© Jones and Bartlett Publishers, LLC. NOT FOR SALE OR DISTRIBUTION

2 NumPy and SciPy

Numerical arrays are vectors, matrices, and tensors that are important to the speedy manipulation of data. There are two reasons that arrays are so useful: (1) they provide dramatic savings in computational time and (2) they can drastically reduce the amount of coding that is required to accomplish a task. The NumPy and SciPy packages offer a wide range of array functions and data types that will be used in almost all of the applications in this book.

2.1 Introduction to NumPy and SciPy

NumPy and SciPy are packages designed for the manipulation of arrays and scientific data. The NumPy package contains the array manipulation routines and the SciPy package contains a variety of scientific packages that are not part of the standard Python installation. Currently, the packages can be downloaded from http://scipy.org. This website also contains documentation for the myriad functions included in these libraries.

The predecessor to NumPy was a package named Numeric. Users who are switching from Numeric to NumPy should note that there are several differences between the two packages, including variable name changes, function name changes, and function operation changes. One of the major changes that affect work in this book is the removal of type names (such as Float, Integer, and Complex). Functions such as **nonzero** now return a tuple instead of an array, and functions such as **eig** (eigenfunctions) return vectors as columns instead of rows in a matrix. There are many more changes, and users who are switching will need to be on the alert for them.

2.2 Basic Array Manipulations

An array is a vector, matrix, or tensor. There are several methods in which arrays can be created, as shown in Code 2-1. The functions **zeros** and **ones** create arrays filled with either 0's or 1's. Some versions of NumPy will create these arrays as integers while newer versions create them as floats. The type of data contained in an array can be controlled with data types *int*, *float*, *complex, double, byte, long*, etc.

```
>>> from numpy import *
>>> a = array( [1,2,3,4.5] ) # creates a vector from a list
>>> b = zeros( 4 )           # creates a vector of length 4
>>> c = ones( 4 )        # vector of length 4 filled with 1
```

© Jones and Bartlett Publishers, LLC. NOT FOR SALE OR DISTRIBUTION

```
>>> d = array( [[1,2],[3,4]] )  # matrix
>>> f = zeros( (4,3) )       # matrix of size 4x3
>>> g = zeros( (4,3), int )  # matrix of integers
>>> h = zeros( (4,3), complex ) # complex valued matrix
```

Code 2-1

It should be noted that the argument to the **zeros** function in the creation of a matrix is a tuple that contains two integers. These integers indicate the vertical and horizontal dimension of the matrix. In Code 2-1, *f* is thus a matrix with 4 rows and 3 columns. Creating a tensor merely requires a tuple with more integers, as illustrated in the example shown in Code 2-2.

```
>>> tensor = zeros( (3,4,5), float )
```

Code 2-2

Python can print out an entire array in a single command, as shown in Code 2-3. Larger arrays are treated differently in various versions of NumPy. Older versions would attempt to print out the entire array, which can lock the Python shell for some time. Newer versions of NumPy print out just a few elements of the array and uses ellipses (…) to indicate that only a small fraction of the elements has been printed.

```
>>> f = zeros( (4,3) )
>>> f
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

Code 2-3

2.3 Basic Math

One of the many advantages of arrays is that a single command is used to perform basic math operations. In other languages such as C and Java these operations usually require a *for* loop or nested loops. Code 2-4 demonstrates the simple arithmetic of vectors. For these operations the arrays must have compatible lengths or an error occurs. Access to the elements in an array is achieved through slicing.

