cost: the signal itself is also distorted. To use a window, multiply it by the time-domain vector, as shown in Listing 8-7.

Listing 8-7. A Hamming Window

```
N = 2**9   # we prefer powers of 2
F = 25.5   # wave frequency
t = arange(N)/float(N)  # sampled over 1 second
f = t*N
x = cos(2*pi*t*F)   # the signal
xh = x*hamming(512) # multiply with a hamming window
figure()
plot(f, abs(fft(x)), 's-', label='original')
plot(f, abs(fft(xh)), 'o-', label='with Hamming')
xlim([0, 50])
xticks(arange(0, 55, 5))
legend()
grid()
title('Signal with Hamming window')
xlabel('Frequency [Hz]')
ylabel('Amplitude []')
show()
```

I've plotted the FFT of two vectors: the original and the one with a Hamming window. In Figure 8-13, you can see I've zoomed in on the 25.5 Hz frequency to show the effects of the window function.

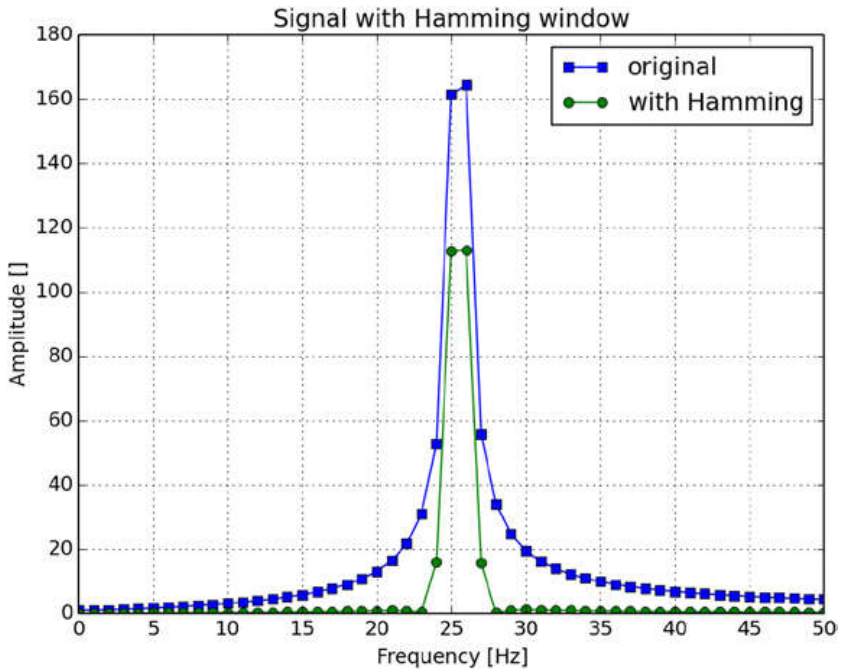

Figure 8-13. *Viewing a signal with a Hamming window*

The *scipy.signal* module provides additional window functions. To access these, issue the following:

```
>>> from scipy import signal
>>> help(signal)
```

Now scroll down to the window functions section.

Filtering

One of the reasons to transform a time-domain signal to a frequency-domain signal is that filtering in the frequency domain is, at times a, lot simpler. A filter is an operation that changes a signal. Much like filters in your kitchen sink, filters let some frequencies pass (water), while stopping other frequencies (large food remains). Filters are used in a variety of applications, ranging from audio to radar systems.

Filters are categorized by their behavior. A filter that lets through low frequencies and stops high frequencies is called a low-pass filter (LPF). Similarly, a high-pass filter (HPF) will allow only high frequencies to pass. There are also other categorizations, such as band-pass filters (allows only a specific band of frequencies), band-stop filters (allows anything but a specific band of frequencies), and notch filters (suppresses very few frequencies).

Filters are further categorized by their behavior to an impulse input; in other words, they are categorized by the output of the filter as a function of time, assuming you were to input a short spike to the filter. Filters that eventually forget the impulse are known as finite-impulse-response (FIR) filters, and filters that never forget are known as infinite-impulse-response (IIR) filters. From a very simplistic approach, if a filter does not rely on previous outputs (no feedback), it is considered an FIR; otherwise, it's an IIR.

Filter Design

Assuming you know what filter you wish to design, this section will help you do so. Filter design is an advanced topic (see http://en.wikipedia.org/wiki/Filter_design); and as such, this section is meant for those who require a few pointers on designing filters in Python with *SciPy*.

The *scipy.signal* module includes several functions to help design a filter. The function `iirdesign()` is used for designing an IIR filter. It is quite complete, and it's best to read the online help and follow it through. Other useful IIR design filters include `butter()`, `cheby1()`, `cheby2()`, and `ellip()`. FIR filter design functionality is provided with functions `remez()` and `firwin()`. I won't be covering those, but should you need to use them, the *SciPy* online help is quite informative. Finally, if you'd like to view the frequency response of a filter, use the functions `freqz()` and `freqs()`.

The code in Listing 8-8 creates a low-pass Butterworth filter (an IIR filter) and plots its frequency response.

Listing 8-8. The Frequency Response of a Filter

```
N = 256     # number of points for freqz
Wc = 0.2    # 3dB point
Order = 3   # filter order

# design a Butterworth filter
[b, a] = signal.butter(Order, Wc)

# calculate the frequency repsonse
[w, h] = signal.freqz(b, a, N)

# plot the results
figure()

subplot(2, 1, 1)
plot(arange(N)/float(N), 20*log10(abs(h)), lw=2)
title('Frequency response')
xlabel('Frequency (normalized)')
ylabel('dB')
ylim(ylim()[0], ylim()[1]+5)
grid()

subplot(2, 1, 2)
plot(arange(N)/float(N), 20*log10(abs(h)), lw=2)
title('Frequency response (3dB point)')
xlabel('Frequency (normalized)')
ylabel('dB')
text(Wc+.02, -3, '3dB point', va='bottom')
ylim([-3, 0.1])
grid()
show()
```

I've used two functions: `butter()` and `freqz()`. The function `butter()` designs an IIR filter with specified parameters (order and cutoff frequency), and the function `freqz()` returns a frequency response. Note that the frequency response is a complex number, so I've plotted the amplitude in dB of the absolute value: `20*log10(abs(h))`, as shown in Figure 8-14.

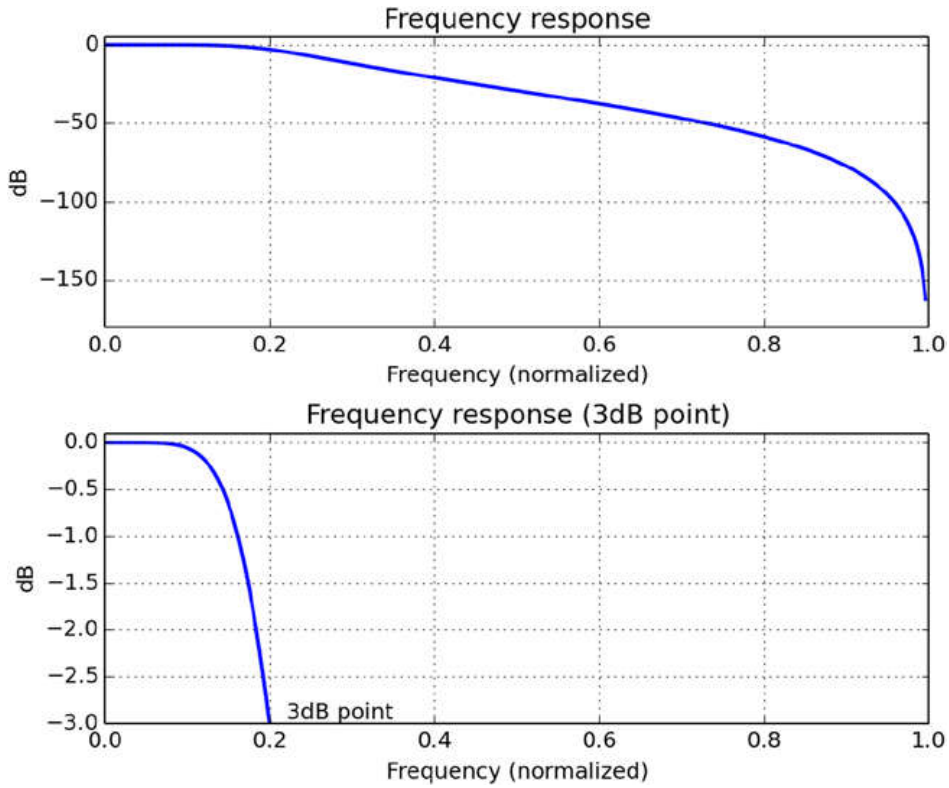

Figure 8-14. *The frequency response of a low-pass filter*

To filter data, given a known filter, use the function, `scipy.lfilter(b, a, x)`. Let's turn to an example.

Example: A Heart-Rate Monitor

For this example, I'll generate a signal that simulates the data generated from a heart-rate monitor connected to a patient. Please do not use this in any sort of production system; it's merely for educational purposes.

The patient walks around, and as a result, two signals are picked up: the heart signal and a signal associated with the patient's movement, or what is typically referred to as a movement artifact. Listing 8-9 shows these signals in my simulation.

Listing 8-9. A Heart Rate Simulation

```
# heart signal simulation
N = 256     # number of samples per second
T = 2       # number of seconds
hr = 1.67   # 100 beats per minutes
F1 = 0.5    # movement frequency

t = arange(T*N)/float(N)
y1 = 5*sin(2*pi*t*F1)     # movement artifact
```

```
# add heart signals
y2 = zeros(size(y1))
for i in range(int(T*hr)):
    y2[i*N/hr:i*N/hr+10] = signal.triang(10)

# combine movement with beats
y = y1+y2

# create a high-pass filter
[b, a] = signal.butter(3, 0.05, 'high')

# filter the signal
yn = signal.lfilter(b, a, y)

# plot the graphs
figure()

subplot(2, 1, 1)
title('Heart signal with movement artifact (simulation)')
plot(t, y, lw=2)
xlabel('t [seconds]')
ylabel('Amplitude []')

subplot(2, 1, 2)
title('Filtered signal')
plot(t, yn, lw=2)
xlabel('t [seconds]')
ylabel('Amplitude []')
show()
```

I've defined several parameters that control the script. The value `N` is equal to the number of samples per second (some are used to naming this value `Fs`, which stands for frequency of sampling). The value `T` is the total number of seconds; in this case, 2 whole seconds. The value `hr` is the patient's heart rate, 100 beats per minute: 100 / 60 = 1.67 Hz. Lastly, I defined the movement artifact frequency at 0.5 Hz. I then constructed a time vector, `t`, and a movement artifact vector, `v1`, and added "beats" with triangular waveforms using the `signal.triang()` function.

Now that I have a heart signal with a movement artifact, I turn to filter out the movement artifact. I design a second-order Butterworth HPF to do so via the call to `signal.butter()`, and then use the filter parameters to filter the signal with `signal.lfilter()`. Figure 8-15 shows the resulting plot.

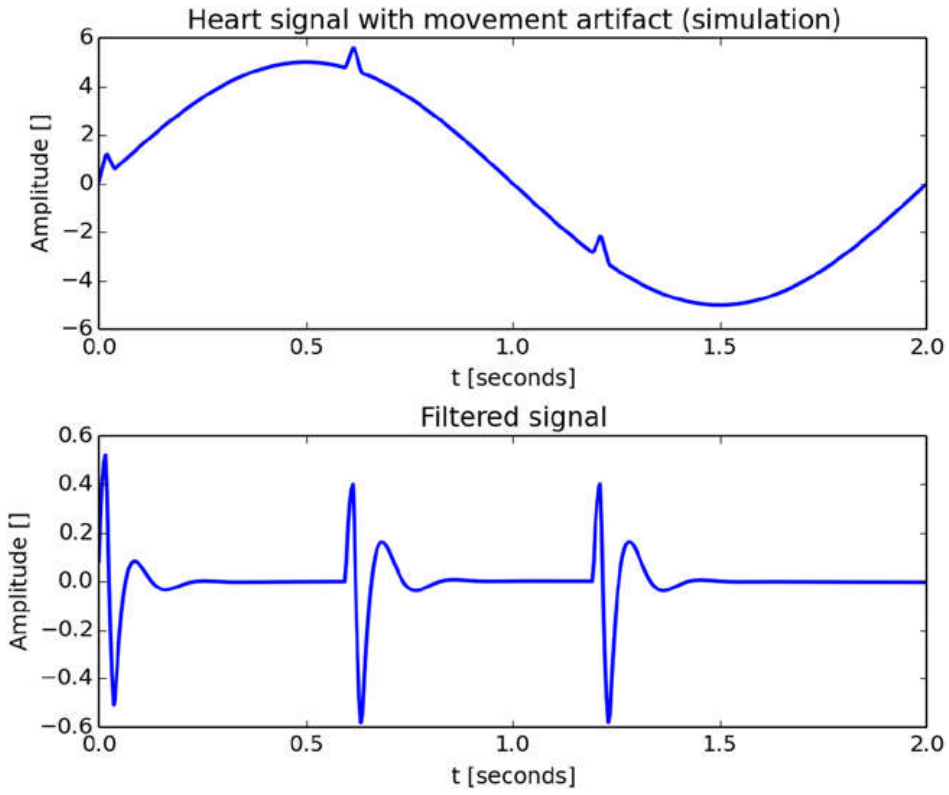

Figure 8-15. *Filtering a signal*

Example: Moving Average

On many occasions, filtering is used to "smooth" a signal. A simple algorithm is that of a moving average. For every two consecutive points, we calculate the average and use that value instead. The points are overlapping, so a result of using the algorithm on the vector `[1, 2, 0, 2]` would be `[1.5, 1, 1]`. But why stop at two samples? Moving average can be performed on several points, returning the average of those points. Listing 8-10 shows how this could be written with *SciPy* in Python.

Listing 8-10. A Moving Average

```
from pylab import *
N = 512
t = linspace(0, 10, N)
x = 1-exp(-t) +randn(N)/10
W = 32  # num points in moving average
xf = zeros(len(x)-W+1)
for i in range(len(x)-W+1):
    xf[i] = mean(x[i:i+W])

plot(t, x)
hold(True)
plot(t[W-1:], xf, lw=3)
title('Moving average')
```

```
legend(['signal with noise', 'filtered signal'])
xlabel('t [seconds]')
ylabel('x []')
show()
```

This is a straightforward implementation using a `for` loop. The input to the filter is arbitrarily chosen as $1\text{-}e^{-t}$, plus noise.

There is an easier approach. A moving average is a FIR filter with all its elements equal to 1/W; where W is the length of the moving average window. In this case, a quick-and-simple way to implement a moving average filter instead of the `for` loop is by calling the `signal.lfilter()` function and passing `ones(W)/W` as the filter values:

```
>>> from scipy import signal
>>> xf = signal.lfilter(ones(W)/W, 1, x)
```

Figure 8-16 shows the results of plotting a moving average.

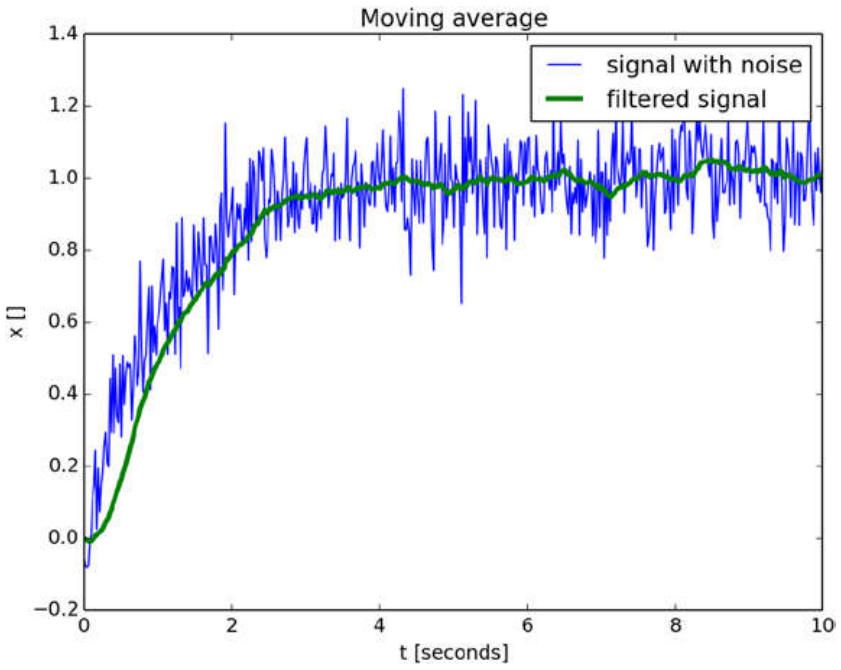

Figure 8-16. *A moving average*

Final Notes and References

The purpose of this chapter is to serve as a cookbook of algorithms in numerical analysis and signal processing. I took great care to limit the amount of math used in the examples and yet still be informative.

Even so, the topics covered in the chapter are far too many to be explored in one book, let alone one chapter. If you find these topics of interest, the following may provide additional information:

- *Numerical Recipes: The Art of Scientific Computing, Third Edition* by William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery (Cambridge University Press, 2007; for more information, see http://www.nr.com)
- *SciPy*, http://www.scipy.org/

CHAPTER 9

Image Processing

Two-Dimensional Data

Up to this point we've mostly dealt with one-dimensional data. That is, we've covered graphs and data that are essentially composed of a series of values. We've plotted the data, analyzed it, and created an image that was later saved to file or displayed to screen.

However, an image data file, or the image on your screen, is two dimensional. It is composed of pixels (which is short for picture elements). Each pixel has a location in two dimensions, x and y, and a value corresponding to the color. In this chapter, we will turn to manipulating images on the pixel level; that is, we'll operate on the two-dimensional matrix of pixels.

Operations on images are similar to operations performed on one-dimensional data. Slicing a 1-D array of values and adding those values to another array is equivalent to copying and pasting images. Saving an array to file is equivalent to writing images in TIFF or JPEG format. And so on.

Copying and pasting, resizing, and cropping are all simple operations supported by most GUI-based graphics applications. But with a GUI application, it's harder to perform these operations in a systematic and automated manner. For example, you might resize several images, and then combine them together to form a collage of images. It's doable in GUI applications, but it's typically not as easy and requires some user skills. Other simple image operations include converting file formats, rotating images, and cropping them. I'll cover these image operations, as well as how to automate them, so the results are consistent.

I'll show you how to deal with data on a numerical level, including reading the image, transforming it into a *NumPy* 2-D array, and then operating on the array itself. I'll also show you how to implement some interesting algorithms that involve image processing. Lastly, I'll touch on some more complex topics like filtering, which is the act of modifying a picture to enhance the visual output.

The basic package we'll use for all these operations is a fork of the Python Imaging Library (PIL), named *Pillow*. Be sure to install *Pillow* per the guidelines in Chapter 2. We'll also be relying on *NumPy* and *matplotlib*.

Make sure you read Chapters 6 and 7 before continuing with this chapter; the concepts taught in those chapters are required to follow along with this chapter.

Reading, Writing, and Displaying Images

The Python Imaging Library provides us with several classes that enable image processing. The basic class, `Image`, supports image operations such as reading an image from file, writing an image to file, copying and pasting, resizing, and rotating. To use *Pillow*, we issue the import statement: `import PIL` (notice that it's `PIL` and not `Pillow`).

Reading Images from File

Let's get started. To use the `Image` class, import it as follows:

```
>>> from PIL import Image
```

■ **Note** Going forward, I'll assume you've issued `from PIL import Image`, and I will refrain from mentioning the `import` statement.

Our first operation is reading an image file. Since we currently don't have any images, let's generate an image and read it from file. We'll use *matplotlib* patches for this, as shown in Listing 9-1. (If you're not familiar with *matplotlib* patches, refer to Chapter 6.)

Listing 9-1. Creating an Image

```
>>> from pylab import *
>>> figure()
>>> gca().add_patch(Circle((0, 0), 1))
>>> axis('off')
>>> axis('scaled')
>>> savefig('../images/circle.png')
```

I've used *matplotlib* to create an image file to work with, but you just as well could use any image file, for example, a JPEG picture you took with your digital camera. The code in Listing 9-1 draws a filled circle and saves it to file `../images/circle.png`.

■ **Note** In accordance with the directory structure presented in Chapter 2, I assume that you're currently running from directory `Ch9/src` and that directory `Ch9/images` holds image files. If that is not the case, be sure to change the path to the images in the examples provided in this chapter.

Our first operation is to read the file and attach it to an `Image` object:

```
>>> im = Image.open('../images/circle.png')
```

Image Attributes

Now that we have an `Image` object associated with an image file, we can query the object's attributes:

```
>>> im.size
```

```
(800, 600)
```

```
>>> im.info
```

```
{'dpi': (100, 100)}
```

```
>>> im.mode
```

```
'RGBA'
```

```
>>> im.format
```

```
'PNG'
```

```
>>> im.filename
```

```
'../images/circle.png'
```

This is quite a bit of information. We know that the image size is 800×600 pixels wide, the resolution is 100 dpi, the mode is RGBA (we'll get to modes a little later in this chapter's "Creating New Images" section), the image format is PNG, and the file associated with the object we've created is `../images/circle.png`.

Example: Image Catalog

My experience with analyzing image data is that images are not always taken in a consistent manner. This means that you, the programmer, have to manually crop, resize, enhance, or even delete images. This also translates into maintaining a catalog file of some sorts. An approach I find helpful is to create an automated catalog file and then annotate the information as I work with the data (see Chapter 4 for a discussion of catalog files).

The purpose of this example is to create a basic image catalog file (see Listing 9-2). The script uses the `Image` attributes presented in the previous section and creates a CSV catalog file in the parent directory of the searched directory. The catalog file has an extension of `.cat.csv`. That is, if you're searching `/home/user`, a catalog file will be created and named `/home/user.cat.csv`. The catalog file includes the name, size, format, and resolution of each image.

Listing 9-2. Creating an Image Catalog

```
from PIL import Image
import os, csv

def image_catalog(srchpath):
    """Creates a catalog file named srchpath.cat.csv."""

    # the CSV header
    catalog = [['Filename', 'Pathname', 'Format', 'Size', 'Resolution' ]]

    # walk directory tree
    for root, dirs, files in os.walk(srchpath):
        for file in files:
            pathname = os.path.join(root, file)
            try:
```

```
                img = Image.open(pathname)
                filesize = os.path.getsize(pathname)
                catalog.append([file, pathname, img.format,
                    img.size, img.info])
            except IOError:     # not an image
                pass

    # create the clean catalog
    f = open(srchpath.rstrip('/')+'.cat.csv', 'w', newline='')
    csv.writer(f).writerows(catalog)
    f.close()
```

The script defines a function named `image_catalog()`, which accepts the directory to search and produces an image catalog file in CSV format. The variable `catalog` is a list of rows containing image information. We iterate through the directory and look for images with the Easier to Ask Forgiveness than Permission (EAFP) approach: try to open a file as if it were an image file. If we succeed, the catalog is updated. If the file is not an image, the exception `IOError` is raised, and we `pass` this file.

■ **Note** If your directory is supposed to contain strictly images, you might want to add a `print` statement before the `pass`, notifying the user that a non-image file was encountered.