```
>>> f = zeros( (4,3) )
>>> a = array( [1,2,4.3] )
>>> b = array( [-0.9, 3, 4] )
# add two vectors
>>> a + b
array([ 0.1,  5. ,  8.3])
# subtract two vectors
>>> a - b
array([ 1.9, -1. ,  0.3])
```

© Jones and Bartlett Publishers, LLC. NOT FOR SALE OR DISTRIBUTION


```
# multiple two vectors
>>> a * b
array([ -0.9,   6. ,  17.2])
# divide two vectors
>>> a / b
array([-1.11111111,  0.66666667,  1.075   ])
```

Code 2-4

```
#slicing
>>> c = array( {[1,2,3,4],[5,6,7,8],[9,10,11,12]] )
>>> c[0] # first row
array([1, 2, 3, 4])
>>> c[-1] # last row
array([ 9, 10, 11, 12])
>>> c[0,0] # first row, first column
1
>>> c[1,2]
7
>>> c[:2] # first two rows
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

Code 2-5

Code 2-5 demonstrates a few slices. Extracting an element in a matrix requires a vertical and horizontal location. As seen with *c[1,2]* the slices receive the vertical dimension first and the horizontal dimension second. Extracting a row from a matrix is performed by commands such as *c[0]*. This makes sense since the first argument in the slice is the vertical dimension. Extracting a column in a matrix requires that all of the elements in the vertical dimension be extracted for a single horizontal dimension, as demonstrated in Code 2-6. The colon indicates that all of the first dimension is used, and the 1 indicates that only a single index of the second dimension is used.

```
>>> c[:,1]
array([ 2,  6, 10])
```

Code 2-6

The dimensions of an array are returned by *shape*, as shown in Code 2-7. Note that this is not calling a function (since there are no parentheses). The *dtype* variable returns a code to indicate the type of data contained in an array. In the case that follows, the code 'i4' indicates an integer that uses 4 bytes.

```
>>> c.shape
(3, 4)
>>> c.dtype
dtype('<i4')
```

Code 2-7

© Jones and Bartlett Publishers, LLC. NOT FOR SALE OR DISTRIBUTION

2.4 More on Multiplication

There are several ways to multiply arrays. The command (*a*b*) shown in Code 2-4 performs an element-by-element multiply, which is

$$c_i = a_i b_i; \quad \forall i. \tag{2-1}$$

The *inner product* (or *dot product*) returns a scalar value,

$$c = \sum_i a_i b_i. \tag{2-2}$$

The *outer product* creates a matrix from two vectors, as in

$$c_{i,j} = a_i b_j; \quad \forall i, j. \tag{2-3}$$

Code 2-8 demonstrates these multiplications. The first command performs the element-by-element multiplication in Equation (2-1). The **dot** command performs the inner product, and the **outer** command performs Equation (2-3). An alternative is to use the function **multiply.outer** shown in Code 2-8, which uses a function named **outer** from the module named *multiply*.

```
>>> c = a * b
>>> c
array([ -0.9,   6. ,  17.2])
>>> c = dot( a,b )
>>> c
22.3
>>> c = outer(a,b)
>>> c
array([[ -0.9 ,   3.  ,   4.  ],
       [ -1.8 ,   6.  ,   8.  ],
       [ -3.87,  12.9 ,  17.2 ]])
>>> c = multiply.outer(a,b)
```

Code 2-8

The NumPy package offers other forms of matrix math such as outer addition and subtraction. One example is the outer addition shown in Code 2-9, which performs

$$c_{i,j} = a_i + b_j; \quad \forall i, j \tag{2-4}$$

```
>>> add.outer( a,b )
array([[ 0.1,  4. ,  5. ],
       [ 1.1,  5. ,  6. ],
       [ 3.4,  7.3,  8.3]])
```

Code 2-9

© Jones and Bartlett Publishers, LLC. NOT FOR SALE OR DISTRIBUTION

There are many more functions than are listed here, and the reader is encouraged to consult NumPy documentation.