Here are the results I got from running the script in the directory `images` (the contents of the file, `images.cat.csv`):

```
Filename,Pathname,Format,Size,Resolution
nightsky1.png,../images/nightsky1.png,PNG,"(800, 600)","{'dpi': (100, 100)}"
nightsky2.png,../images/nightsky2.png,PNG,"(800, 600)","{'dpi': (100, 100)}"
circle.png,../images/circle.png,PNG,"(800, 600)","{'dpi': (100, 100)}"
nightsky.png,../images/nightsky.png,PNG,"(800, 600)","{'dpi': (100, 100)}"
collage.png,../images/collage.png,PNG,"(600, 600)",{}
```

Displaying Images

You can view an image by calling the `Image` method `show()`, or `Image.show()`. The method in turn calls the operating system's default image viewer, which is usually provided by the OS. To use a different viewer from the one supplied by the OS, associate the image with an image viewer you desire. The following will display the image `../images/circle.png`, which was created previously:

```
>>> Image.open('../images/circle.png').show()
```

Converting File Formats

One of the common operations to perform on images is to convert the image file format. Perhaps you want to store images in a more efficient format using compression, or maybe the application you're using requires the image in a different format than you received it in. No matter what your reason, the `Image` method `save()` enables saving an image to file in a specified image format. There are two approaches for specifying a format: by using a file name extension or by explicitly specifying the `format` argument.

Assuming you've created an image file `../images/circle.png` per the previous listing, you can read the image and convert the file format to a JPEG file format, as follows:

```
>>> im = Image.open('../images/circle.png')
>>> im.save('../images/circle.jpg')
>>> import os
>>> os.listdir('../images')
```

```
['circle.jpg', 'circle.png']
```

In this particular example, you're not really converting the file, but creating another file with a different image format (converting it would mean that you also delete the original file).

Or, you could create a function to convert an image to JPEG format, as shown in Listing 9-3.

Listing 9-3. A Function to Convert an Image to JPEG Format

```
from PIL import Image
from os.path import splitext

def ConvertToJpeg(filename):
    """Convert an image file to a Jpeg file."""

    jpegname = splitext(filename)[0]+'.jpg'
    Image.open(filename).save(jpegname)
```

In the preceding example, I've used the `splitext()` function, which is part of the *os* module to replace the original extension with a `.jpg` extension. The `.jpg` extension instructs the `save()` function to create a JPEG file.

As mentioned previously, you can also explicitly specify a format:

```
>>> im = Image.open('../images/circle.png')
>>> im.save('../images/circle', format='Jpeg')
>>> import os
>>> [fn for fn in os.listdir('../images') if fn.startswith('circle')]
```

```
['circle.png', 'circle', 'circle.jpg']
```

In this case, `save()` does not add an extension to the file name (that is, the file created is `circle`, not `circle.jpg`) because that is the filename passed to it: `circle`.

PIL supports a large number of file formats, and the `Image` class can read most popular image formats. Furthermore, most images can be saved using known file formats, including JPEG, TIFF, and PNG. However, some image formats can only be read. Other formats such as MPEG (video files) are supported in identify mode only. For a full account, refer to the *Pillow* reference: http://pillow.readthedocs.org/en/latest/reference/index.html.

Example: A Function to Convert All Images in a Directory to JPEG Format

A direct continuation of the idea of converting an image file format is to write a function that iterates through a directory and converts all images to JPEG format, as shown in Listing 9-4. We'll also keep the original image because JPEG uses a lossy compression algorithm, which might lower the original image quality. However, you can easily modify the example to remove the original images.

Listing 9-4. Converting All Images in a Directory to JPEG

```
import os, csv
from PIL import Image

def ConvertDirToJpeg(srchdir):
    """Converts all images in a directory to a jpeg file."""

    # walk directory tree
    for root, dirs, files in os.walk(srchdir):
        for file in files:
            # pathname holds the image filename
            pathname = os.path.join(root, file)
            try:
                # convert the file to a Jpeg file
                img = Image.open(pathname)
                jpegname = os.path.splitext(pathname)[0]+'.jpg'
                if os.path.exists(jpegname):
                    print("Did not create %s; file already exists." % jpegname)
                else:
                    img.save(jpegname)
                    print("Created file ", jpegname)
            except IOError:     # oops, not an image
                pass
```

Again, the preceding script uses the EAFP approach: try to open a file as an image, and if all goes well, convert it to a JPEG image. To run the function, enter `ConvertDirToJpeg(dirname)`.

■ **Note** If the function `ConvertDirToJpeg()` is called with a non-existent directory, no output is generated, not even a warning message. If you require such functionality, be sure to modify the function and include it.

Image Manipulation

So now we can read images, display them, and convert file formats. But in converting file formats (and provided we do not use a lossy compression algorithm), we haven't really changed the images; we've merely saved them in a different format.

In this section, we'll turn to performing basic image manipulations. In other words, we'll modify the contents of an image by cutting and pasting, cropping, and rotating it.

Creating New Images

Pillow provides us with the ability to create images, not just read them from files. This is especially useful when you want to copy and paste images from other sources to a new image. The syntax for creating a new image is `Image.new(mode, size, color=0)`. The `mode` argument can take one of the values listed in Table 9-1.

Table 9-1. *Image Modes*

Mode	Description
`'1'`	1 bit per pixel; useful for black-and-white images.
`'L'`	1 byte per pixel (values from 0 to 255), black and white; useful for working with a one color band (see a discussion about color later in the chapter).
`'RGB'`	Red, green, and blue, 1 byte per color, also known as true color. RGB is common when the image background is black, such as on a screen monitor.
`'RGBA'`	Red, green, blue, and a transparency mask, 1 byte per color; common in several file formats, including PNG.
`'CMYK'`	Cyan, magenta, yellow, and black, 1 byte per color. CMYK is common in print.

There are additional image modes, but I won't be covering them in this chapter. To view the list of available modes, issue the following:

```
>>> from PIL import Image
>>> Image.MODES
```

```
['1', 'CMYK', 'F', 'I', 'L', 'LAB', 'P', 'RGB', 'RGBA', 'RGBX', 'YCbCr']
```

Refer to the *Pillow* web site for additional information: http://pillow.readthedocs.org/en/latest/handbook/concepts.html.

The `size` argument in the `Image.new()` function is a two-element tuple describing the width and height of the image. The `color` argument is a function of the mode. For example, in the case of an RGB image, the color is a tuple in the form (red, green, blue); in the case of CMYK, the color takes the form (cyan, magenta, yellow, black):

```
>>> im1 = Image.new('L', (800, 600)) # black, one-band image
>>> im2 = Image.new('CMYK', (800, 600), (0, 255, 0, 0)) # magenta image
>>> im3 = Image.new('RGB', (800, 600), (255, 0, 0)) # red image
```

You can view the images by calling the `show()` method e.g., `im1.show()`.

Copy and Paste

The methods `copy()` and `paste()` let us copy images and paste images into other images, respectively. The method `copy()` requires no parameters and creates a copy of the current image. The method `paste(im, xy)` pastes the image `im` into the current image; the `xy` argument is a tuple indicating the (x, y) location to paste (top left). Let's turn to an example that uses the `paste()` method.

Example: Fractal Collage

In this example, we'll use the functions `new()`, `open()`, `paste()`, and `save()` to create a collage of images. To follow along, you'll need to modify the fractal script presented in Chapter 7, so that it's a function instead of a script (see Listing 9-5). Once you create the function, save it under `Ch9/src/fractal.py`.

Listing 9-5. A Function to Create Fractals

```
from PIL import Image
from cmath import *

def fractal(delta=0.000001, res=800, iters=30):
    """Creates a z**4+1=0 fractal using the Newton-Raphson method."""

    # create an image to draw on, paint it black
    img = Image.new("RGB", (res, res), (0, 0, 0))

    # these are the solutions to the equation z**4+1=0 (Euler's formula)
    solutions = [cos((2*n+1)*pi/4)+1j*sin((2*n+1)*pi/4) for n in range(4)]
    colors = [(1, 0, 0), (0, 1, 0), (0, 0, 1), (1, 1, 0)]

    for re in range(0, res):
        for im in range(0, res):
            z = (re+1j*im)/res
            for i in range(iters):
                try:
                    z -= (z**4+1)/(4*z**3)
                except ZeroDivisionError:
                    # possibly divide by zero exception
                    continue
                if(abs(z**4+1) < delta):
                    break

            # color depth is a function of the number of iterations
            color_depth = int((iters-i)*255.0/iters)

            # find to which solution this guess converged to
            err = [abs(z-root) for root in solutions]
            distances = zip(err, range(len(colors)))

            # select the color associated with the solution
            color = [i*color_depth for i in colors[min(distances)[1]]]
            img.putpixel((re, im), tuple(color))

    return img
```

Armed with the function `fractal(delta, res, iter)`, we create a fractal collage, as shown in Listing 9-6.

Listing 9-6. A Collage of Fractals

```
from PIL import Image, ImageOps

# import the fractal function, from fractal.py file
from fractal import fractal

fsize = 200     # small fractal image width and height
nx = 3          # number of images, width
ny = 3          # number of images, height
```

```
collage = Image.new("RGB", (fsize*nx, fsize*ny))
for i in range(ny):
    for j in range(nx):
        im = fractal(0.000001, fsize, i*nx+j+1)
        print("Processing image %d of %d" % (i*nx+j+1, nx*ny))
        collage.paste(im, (fsize*j, fsize*i))

collage.save('../images/collage.png')
```

The script generates fractals with increasing numbers of iterations and pastes them into an image that serves as the image collage. The arguments to the `paste()` method are chosen so that the images are pasted from the top left to the bottom right. I've saved the image to file `../images/collage.png`.

The result from running this script is shown in Figure 7-3 in Chapter 7.

Crop and Resize

Cropping and resizing modify an existing image. The function `crop()` selects part of the original image, and the function `resize()` resizes an existing image; that is, it scales it so it fits the new size.

Assuming you have run the previous collage example, you should now have a file named `collage.png`. The function `crop()` accepts a tuple of four values, detailing the box to crop: (x0, y0, x1, y1). Let's read the `collage.png` file and crop it to show only 2 by 2 images from the fractal collage:

```
>>> img = Image.open('../images/collage.png')
>>> img.size
```

```
(600, 600)
```

```
>>> cropped_img = img.crop((0, 0, 400, 400))
>>> cropped_img.size
```

```
(400, 400)
```

```
>>> cropped_img.show()
```

Suppose you want to show the entire image, but scaled to size (400, 400); in this case, you'd use the `resize(xy)` function, where `xy` is a two element tuple detailing the width and height of the resized image:

```
>>> img = Image.open('../images/collage.png')
>>> img.size
```

```
(600, 600)
```

```
>>> resized_img = img.resize((400, 400))
>>> resized_img.show()
>>> resized_img.size
```

```
(400, 400)
```

You can also use the method `thumbnail()`, which is similar to `resize()`. The difference is that `resize()` returns a modified image copy, whereas `thumbnail()` modifies the image itself.

```
>>> img.thumbnail((400, 400))
>>> img.size
```

```
(400, 400)
```

In both the `resize()` and `thumbnail()` methods, you can provide a `filter` argument that determines the method of resampling. The acceptable values are `Image.NEAREST` (default), `Image.BILINEAR`, `Image.BICUBIC`, and `Image.ANTIALIAS` (best quality); see http://pillow.readthedocs.org/en/latest/handbook/concepts.html for details. Antialiasing has the best results, but it might take longer to compute:

```
>>> img.thumbnail((400, 400), Image.ANTIALIAS)
```

Rotate

Last on our list of basic operations is the `rotate()` function. The function `rotate(theta)` rotates an image `theta` degrees.

From a user's perspective, rotating is a basic operation, such as when rotating a scanned document by a few degrees so it's properly displayed. But in reality, rotation isn't such a basic operation; it requires changing the width and height of the image. In the case of rotating by multiples of 90 degrees, the `rotate()` function knows to swap the x-axis and the y-axis; however, in the case of other rotation values, both axes change, so the total area of the image changes. You can control whether you want `rotate()` to expand the image so it includes the entire rotated image or not by passing `expand=True` or `expand=False`, respectively:

```
>>> img = Image.new('RGB', (200, 300), (0, 255, 255))
>>> img30 = img.rotate(30)
>>> img30.show()
>>> img30.size
```

```
(200, 300)
```

```
>>> img30e = img.rotate(30, expand=True)
>>> img30e.show()
>>> img30e.size
```

```
(324, 360)
```

In the first line, I've created a simple blue image that is 200 pixels wide and 300 pixels high. I've then rotated the image 30 degrees with and without expanding. The results are shown in Figure 9-1.

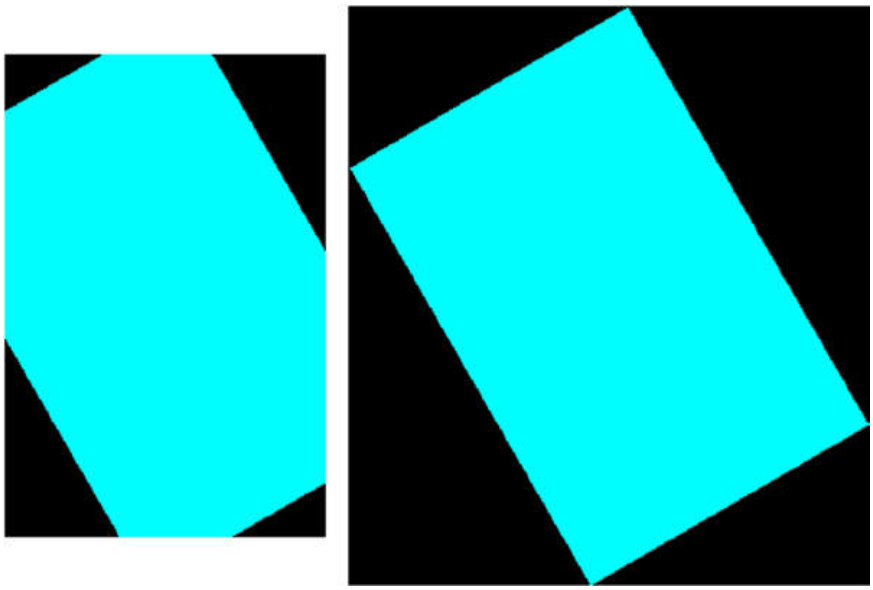

Figure 9-1. *Rotated images: the left is not expanded, while the right is expanded*

Image Annotation

Annotating images is just as important as annotating graphs. It gives extra, important information. When was this picture taken? What camera was used? And so on. However, in some cases, annotating an image with text disrupts the pleasing visual result. That's probably why it's less common in pictures. There's also the issue of what color to choose. In cases where the picture is mostly white, you probably want to choose nonwhite annotation.

In this section we'll cover text annotation, as well as geometrical shapes to highlight specific image features.

Annotating with Geometrical Shapes

Pillow provides us with the `ImageDraw` object, which allows us to annotate an existing image. To import the `ImageDraw` object, issue `from PIL import ImageDraw`. To use the `ImageDraw` object, attach it to an existing image:

```
>>> from PIL import Image, ImageDraw
>>> img = Image.new('RGB', (200, 300), (0, 0, 255))
>>> draw = ImageDraw.Draw(img)
```

I've created an `ImageDraw` object named `draw` and attached it to the image, `img`. Going forward, operations performed with the `ImageDraw` object will be performed on the `Image` object:

```
>>> draw.line((100, 100, 200, 200))
>>> img.show()
```

This will draw a line from (100, 100) to (200, 200).

You can use the functions in Table 9-2 to annotate an image. In the table, assume `draw = ImageDraw.Draw(Image)`.

Table 9-2. *Some ImageDraw Functions*

Function	Description	Example
`arc(xybox, start, end)`	Draws an arc, a part of the circle bound by the rectangle, `xybox` (a tuple of four elements), starting at angle `start` and ending at angle `end`.	`draw.arc((100, 100, 200, 200), 90, 180)` will draw a quarter of a circle.
`chord(xybox, start, end)`	Similar to `arc()`, only it draws a line connecting the arc edges.	`draw.chord((0, 0, 100, 100), 90, 180)`
`ellipse(xybox)`	Draws an ellipse bound by the four-element tuple `xybox`. If you'd like a circle, use a square for the `xybox` values.	`draw.ellipse((50, 50, 150, 100))`
`line(xyseq)`	Draws lines connecting elements in the sequence, `xyseq`.	`draw.line([0, 0, 10, 10, 20, 10, 20, 20])`
`point(xy)`	Draws a point at location, `xy`.	`draw.point((40, 40))`
`polygon(xyseq)`	Draws a polygon connecting elements in the sequence, `xyseq`. The difference from the `line()` function is that the polygon is always a closed shape, allowing the use of the `fill` argument.	`draw.polygon([10, 20, 40, 40, 50, 30, 70, 80])`
`rectangle(xybox)`	Draws a rectangle specified by the four-element tuple, `xybox`.	`draw.rectangle((20, 60, 80, 140), fill=128)`

The `ImageDraw` annotation functions accept the following optional arguments: `fill`, which determines the color of the annotation or the fill object (similar to the `facecolor` argument in *matplotlib*); `outline`, which determines the line to draw the object (similar to the *matplotlib* `edgecolor` argument); and `font`, in the case of text annotations.

Text Annotations

Other than geometrical shapes, `ImageDraw` also provides text annotation with the function, `text((x,y), string)`:

```
>>> from PIL import Image, ImageDraw
>>> img = Image.new('L', (160, 160), 255)
>>> draw = ImageDraw.Draw(img)
>>> draw.ellipse((0, 0, 160, 160), fill=128)
>>> draw.text((80, 80), 'A long string')
>>> img.show()
```

Originally, I had intended to have the text centered horizontally. However, the text string has a width, so I require a method to calculate the width and height of the text in pixels. Once I have the width and height, I can draw the text at location (80 – width/2, 80 – height/2). This is done using the function, `textsize()`:

```
>>> from PIL import Image, ImageDraw
>>> img = Image.new('L', (160, 160), 255)
>>> draw = ImageDraw.Draw(img)
>>> draw.ellipse((0, 0, 160, 160), fill=128)
>>> s = 'A long string'
>>> width, height = draw.textsize(s)
```

```
>>> width, height
```

```
(78, 11)
```

```
>>> draw.text((80-width/2, 80-height/2), s)
>>> img.show()
```

Figure 9-2 shows the results with and without taking into consideration the string width and height.

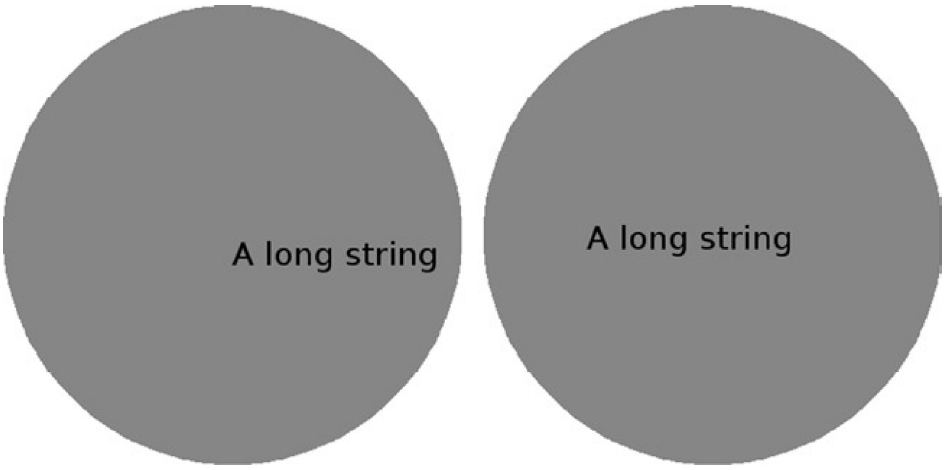

Figure 9-2. *Text annotation using* `text()` *and* `textsize()`

Fonts

It's also possible to use other fonts with the `text()` function. To do so, first create an `ImageFont` object. The `ImageFont` object is part of *Pillow*; to import it, issue `from PIL import ImageFont`. Once `ImageFont` is imported, you can use a font with a call to `ImageFont.truetype(fontname, size)`. The returned `ImageFont` object can be passed as an argument to the `text()` function by means of the `font` argument.

Of course, before you can use fonts, they must first be installed in your system. Windows and MacOS typically come with built-in fonts; and on Linux, fonts are usually installed with X (as well as other applications). You can also use fonts from the GNU FreeFont project (http://www.gnu.org/software/freefont/).

However, font and font names are different on varying systems, and not just different operating systems: my Windows system might have different fonts than your Windows system. This means calling `ImageFont.truetype(fontname, size)` might work on one system and not on another. To overcome this problem, I use the function `findfont()`, which is part of the *matplotlib.font_manager* module. The function `findfont()` returns a string with the location of a font that best matches the requested font.

The following script annotates text with the Vera font using the `findfont()` function:

```
>>> from matplotlib import font_manager
>>> from PIL import Image, ImageDraw, ImageFont
>>> img = Image.new('L', (250, 100), 255)
>>> draw = ImageDraw.Draw(img)
>>> font_str = font_manager.findfont('Vera')
>>> font_str
```

```
'C:\\Python33\\lib\\site-packages\\matplotlib\\mpl-data\\fonts\\ttf\\Vera.ttf'
```

```
>>> ttf = ImageFont.truetype(font_str, 54)
>>> s = 'ABCabc'
>>> (w, h) = draw.textsize(s, font=ttf)
>>> draw.text(((250-w)/2, (100-h)/2), s, font=ttf)
>>> img.show()
```

The first two statements import the proper objects from *Pillow*, as well as *matplotlib*'s font manager. I then create a one-band image of size (250, 100), followed by the instantiation of an `ImageDraw` object attached to the image.

Next, I use *matplotlib*'s `findfont()` function to find a font that's closest to the font named Vera. The path to the font is stored in the string, `font_str`. Following that, I create an `ImageFont` object named `ttf` and use that font object to render the text. I then calculate the size of the text and render it in the middle of a white background, as shown in Figure 9-3.

Figure 9-3. *Font rendering*

■ **Note** To use a font, you must supply the `font` argument in calls to two functions, `text()` and `textsize()`.

Example: Thumbnail Index Image

In a previous example (Listing 9-2), we created a catalog of images. While that catalog is quite useful, it doesn't show the contents of those images. Perhaps a more useful catalog would be a collage of the images annotated with text showing each image's file name (see Listing 9-7).