2.5 More Math

The NumPy package offers math functions that apply to an entire array. Code 2-10 begins with a random array and then computes the square root, square, exponential, and logarithm; the last function performs the $\log_2$. Each of these functions performs the mathematical operation on each element in the array. There are other functions as well, including trigonometric ones (sin, cos, tan, tan2, cosh, etc.).

```
>>> from numpy import random, sqrt, exp, log, log2
>>> a = random.ranf( (4,3) )
>>> a
array([[ 0.26978006,  0.63229368,  0.79582955],
       [ 0.96121207,  0.29647525,  0.5989877 ],
       [ 0.45980855,  0.76866697,  0.5310724 ],
       [ 0.58531764,  0.28331364,  0.59143308]])
>>> sqrt( a )
array([[ 0.51940356,  0.79516896,  0.89209279],
       [ 0.98041423,  0.5444954 ,  0.77394296],
       [ 0.67809185,  0.87673654,  0.72874715],
       [ 0.76506054,  0.53227215,  0.76904687]])
>>> a**2
array([[ 0.07278128,  0.3997953 ,  0.63334468],
       [ 0.92392865,  0.08789757,  0.35878627],
       [ 0.21142391,  0.5908489 ,  0.2820379 ],
       [ 0.34259673,  0.08026662,  0.34979309]])
>>> exp( a )
array([[ 1.30967637,  1.88192216,  2.21627876],
       [ 2.61486395,  1.34510926,  1.82027521],
       [ 1.58377075,  2.15688913,  1.70075522],
       [ 1.79556123,  1.32752146,  1.80657553]])
>>> log( a )
array([[-1.31014824, -0.45840131, -0.22837024],
       [-0.03956022, -1.21579155, -0.51251421],
       [-0.77694506, -0.26309748, -0.63285692],
       [-0.53560061, -1.26120073, -0.52520673]])
>>> log2( a )
array([[-1.89014437, -0.66133329, -0.32946862],
       [-0.05707333, -1.75401644, -0.73940171],
       [-1.12089479, -0.37956943, -0.91301954],
       [-0.77270835, -1.81952804, -0.75771315]])
```

Code 2-10

2.5.1 Equals or Copy

The one math operator that does behave as expected at times is the equals sign. The equals sign does not copy one array into another but rather copies the address. Basically, this means that both arrays are pointing to the same location in memory and there is only one version of the array. Changes to one will cause changes to the other

© Jones and Bartlett Publishers, LLC. NOT FOR SALE OR DISTRIBUTION

variable. The code in Code 2-11 shows the creation of an array *b*, and the *c* is set equal to *b*. The first element of *c* is changed, but it is *b* that is printed and, as can be seen, the first element of *b* was changed. In this case *b* and *c* are using the same memory in the computer, and there is no difference between the two.

```
>>> b = array( [5.6, 4.2, 3.8] )
>>> c = b
>>> c[0] = -1
>>> b
array([-1. ,  4.2,  3.8])
```

Code 2-11

For applications in this book, this feature is a bit unfortunate as it is desired that *b* and *c* occupy different locations in memory, changes in one array will thus not change the other. There are, however, two methods to easily accomplish this. The first is to use the **copy.copy** function, and the second is to simply add 0. These are shown in Code 2-12.

```
>>> import copy
>>> b = array( [5.6, 4.2, 3.8] )
>>> c = copy.copy( b )
>>> c[0] = -1
>>> b
array([ 5.6,  4.2,  3.8])
>>> c
array([-1. ,  4.2,  3.8])
>>>
>>> c = b + 0
>>> c[0] = -1
>>> b
array([ 5.6,  4.2,  3.8])
>>> c
array([-1. ,  4.2,  3.8])
```

Code 2-12

The addition of a 0 may seem a bit odd at first. The use of the addition sign will actually create a copy in memory of the array. The *b+0* will create a new array, and with the use of 0 it has the same values as *b*. The equal sign still shares the memory with the data on the right-hand side of the equation, but this time it is the *b+0* array, not the original *b*.

2.5.2 Comparisons

The elements in two arrays can be compared to each other as long as the arrays have axes (or dimensions) of the same length. For example, Code 2-13 shows the comparison of two vectors to determine if the values in one vector are greater than the values in the other vector. The **greater** function returns an array of Boolean values that represent the element by element comparisons.