Listing 9-7. A Thumbnail Index Image

```
# thumbnail index
import os
from PIL import Image, ImageDraw

def thumbnail_index(dirpath):
    """Create a thumbnail index from images in dirpath."""

    num_images = 5
    thumb_size = (128, 96)
    cat_size = (num_images*thumb_size[0], num_images*thumb_size[1])

    fn_index = 0            # filename index
    img_index = 0           # image index

    # go through all the pictures in a directory
    for file in os.listdir(dirpath):
        # get the pathname for the file
        pathname = os.path.join(dirpath, file)

        try:    # is this an image file?
```

```
            # open the image file
            img = Image.open(pathname)

        except IOError:
            print(file, "is not an image file")
            continue

        # create a thumbnail
        img.thumbnail((thumb_size), Image.ANTIALIAS)
        draw = ImageDraw.Draw(img)
        draw.text((2, 2), file)

        # do we need to create a new catalog image?
        if img_index == 0:
            thumbs_img = Image.new('RGB', cat_size)

        # calculate the location for this image
        x = img_index % num_images
        y = img_index // num_images

        # paste the thumbnail
        thumbs_img.paste(img, (x*thumb_size[0], y*thumb_size[1]))

        # increment the image index
        img_index += 1

        # have we reached the end of the catalog image?
        if img_index==num_images**2:
            img_index = 0
            thumbs_img.save('%s-%03d.cat.jpg' % (dirpath, fn_index))
            fn_index += 1

    # save the last catalog file
    if img_index:
        thumbs_img.save('%s-%03d.cat.jpg' % (dirpath, fn_index))
```

The function `thumbnail_index()` accepts a directory and produces a thumbnail index image. Figure 9-4 shows the result from running the function on a collection of images my daughter particularly likes.

Figure 9-4. *A thumbnail index image*

For the purpose of this example, I decided not to use `os.walk()` to iterate through the directory listing. Instead, I used `os.listdir()`. I defined two parameters: `num_images`, which holds the number of images on either x- or y-axis; and `thumb_size`, which holds the thumbnail width and height. Next, I composed a list of all the files in the requested directory. For every file, the script tries to open the file as if it were an image file. If a file is indeed an image file, a thumbnail of the image is created and pasted to the index image. Additionally, the thumbnail is annotated with the file name in the top-left corner. There's some indexing used to determine the exact location of an image in the thumbnail index image, as well as to create a new thumbnail index image once the current one has filled up.

■ **Tip** An alternative approach to displaying the text directly on the thumbnail image is to display it below the image. You can do this by adding a stripe of black (or white) between rows of images.

Image Processing

So far, we've performed tasks that can also be performed by most GUI-based image editing applications, including GIMP, the GNU Image Manipulation Program (http://www.gimp.org/). However, GUI-based applications have a GUI user in mind and are not easily automated. We now turn to explore possibilities of writing scripts to automatically perform operations on images.

Furthermore, as you start thinking about higher-level image processing algorithms, you may realize you require access to the actual data, the numbers that represent the image. In this section, we'll also show how this can be achieved.

A word of caution: image processing is a vast field. I won't be covering even the basics here; instead, I'll show that, if you do have an image processing algorithm, it's quite likely you can implement it in Python.

Matrix Representation and Colors

An image can be represented by a matrix, with each (x, y) point corresponding to a column and row in the matrix, and the value corresponding to the color.

The color value is a function of the mode (see Table 9-1 earlier for details). For example, in the case of an RGB image, each value of the matrix is a tuple of 3 bytes, with each byte representing a different color. In a sense, you can think of the entire image as three matrices, with each matrix corresponding to the colors red, green, and blue, which are also known as color bands, or channels.

Furthermore, each image can be split into these colors (depending on the mode, of course—there's no splitting of a 1-bit image into individual colors). You can do this with the `Image` method, `split()`:

```
>>> from PIL import Image
>>> im = Image.open('../images/circle.png')
>>> im.mode
```

```
'RGBA'
```

```
>>> R, G, B, A = im.split()
>>> R.mode, G.mode, B.mode, A.mode
```

```
('L', 'L', 'L', 'L')
```

■ **Note** I've assumed you have followed along with the chapter and created a file named `../images/circle.png`; if not, follow Listing 9-1 to create a `circle.png` image.

Each split image is an image by itself, but it now contains only one-color information—hence, its mode is `'L'`, not `'RGBA'`.

To retrieve the *data* associated with the color—that is, the actual values—call the function, `getdata()`. We can then transform the values to a *NumPy* array for some interesting numerical processing. The following continues on the previous listing:

```
>>> from pylab import *
>>> data = array(R.getdata())
>>> type(data)
```

```
<class 'numpy.ndarray'>
```

```
>>> data.size
```

```
480000
```

```
>>> data.shape
```

```
(480000,)
```

The image data is stored as a list of all the values in the image, not a matrix representation of the image. To change it to a matrix, we use the *NumPy* method, `reshape()`:

```
>>> im.size
```

```
(800, 600)
```

```
>>> data = data.reshape(im.size)
>>> data.size   # size should be the same
```

```
480000
```

```
>>> data.shape   # reshaped as a matrix
```

```
(800, 600)
```

Now that we have the data as a matrix, we can operate on the matrix instead of the image. This gives us great flexibility. Say we want to arbitrarily draw a magenta stripe in the middle of the circle; all we need to do is modify the matrix associated with the red channel, and then merge it to form a new, modified image.

■ **Tip** Why does changing the red channel generate a magenta output? The way I'm going to modify the matrix is by setting the red channel to `255`. This means that my previously blue circle will now be a combination of blue and red, which is magenta, while the rest of the background, which is white, will remain white.

Let's do this a step at a time, from the top:

```
>>> from pylab import *
>>> from PIL import Image
>>> im = Image.open('../images/circle.png')
>>> im.mode
```

```
'RGBA'
```

```
>>> im.size
```

```
(800, 600)
```

```
>>> R, G, B, A = im.split()
>>> data = array(R.getdata()).reshape(im.size)
>>> (w, h) = data.shape
>>> data[w/2-100:w/2+100, :] = 255*ones((200, h))
>>> R.putdata(data.reshape(h*w))
>>> new_img = Image.merge('RGB', (R, G, B))
>>> new_img.show()
```

The first line reads the image from file and displays some image information. I then split the image into four channels: red, green, blue, and the transparency mask. From here on, I'll restrict myself to dealing with the red channel only. First, I retrieve the actual numerical values associated with the red channel. This is done with a call to `getdata()`. In the same line, I also transform the data into a *NumPy* array and reshape the list to a matrix form.

Next I change the data values associated with 200 rows in the middle and set their value to `255`. This effectively creates the magenta stripe. I then update the red channel with the modified data by calling the function, `putdata()`. The function `putdata()` complements `getdata()` and expects a list, not a matrix, so I reshape the data back to a 1-D array.

Finally, I create a new image, this time in RGB mode (I don't require transparency) by combining the original green and blue channels with the modified red channel. This is done by calling the `merge()` function, which is the opposite of the `split()` function. Figure 9-5 shows the results.

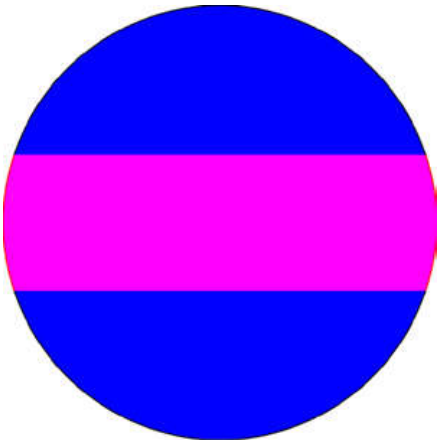

Figure 9-5. *A circle with a stripe*

■ **Note** It's also possible to perform this task using the `ImageDraw` object.

So far we've covered an interesting number of functions that enable working on images with numerical values: `split()`, `merge()`, `getdata()`, and `putdata()`. Next, we'll use some of them in a more complex example.

Reading an Image to a *NumPy* Array

An alternative method to using *Pillow*'s `Image` object and converting it to a *NumPy* array is to directly read the image as a *NumPy* array. You can do this by using *NumPy*'s `imread()` function. To save a *NumPy* array as an image, use `imsave()`. Finally, to view the image represented by the *NumPy* array, use *matplotlib*'s `imshow()`. Notice that *matplotlib* will display a 2-D image, along with the image's axis and its title; if you don't want the axis to show, you can remove it by issuing `axis('off')`.

The following shows how to read the image `../images/circle.png` using *NumPy* and display it using *matplotlib*:

```
>>> from pylab import *
>>> data = imread('../images/circle.png')
>>> data.shape
```

```
(600, 800, 4)
```

```
>>> data.size
```

```
1920000
```

```
>>> data.max()
```

```
1.0
```

```
>>> data.min()
```

```
0.0
```

```
>>> imshow(data)
>>> axis('off')
```

```
(-0.5, 799.5, 599.5, -0.5)
```

Notice that the shape of the variable `data`, as shown in `data.shape`, is `(600, 800, 4)`; this is to show that the *NumPy* array is a 3-D array, pixels are 800×600, and there are 4 channels: R ,G, B and A.

The next thing to notice is that `imread()` scales the pixel values from 0 to 1. This is shown in the results printed from `data.max()` and `data.min()`.

Now, let's modify the image by operating on the `data` array. In the following example, I invert the values of the channels R, G and B, and then write the result to file: `../images/inverted_circle.png`:

```
>>> data[..., :3] = 1 - data[..., :3]
>>> imshow(data)
>>> imsave('../images/inverted_circle.png', data)
```

If you're working on an algorithm, and you want *NumPy* and *SciPy* functionality (i.e., to operate on *NumPy* arrays), then I suggest you read the image directly to a *NumPy* array with the function shown in this section. If you require actual image operations (e.g., image filtering, image rotation, and so on), I would advise you to use *Pillow* instead.

Example: Counting Objects (Five Parts)

The following example is rather long and deals with an interesting aspect of image processing: counting objects in an image. The idea is to write a script that counts the number of elements in a picture. Counting elements is a complex task, even for the human mind: What objects should I count? What constitutes an object? And so on.

The task of counting objects is very useful in a wide variety of applications, as indicated by just a few examples:

- *Biology*: Estimating the number of bacteria in an image from a microscope
- *Medicine*: Counting the number of axons in a tissue cross-section
- *Electronics board manufacturing*: Counting the number of imperfections in a printed circuit board or counting the number of resistors
- *Astronomy*: Counting the number of stars

For the purpose of this example, we'll create an image of the sky at night, with stars placed randomly. We'll then write a script to count the number of stars. We'll have a very sterile image, one that has a very clean background (black, night sky) and most information in one channel. However, we'll add a bit of complexity by varying shapes and sizes of stars.

Once we have an image of the sky at night, I'll talk a bit about recursion, a topic I have been avoiding thus far. Recursion will be used to implement an algorithm to fill an image. Lastly, I'll discuss some ideas and methods you could use to expand upon this example and add more capabilities to the algorithm.

Part 1: Twinkle, Twinkle, Little Star

First, we create the stars for our image of the sky at night. The night sky will be composed of white stars and a black background. Since we want the stars to be of varying sizes and shapes, we'll create a function named `star()` that creates a *matplotlib* patch object (see Listing 9-8).

Listing 9-8. A Star Patch, the Source of `star_patch.py`

```
# create a star patch object
from pylab import *

def star(R, x0, y0, color='w', N=5, thin=0.5):
    """Returns an N-pointed star of size R at (x0, y0) (matplotlib patch)."""

    polystar = zeros((2*N, 2))
    for i in range(2*N):
        angle = i * pi/N
```

```
        r = R*(1-thin * (i%2))
        polystar[i] = [r*cos(angle)+x0, r*sin(angle)+y0]
    return Polygon(polystar, fc=color, ec=color)
```

The values that control the star patch are `R`, which determines the star's radius; `x0` and `y0`, which control the star's location; `color`, which determines both the fill and edge color; `N`, which controls the number of pointy edges a star has; and `thin`, which controls how thin or thick a star is (on the range of 0 to 1, with 1 being very thin).

■ **Tip** The default star patch is white because we'll be using it for the night sky. Be sure to change it to a different color if you're using a white background.

I've used the `Polygon` object to create the star patch, with some mathematical trickery. The idea is this: I place N pointy edges on a circle of radius R with the center at (x0, y0) at fixed angle increments. I then place another set of points at a smaller radius to serve as the inner edges of the star, again at fixed angle increments, but shifted so that each inner point resides exactly in the middle of the outer edge's points. The `thin` parameter determines the radius of the inner circle: the larger the value, the smaller the radius, and the "thinner" the star is. Lastly, I draw a line connecting all these points using the `Polygon` patch object.

■ **Note** Be sure to save the star patch listing as file `star_patch.py`; we'll use it in future scripts.

USING A LIST COMPREHENSION

It's also possible to implement the star patch with list comprehensions. The idea is to zip together the elements in the polygon list:

```
def another_star(R, x0, y0, color='w', N=5, thin = 0.5):
    """Returns an N-pointed star of size R at (x0, y0)."""

    a = arange(0, 2*pi, 2*pi/N)
    r = (1-thin)*R
    polystar = array(list(zip(R*cos(a)+x0, R*sin(a)+y0, \
        r*cos(a+pi/N)+x0, r*sin(a+pi/N)+y0)))
    return Polygon(polystar.reshape(N*2, 2), fc=color, ec=color)
```

Some prefer this implementation over the for-loop implementation. Personally, I think both are fine; choose whichever is easier for you to follow.

It's also possible to code the entire function as a single `return` statement, but I strongly recommend against it, as the code would be hard to understand.

The script in Listing 9-9 generates some interesting stars.

Listing 9-9. Generating Some Interesting Stars

```
# show some star examples
from pylab import *
from star_patch import star

examples=[
    "star(10, 0, 0, 'k')",
    "star(10, 0, 0, 'k', 10)",
    "star(10, 0, 0, 'k', 5, 0.2)",
    "star(10, 0, 0, 'k', 3, 0.9)" ]

for i, example in enumerate(examples):
    subplot(2, 2, i+1)
    exec("new_star="+example)
    gca().add_patch(new_star)
    title(example)
    axis('scaled')
    axis([-10, 10, -10, 10])

show()
```

In this script, I've decided to iterate over a list of strings and use the `exec()` function. The same string used for the `exec()` function is also used to create the title for the subplots (see Figure 9-6).

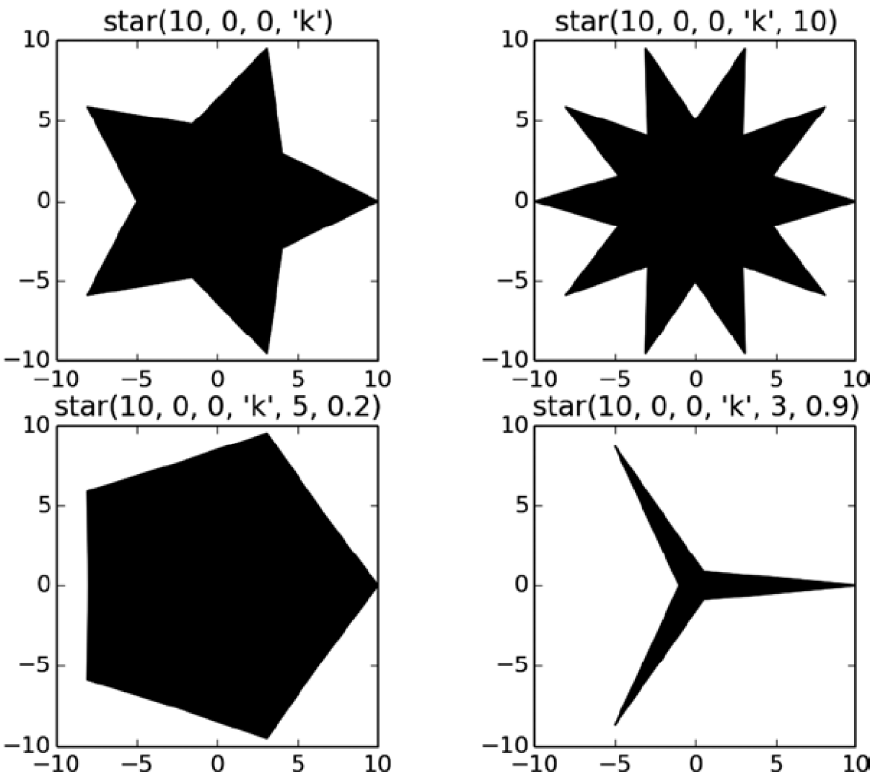

Figure 9-6. *Some star patches*

There's room for additional work on the `star()` patch object; for example, you could add a `rotation` parameter, rotating the entire star by `rotation` degrees. This can be done by changing the argument to the functions `sin()` and `cos()` in the `star()` function. Another modification could include a hollow star, implemented by splitting the `edgecolor` and `facecolor` functionality.

Armed with the star patch, let's turn to the second part of this example: creating an image of the sky at night.

Part 2: The Sky at Night

To create a simulated image of the night sky, we use the script in Listing 9-10. The script places 25 stars of random shape at random locations, and then stores the result in the image file, `nightsky.png`.

Listing 9-10. Creating an Image of the Sky at Night, `nightsky.py`

```
# create a fictitious night sky
from pylab import *
from random import randrange as rr
from star_patch import star

# parameters for the simulated night sky image
img_size  = 800
num_stars = 25

# star parameters: number of pointy edges and radius
min_num_points  = 5
max_num_points  = 11
min_star_radius = 2
max_star_radius = 10

# star parameter 'thinness' is on a scale of 1 to 10
min_thin = 5
max_thin = 9

# draw the night sky
figure(facecolor='k')
cur_axes = gca()

# patch stars
for i in range(num_stars):
    new_star = star(rr(min_star_radius, max_star_radius),
        rr(0, img_size) , rr(0, img_size), 'w', \
        rr(min_num_points, max_num_points), \
        rr(min_thin, max_thin)/10.0)
    cur_axes.add_patch(new_star)

# modify axis behaviour
axis([0, img_size, 0, img_size])
axis('scaled')
axis('off')

# save the figure without the extra margins
savefig('../images/nightsky', facecolor='k', edgecolor='k')
```

I've imported the function `randrange()` from the module *random* and renamed it to `rr()`, which I think is clearer to read. I then define a set of values you can tweak and observe the results. The values are self-explanatory and include such values as the image size and number of stars in the image.

The patching of stars is done in the `for` loop, which creates a new star with random values and adds it to the current figure. I then follow up by updating the image size and removing the axes. Finally, I save the image to file, `../images/nightsky.png`. Figure 9-7 shows random output from the simulated night sky.

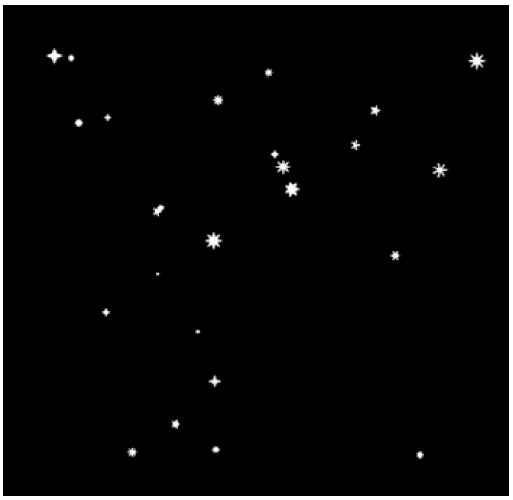

Figure 9-7. *A simulated (random) night sky*

Part 3: Flood Fill and Recursion

We now turn to something completely different: recursion. *Recursion* describes a scenario where a function calls itself. Some known recursion algorithms implement the factorial operation and Fibonacci sequences. We'll use recursion for image processing, specifically to fill an image using a flood fill algorithm.

Flood fill, sometimes also referred to as *bucket fill*, is an algorithm to fill a closed area of a specific color with a different color. This is a quite common operation in most image processing applications. Kids love to use it to paint digital coloring images.

To implement flood fill, we'll use recursion. In this implementation, we'll assume that the image to fill is given to us as a *NumPy* array, and more specifically as a 2-D array (i.e., a matrix). It's also possible to manipulate a *Pillow* `Image` object, but I prefer using a matrix for two reasons:

- It's more generic. I can port the flood-fill algorithm to other objects, as long as I can convert the objects to a *NumPy* array (matrix).
- It's easier to view the code by indexing over matrix elements than to use the `getpixel()` and `setpixel()` methods provided by the `Image` object.

So how does flood fill work? Flood fill starts by receiving a point to start filling from. If the point is the color to be converted, flood fill will change the color to the desired color. It then moves to a point adjacent to it—say, to the right—and calls itself. As the process continues, points to the right will start filling up with the new color. If the point to the right is not in the desired color (that is, it shouldn't be painted), the point to the top is checked, and the process resumes. This process is repeated for left and bottom points surrounding each point. The end process is a filled, closed object.

FLOOD FILL AND MINESWEEPER

The flood-fill algorithm can also be used in the coding of the game Minesweeper (shipped with Windows). You can use the algorithm to expand an area and reveal points adjacent to mines. The algorithm will follow a similar path, and one option would be to create a matrix of values corresponding to whether a square is empty (value 0) or adjacent to a mine (value equal to the number of mines it is adjacent to), with a different value indicating a mine (say, value -1). When the user clicks a square, the flood-fill algorithm kicks in and decides how many squares to reveal. If you're not familiar with Minesweeper, I suggest you *refrain* from trying it; the game is addictive!

Listing 9-11 presents a simple flood-fill implementation.

Listing 9-11. A Flood-Fill Implementation Using Recursion, `flood_fill.py`

```
from numpy import *
from sys import getrecursionlimit

def flood_fill(x, y, m, total):
    """A function to flood fill an image (matrix)."""

    if total > getrecursionlimit():
        return total

    # nothing to fill
    if m[x, y] != 1.0:
        return total

    m[x, y] = 0.5
    if(x-1 >=0):
        total = flood_fill(x-1, y, m, total+1)
    if(x+1 <= m.shape[0]-1):
        total = flood_fill(x+1, y, m, total+1)
    if(y-1 >=0):
        total = flood_fill(x, y-1, m, total+1)
    if(y+1 <= m.shape[1]-1):
        total = flood_fill(x, y+1, m, total+1)
    return total+1
```

The function `flood_fill()` is an implementation of the flood-fill algorithm described previously. I've indicated where recursion actually happens (the function calling itself) with boldface type.

The function accepts the values `x` and `y`, denoting the point to fill; `m`, which is the *NumPy* matrix; and `total`, which is a variable used to keep track of the recursion depth (i.e., how many times a function calls itself repeatedly).

I've chosen to fill all values corresponding to 1.0 with 0.5. Essentially, this means that, if the object is fully red (or green or blue, depending on the band selected), it will be changed to "half" red. You can modify the function `flood_fill()` to accept an original color and a new color as parameters; I chose not to do so because I think the code looks clearer that way.

Every time a function is called in a recursion, additional memory is consumed. Python limits the recursion depth with the value, `sys.getrecursionlimit()`. If the running code exceeds this limit, a recursion exception is raised. It's possible to increase this number by calling `sys.setrecursionlimit()`, but that's only a small fix; inevitably, you'll reach a memory limit, which might cause a system crash.

Therefore, it's best if your code can detect these events beforehand and alert the user if such an event transpired. I have chosen to do so by returning the value `total`. If `total` is greater than the maximum recursion depth, I can notify the user of the event.

It's also important to note that, if your night sky image gets larger or the size of stars get larger (e.g., a larger radius)—or if you save the image at a higher resolution (more points per star)—inevitably you will hit a recursion limit because the areas to fill get larger and larger. So while this is a viable option to fill objects, maybe a different algorithm should be employed for production-level code, so as to avoid the recursion limit (which you wouldn't want in production level code). For example, you might use `ImageDraw`'s `floodfill()` method, instead.