© Jones and Bartlett Publishers, LLC. NOT FOR SALE OR DISTRIBUTION


```
>>> from numpy import greater
>>> a = random.rand( 5 )
>>> b = random.rand( 5 )
>>> a
array([ 0.46302608,  0.77595201,  0.9810886 ,  0.71411958,  0.14652436])
>>> b
array([ 0.32756491,  0.37556837,  0.55483111,  0.12703247,  0.72902361])
>>> greater( a, b )
array([True, True, True, True, False], dtype=bool)
```

Code 2-13

Other comparison functions are **less**, **less_equal**, **greater_equal**, **equal**, and **not_equal**. The functions return an array of Boolean values that can be converted to integers (or other types) using **astype**, as shown in Code 2-14. The sign of an array is returned by the **sign** function, as shown in Code 2-15.

```
>>> greater( a, b ).astype(int)
array([1, 1, 1, 1, 0])
```

Code 2-14

```
>>> a = random.rand( 5 )-0.5
>>> a
array([-0.49158702,  0.3527218 , -0.01372066, -0.4705638 , -0.48136565])
>>> sign( a )
array([-1.,  1., -1., -1., -1.])
```

Code 2-15

Finally, it is possible to combine results from the logical operators using functions such as **logical_and**, as shown in Code 2-16. These logical functions perform the functions on the values in the array. To perform functions on the bit patterns in the data, functions such as **bitwise_and** are used.

```
>>> a = random.rand( 5 )
>>> a
array([ 0.07869375,  0.94517074,  0.62396456,  0.11409584,  0.47865114])
>>> a1 = greater( a, 0.5 )
>>> a1
array([False, True, True, False, False], dtype=bool)
>>> a2 = less( a, 0.9 )
>>> a2
array([True, False, True, True, True], dtype=bool)
>>> logical_and( a1, a2 )
array([False, False, True, False, False], dtype=bool)
>>> logical_or( a1, a2 )
array([True, True, True, True, True], dtype=bool)
>>> logical_xor( a1, a2 )
```

© Jones and Bartlett Publishers, LLC. NOT FOR SALE OR DISTRIBUTION

```
array([True, True, False, True, True], dtype=bool)
>>> logical_not( a1)
array([True, False, False, True, True], dtype=bool)
```

Code 2-16

2.5.3 More on Slicing

Code 2-5 showed some of the ways to extract information from an array. Of course, the same slicing techniques can be used to alter the data in an array such as the one shown in Code 2-17.

```
>>> c = array( [[1,2,3,4],[5,6,7,8],[9,10,11,12]] )
>>> c[0,2] = -1
>>> c[0]
array([ 1,  2, -1,  4])
>>> c[1] = ones( 4 )
>>> c[:,2] = zeros( 3 ) -2
>>> c
array([[ 1,  2, -2,  4],
       [ 1,  1, -2,  1],
       [ 9, 10, -2, 12]])
```

Code 2-17

A caution needs to be exhibited here. Code 2-17 illustrates an array of integers, and attempting to convert an element to a float will not change the data type. This is shown in the first example in Code 2-18. However, an entire array can be changed to a new data type by adding a scalar of the higher-level data type. The last example uses the **astype** command to change the data type of the entire array. The **dtype** command returns the type of data stored in an array.

```
>>> c[0,2] = 5.5
>>> c
array([[ 1,  2,  5,  4],
       [ 1,  1, -1,  1],
       [ 9, 10, -1, 12]])
>>> c = c + 0.0 # adding a float to everything
>>> c
array([[  1.,   2.,   5.,   4.],
       [  1.,   1.,  -1.,   1.],
       [  9.,  10.,  -1.,  12.]])
>>> c = c.astype(int)
>>> c
array([[ 1,  2,  5,  4],
       [ 1,  1, -1,  1],
       [ 9, 10, -1, 12]])
>>> c.dtype
dtype('int32')
```

Code 2-18

© Jones and Bartlett Publishers, LLC. NOT FOR SALE OR DISTRIBUTION

The methods shown so far can access rows, columns, or individual elements in a regular pattern in an array. Accessing a regular pattern of elements can be achieved by the third argument in slicing. As demonstrated in Chapter 1, the following will extract every other element *a[::2]*. If *a* is a matrix, then this will extract every other row.