USING IMAGEDRAW FOR FLOOD FILL

The `ImageDraw` object also provides a `floodfill()` function, which may be used for the algorithm presented here. There are several reasons I chose to implement `flood_fill()` instead of using the `ImageDraw` function:

- I wanted to talk about recursion.
- `ImageDraw`'s `floodfill()` doesn't return information like the size of the filled region, which can be used to enhance the algorithm. That said, it's quite possible to use other methods to complement this, such as comparing the image before and after flood filling it.
- I wanted to show you how to tweak flood fill, for example, to include diagonals cells as adjacent cells (and not just up, down, left, and right).

So now that we have the `flood_fill()` function, how does that help us count the number of stars at night?

Part 4: Counting Objects

Counting objects is easy, once you've implemented `flood_fill()` (see Listing 9-12). The idea is simple: go through every point in your image and fill it. The return value from `flood_fill()` is the actual number of points filled. If there was nothing to fill, the value will be zero; but if `flood_fill()` fills an object, the return value will be nonzero, which indicates that `flood_fill()` found and filled an object. Future calls to `flood_fill()` for that pixel will not fill the object, as it is already filled. Now all that's required is to count the number of times `flood_fill()` returns a nonzero value, and you have the number of objects!

Listing 9-12. Counting Objects in a Picture

```
from pylab import *
from PIL import Image
from sys import getrecursionlimit
from flood_fill import flood_fill

# read the image, and retrieve the Red band
data = imread('../images/nightsky.png')[..., 0]
rows, cols = data.shape

# set all values that are nonzero to 1.0
# (could be values due to antialiasing)
data[nonzero(data)] = 1.0

# count the stars
count, recursion_limit_reached = 0, 0
```

```
for i in range(rows):
    for j in range(cols):
        tot = flood_fill(i, j, data, 0)
        if tot > getrecursionlimit():
            recursion_limit_reached += 1
        elif tot > 0:
            count+=1

if recursion_limit_reached:
    print("Recursion limit reached %d times" % recursion_limit_reached)
print("I counted %d stars!" % count)

imshow(data)
```

The script is an implementation of the preceding algorithm. We start by importing the necessary modules, as well as the module `flood_fill.py`, which contains the `flood_fill()` function implementation. Next, we open the image of the sky at night using *NumPy*'s `imread()` and select the red band with the indexing `[..., 0]`. I decided to work strictly on the red band; however, because we were dealing with black-and-white pictures, I could just as well have chosen any other channel (other than the transparency).

Next, I implement a simple threshold. What I do is change all nonblack values to white by setting all nonzero (i.e., not black) values to `1.0` (i.e., black); this is done using the `nonzero()` function call as follows: `data[nonzero(data)] = 1.0`. Other algorithms use a different approach, such as setting all values above and including 0.5 to 1.0 and all values below 0.5 to 0. In this particular case, the results would be very similar.

Now, I focus on using the `flood_fill()` function. I go through every pixel in the matrix and call the function `flood_fill()`. If the return value is greater than the recursion limit, I increment the number of times a recursion limit has been reached. If the return has not reached the recursion limit and is nonzero, I increment the count of objects.

Lastly, I report my results: the number of recursions that exceeded the maximum allowed value (for debugging purposes more than anything) and the number of stars counted. Here's a result from running the script on the night sky image presented earlier:

```
I counted 23 stars!
```

(We'll get to why that number is not 25 in the next section; your result may vary.)

To be sure I counted all the stars in the night sky picture—and to also be sure that I did not accidentally count objects that are not stars—I decided on some sort of visual feedback of the result. I do this in my last two line of code with a call to `imshow()`.

Part 5: Optimizing the Algorithm

So why did the algorithm return 23 stars and not 25? (See the `num_stars` value in Listing 9-10.) A plausible reason is that several stars overlapped. This would cause the algorithm to combine several objects into one. In real pictures (nonsterile, as presented in the example), there could be other reasons, and this is where you can start tweaking your image-processing algorithm.

But as you start working with "real" data, you'll find that sometimes the opposite happens; that is, the algorithm counts more objects than there really are. The reason for this could be because the images are not ideal, and even small specks, or noise, could throw off the number count. In that case, a possible solution would be to count only elements where the size is greater than a fixed value. That is, when reading the value returned by `flood_fill()`, you would discard objects where the size is too small. Another option would be to preprocess the image using a filter (see the "Image Filtering" section later in this chapter).

Another improvement to the algorithm could be giving it the capability to find the largest object. Again, this is quite possible by reading the value returned from the function `flood_fill()` and then sorting the results or finding the maximum.

And you can also try to evaluate the luminosity of the night sky, by counting the areas of all the stars. You might use this to estimate how clear the skies are. Or, in the case of a microscopic image, you could use this approach to help determine whether the size of a bacteria colony has changed.

Some real images might have objects so small that you'll need to think about flood filling diagonals, as well. For example, consider the character "x" drawn on a 3×3 pixel grid: there's no pixel that's adjacent to another unless you count diagonals. Modifying flood fill to include diagonals will combine the pixels that make up this "x" into one object.

The point of the matter, now that data is accessible as a *NumPy* matrix, is that you can implement whatever algorithm or image-processing idea you might have. In many cases, you don't even have to resort to the matrix level; *Pillow* provides a good number of support functions.

Image Arithmetic

Pillow provides a set of arithmetic operations via the module *ImageChops* (Chops is short for channel operations). In the night sky example, some people would prefer working on a white background; this could save quite a bit of ink if you're printing the images (see Figure 9-8). Per the previous section, you could transfer the image to a *NumPy* array and then convert it; but in such a simple case, it makes more sense to use the *ImageChops* `invert()` function:

```
# display an image and its inverse
from PIL import Image, ImageChops
im = Image.open('../images/nightsky.png')
new_img = Image.new('RGB', (im.size[0]*2, im.size[1]))
new_img.paste(im, (0, 0))
new_img.paste(ImageChops.invert(im), (im.size[0], 0))
new_img.show()
```

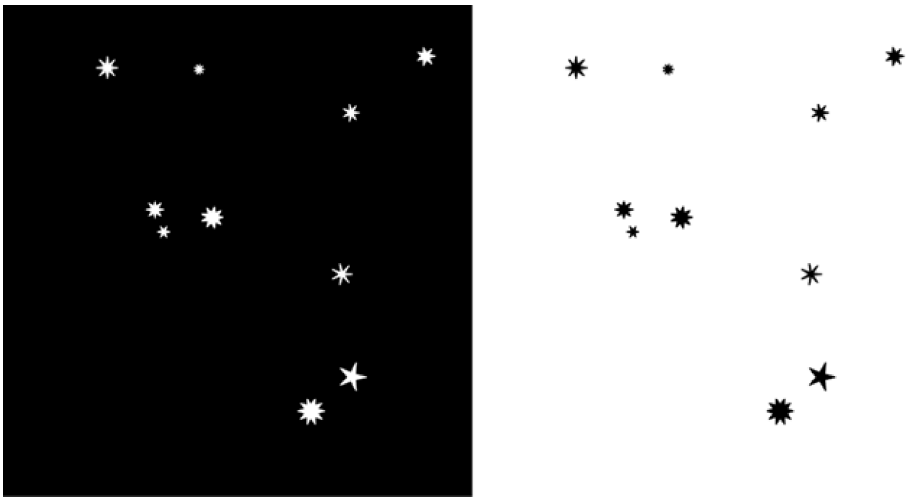

Figure 9-8. *Inverting an image: the original is on the left, and the inverted image is on the right*

In Figure 9-8, I've used the image generated by the script `nightsky.py` (see Listing 9-10) with `num_stars=10`, `min_star_radius=10` and `max_star_radius=30` to show a more pronounced effect of the image inversion.

Table 9-3 lists some additional *ImageChops* operations. Notice that *ImageChops* operations operate only one channel (L) or RGB images.

Table 9-3. *Some ImageChops Operations*

Function	Description
`add(img1, img2, scale=1.0, offset=0)`	Adds two images as follows: `(img1+img2)/scale+offset`. The default values of `scale` and `offset` mean a simple addition.
`constant(img1, value)`	Returns an image of size `img1` filled with color `value`.
`darker(img1, img2)`	Returns an image with the darker pixel from both images. This a minimum of the two images, on a pixel-by-pixel level.
`difference(img1, img2)`	Returns the absolute difference of two images. This is `abs(img1-img2)`, on a pixel-by-pixel level.
`lighter(img1, img2)`	Returns an image with the lighter pixel from both images. This a maximum of the two images, on a pixel-by-pixel level.
`subtract(img1, img2, scale=1.0, offset=0)`	Subtracts two images as follows: `(img1-img2)/scale+offset`. The default values of `scale` and `offset` mean a simple subtraction.

There are additional functions available in *ImageChops*; to learn more, check out either `help(ImageChops)` or the *Pillow* web site (`http://pillow.readthedocs.org/en/latest/reference/ImageChops.html`).

You can create some interesting effects using these simple operations. And these effects can in turn be used for some fast image-processing algorithms. Listing 9-13 presents a script that uses the `lighter()` method on two night sky images. To follow along, run the `nightsky.py` script and rename the generated file `images/nightsky.png` to `images/nightsky1.png`. Now do it again, this time renaming the generated image to `images/nightsky2.png`.

Listing 9-13. Using lighter() on Two Images

```
from PIL import Image, ImageDraw, ImageFont, ImageChops
from matplotlib import font_manager

# read the images
img1 = Image.open('../images/nightsky1.png')
img2 = Image.open('../images/nightsky2.png')

# create a new image, made of the lighter of the two
img3 = ImageChops.lighter(img1, img2)

# create a collage of three images
width, height = img1.size
delta = 10
img = Image.new('RGB', (width*2+delta, height*2+delta), (255, 255, 255))
img.paste(img1, (0, 0))
img.paste(img2, (width+delta, 0))
img.paste(img3, ((width+delta)//2, height+delta))
```

```
# annotate the images with text
font_str = font_manager.findfont('Vera')
ttf = ImageFont.truetype(font_str, 54)

draw = ImageDraw.Draw(img)
draw.text((delta, delta), 'Night Sky (1)', fill='white', font=ttf)
draw.text((delta*2+width, delta), 'Night Sky (2)', fill='white', font=ttf)
draw.text(((width+delta)//2+delta, height+delta*2), \
    'Combined', fill='white', font=ttf)

# display the final image
img.show()
```

I've made a collage and separated the images with a white `delta` band. Figure 9-9 shows the result.

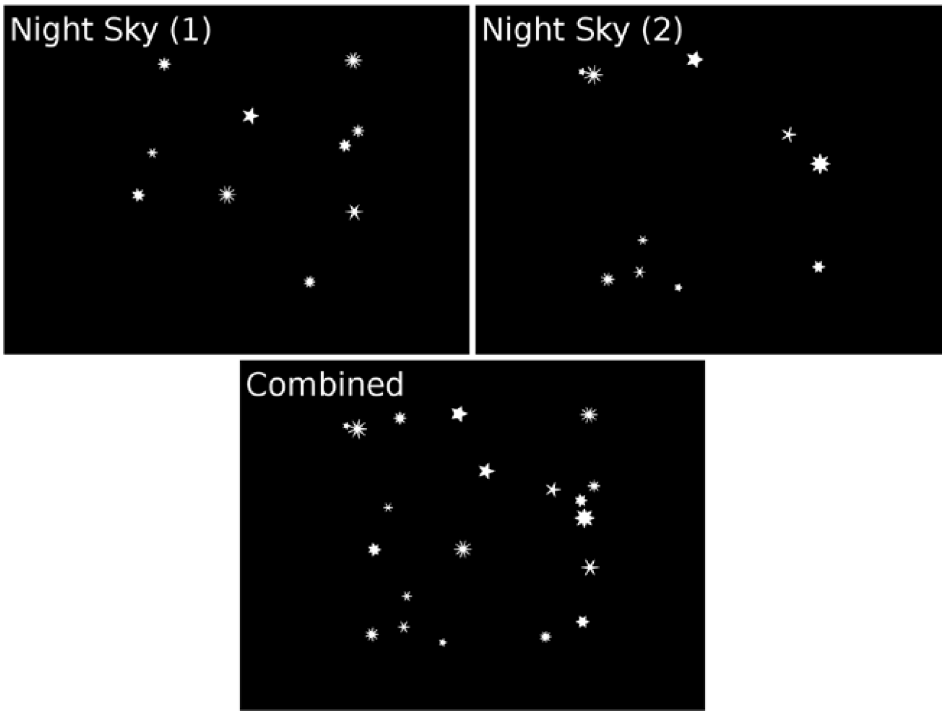

Figure 9-9. *Using lighter( )*

It's interesting to note that, in this specific case, using the function `add()` would have resulted in a similar image.

Image Filtering

Most GUI-based image-processing applications come with bundle of image filters. There's a wide variety of filters available, and different applications group them into different categories. Some of the common filtering categories are blur, enhancement, edge detection, and so on.

From an image-processing standpoint, *image filters* are known operations that help us achieve a specific effect. For example, I once used the counting objects algorithm (see Listing 9-12) to count the number of bubbles in a printed circuit board soaked in water. As you probably realize, pictures obtained from a real-life image are not as

sterile as those presented in the sky at night example. And so, prior to using the algorithm, I had to clean up the images. By "clean up," I mean I had to filter the image using known filters. I ended up using a threshold combined with a median filter, and then converted the image to a 1-bit (black-and-white) version prior to running the algorithm.

The following text assumes you have some background in image filtering. If not, my suggestion is that you experiment with a GUI application such as GIMP to get a feel for what filters you can use and how they can help you with basic image processing. Once you have the preprocessing figured out; that is, once you know what filters you want to run on your image prior to the final algorithm, you can implement the filters with a Python script that uses *Pillow* filters (see Figure 9-10). (You might not even require a final algorithm if you select the proper filters.)

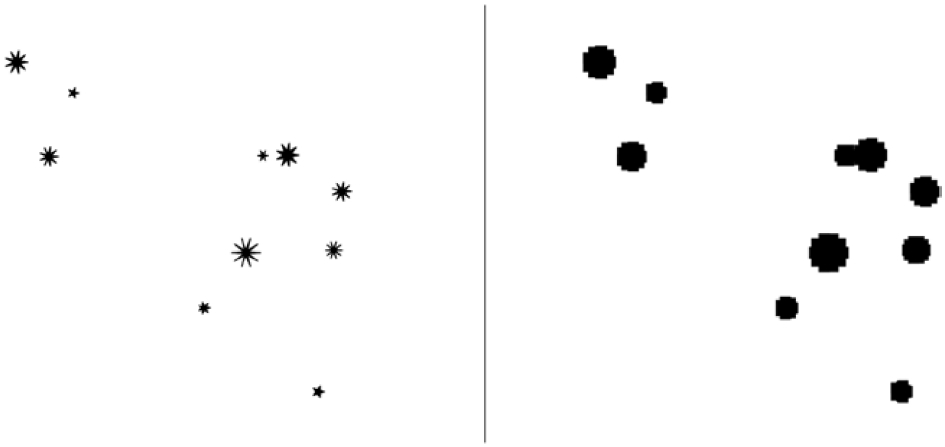

Figure 9-10. *Filtering an image: left is the original, and right shows the image filtered with a* `MinFilter` *set to* `15`

Pillow provides us with the class `ImageFilter`, which supports a good number of filters. To use `ImageFilter`, import it as follows: `from PIL import ImageFilter`. Once you've imported `ImageFilter`, call the `filter()` method that's part of the `Image` object (not the `ImageFilter` object) to filter an image:

```
>>> from PIL import Image, ImageChops, ImageFilter
>>> img = Image.open('../images/nightsky.png')
>>> inv_img = ImageChops.invert(img)
>>> filt_img = inv_img.filter(ImageFilter.MinFilter(15))
>>> filt_img.show()
```

In the preceding example, I've used the night sky images you've seen before and inverted the output so as to work on black stars over white background. I then filtered the image using a `MinFilter` filter (see Figure 9-10). The `MinFilter` works on a pixel-by-pixel level. For every pixel, it returns the minimum pixel from the square of size `n` (15, in the example) centered on the given pixel. As you can see, even from this small example, there's quite a bit to be gained by working with image filters.

`ImageFilter` provides *fixed* image-enhancement filters that are easily distinguished by their capitalized names:

```
>>> from PIL import ImageFilter
>>> [filt for filt in dir(ImageFilter) if filt.isupper()]
```

```
['BLUR', 'CONTOUR', 'DETAIL', 'EDGE_ENHANCE', 'EDGE_ENHANCE_MORE', 'EMBOSS',
'FIND_EDGES', 'SHARPEN', 'SMOOTH', 'SMOOTH_MORE']
```

By *fixed*, I mean that the filters accept no parameters. To use these filters, call the `filter()` method with the fixed filter, as follows:

```
>>> new_img = img.filter(ImageFilter.CONTOUR)
```

The names of these filters should provide direction as to what action they perform.

`ImageFilter` also provides nonfixed filters (i.e., filters that accept parameters). Table 9-4 lists some additional filters supported by the `ImageFilter` object.

Table 9-4. *Some Image Filters*

Function	Description
`MaxFilter(size=3)`	For every pixel in the original image, returns the pixel with the maximum value from a square of width `size` placed around the original pixel. `size` must be odd (3, 5, 7, . . .).
`MedianFilter(size=3)`	For every pixel in the original image, returns the median pixel from a square of width `size` placed around the original pixel. `size` must be odd (3, 5, 7, . . .).
`MinFilter(size=3)`	For every pixel in the original image, returns the pixel with the minimum value from a square of width `size` placed around the original pixel. `size` must be odd (3, 5, 7, . . .).
`ModeFilter(size=3)`	For every pixel in the original image, returns the most common pixel from a square of width `size` placed around the original pixel. `size` must be odd (3, 5, 7, . . .).

Making Movies

Making movies is a somewhat advanced topic, and here I'll assume you have some basic knowledge of movie formats.

Movies are sequences of images. Once you know how to operate on an image, you can easily apply that knowledge to operate on a movie, as well.

Splitting Movies

The method I'll present here involves splitting a given movie into individual images, operating on these images, and then combining the images again to create a movie. Of course, if you wish to create a movie from a sequence of images, the first step (splitting a given movie) is not required. It should also be noted that movies also contain a sound track; in this discussion, the sound track (audio) is discarded.

To be able to follow along, you will need to download and install an application that has the ability to split a movie into individual frames. It should also be able to do the opposite: create a movie from a sequence of images. I chose to use the MPlayer application, The Movie Player, available at `http://mplayerhq.hu`.

■ **Caution** Be sure you download the MPlayer application from the official web site (`http://mplayerhq.hu`) to avoid any malware issues. If possible, use the OS package manager (Linux) to install the proper application.

If you use Linux or Mac, once you have MPlayer installed, you can easily access it via the terminal since it will be available to all users. However, in Windows, the location of MPlayer is important, and you should remember where MPlayer is installed. In this section, I assume the location is at c:\mplayer.

To split a movie named `movie.avi` to individual images, use the following command line in a Linux/Mac terminal. Note that this will discard any audio information:

```
$ mplayer -vo png movie.avi
```

In Windows, this looks slightly different:

```
C:\mplayer> mplayer -vo png movie.avi
```

In Windows, if you're running scripts outside the c:\mplayer directory, you will need to provide to full path to mplayer.exe or add it to the path, as follows:

```
C:\> path=%PATH%;c:\mplayer
```

The result is a sequence of indexed PNG images, generated in the working directory, as follows: 00000001.png, 00000002.png, and so on.

Creating Movies from Images

Combining images into movies is similar to splitting a movie into images, only in reverse. To generate a video from images, we use the MEncoder utility. This utility is part of the MPlayer application, so there's no need to install additional software.

To generate a movie in Mpeg4 format from a list of images, issue the following:

```
mencoder mf://*.png -mf type=png fps=25 -ovc lavc -o new.avi
```

I chose some command-line options in this example. First, I chose the frame rate, `fps` (which stands for frame-per-second), which I set to 25. I also chose to use a movie codec, Mpeg-4, as evident in this option: `vcodec=mpeg4`. For a full account of these options, refer to MPlayer's web site (a direct link: http://www.mplayerhq.hu/DOCS/HTML/en/menc-feat-enc-images.html) or consult with the online help.

Example: A Fractal Movie

In this example, we create a sequence of images from the Mandelbrot fractal presented in Chapter 7.

The example is straightforward. We generate fractal images with an increasing number of iterations. We store the images to the files mandelbrot0001.png, mandelbrot0002.png, and so on. Once we're done, we use the *os* module to execute a few shell commands. We do this with a call to `os.system()` with the command to execute (calling MEncoder; calling MPlayer) as a string. First, we change to the directory where the images are stored. Next, we call MEncoder to encode files. Once the encoding is complete, we use MPlayer to view the file. We can also view the file manually by opening the generated file, Mandelbrot.avi. Lastly, we change to our previous working directory. Listing 9-14 shows the entire script to generate a Mandelbrot fractal movie.

Listing 9-14. The Mandelbrot fractal movie

```
from PIL import Image, ImageOps
import os

# creates an image of the Mandelbrot set
res = 400
```

```
for iters in range(50):
    img = Image.new("L", (res, res), 255)
    for im in range(res):
        for re in range(res):
            z = 0
            # a scaling to show the "interesting" part of
            # the Mandelbrot fractal
            c = (re*2/res-1.5)+1j*(im*2/res-1)
            for k in range(iters):
                z = z**2+c
                if abs(z)>2:
                    img.putpixel((re, im), 255-k*255/iters)
                    break

    # create a uniform distribution of gray levels
    img = ImageOps.equalize(img)

    # save to file
    img.save('../images/mandelbrot_%04d.png' % iters, dpi=(150,150))

last_path = os.getcwd()

# handle Linux/Mac and Windows properly
os.chdir('../images')
if os.sys.platform.startswith('win'):
    mee = 'c:/mplayer/mencoder '
    mep = 'c:/mplayer/mplayer '
else:
    mee = 'mplayer '
    mep = 'c:/mplayer '

# encode the movie
os.system(mee+'mf://mandelbrot*.png -mf type=png -ovc lavc -o mandelbrot.avi')

# play the movie
os.system(mep+'mandelbrot.avi')
os.chdir(last_path)
```

Combining these two abilities (splitting movies and generating movies from sequences of images) with the image-processing methods shown previously in this chapter will get you started on movie processing in no time.

Final Notes and References

Image processing is a large field, and it is gaining more and more popularity as computers increase in performance. And image processing is only two dimensional; nowadays we see more and more 3-D data processing, as well, including video.

Armed with Python, the Python Imaging Library, and *NumPy*, you can prototype even complex image-processing tasks. However, image processing requires a great deal of memory and processing power; as you work with images, you'll realize you may require faster tools, and you may even need to port parts of your code to a lower-level programming language such as C to gain performance. Nevertheless, Python is an excellent prototyping environment; it provides fast responses in an interactive environment, and it can help you define your image-processing algorithm.

Additional information can be found at the following sites:

- The Python Imaging Library, http://www.pythonware.com/library/pil/handbook/
- GIMP, http://www.gimp.org/docs/
- MPlayer, The Movie Player, http://www.mplayerhq.hu