Accessing an irregular pattern of elements is also accomplished through list slicing. In Code 2-19 a vector is created using the **arange** function. This function is quite similar to the **range** function except that it returns an array rather than a list. A list *b* is then created, which contains the indices of the array that are to be accessed. In the command *a[b]*, the elements of *a* are extracted according to the list *b*. In the following command those same elements are changed. It should be noted that they are changed according to the order of *b*. Thus, *a[b[0]]* becomes the first element in the tuple and so on. This is an excellent method of accessing irregular patterns of data, which does occur often in the applications.

```
>>> a = arange( 15 )/4.
>>> a
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2.  ,  2.25,
2.5 ,  2.75,  3.  ,  3.25,  3.5 ])
>>> b = [6,9,14,7,3]
>>> a[b]
array([ 1.5 ,  2.25,  3.5 ,  1.75,  0.75])
>>> a[b] = -1,-2,-3,-4,-5
>>> a
array([ 0.  ,  0.25,  0.5 , -5.  ,  1.  ,  1.25, -1.  , -4.  ,  2.  , -2.  ,
2.5 ,  2.75,  3.  ,  3.25, -3.  ])
```

Code 2-19

2.5.4 Sorting and Shaping

There are several other useful functions that are used throughout this book. An array can be sorted using the **sort** command. The indices of the sort order can also be returned using the **argsort** command. These are shown in Code 2-20. The **sort** command changes the arrangement of the elements in the array, whereas the **argsort** command indicates the order of the elements but does not change the arrangement of the data.

```
>>> a = random.rand( 5 )
>>> a
array([ 0.09541768,  0.02161618,  0.46247117,  0.83177906,  0.95871478])
>>> a.argsort()
array([1, 0, 2, 3, 4])
>>> a.sort()
>>> a
array([ 0.02161618,  0.09541768,  0.46247117,  0.83177906,  0.95871478])
```

Code 2-20

The **take** command will extract from an array the elements according to a list. In Code 2-21 an array *a* and a list *b* are created. The **take** command gathers the data from *a*

© Jones and Bartlett Publishers, LLC. NOT FOR SALE OR DISTRIBUTION

according to the prescription defined in *b*. An error will occur if a number in *b* is outside of the range of *a*.

```
>>> a = random.rand( 5 )
>>> a
array([ 0.99325464,  0.37953944,  0.72033516,  0.58302183,  0.65205877])
>>> b = [3,1,2,1,1]
>>> take( a, b )
array([ 0.58302183,  0.37953944,  0.72033516,  0.37953944,  0.37953944])
```

Code 2-21

Arrays can be reshaped as long as the number of elements does not change. The **ravel** function will convert a matrix to a long vector, as shown in Code 2-22. The **reshape** command can convert the array to any other shape as long as the number of elements is the same, as shown in Code 2-23. The **concatenate** function joins two arrays. A second argument can be used to indicate which axes are being joined, as shown in Code 2-24.

```
>>> a = random.ranf( (3,2) )
>>> a
array([[ 0.04897374,  0.35842065],
       [ 0.60357182,  0.98938948],
       [ 0.39334855,  0.44884645]])
>>> ravel( a )
array([ 0.04897374,  0.35842065,  0.60357182,  0.98938948,  0.39334855,
0.44884645])
```

Code 2-22

```
>>> a.reshape( (2,3) )
array([[ 0.04897374,  0.35842065,  0.60357182],
       [ 0.98938948,  0.39334855,  0.44884645]])
```

Code 2-23

```
>>> a = random.ranf( (3,2) )
>>> b = random.ranf( (2,2) )
>>> a
array([[ 0.20894554,  0.72381952],
       [ 0.08718544,  0.9803592 ],
       [ 0.51509276,  0.72995728]])
>>> b
array([[ 0.33436572,  0.23557135],
       [ 0.78765853,  0.41989476]])
>>> concatenate( (a,b) )
array([[ 0.20894554,  0.72381952],
       [ 0.08718544,  0.9803592 ],
       [ 0.51509276,  0.72995728],
       [ 0.33436572,  0.23557135],
       [ 0.78765853,  0.41989476]])
>>> b = random.ranf( (3,3) )
```

© Jones and Bartlett Publishers, LLC. NOT FOR SALE OR DISTRIBUTION


```
>>> concatenate( (a,b),1 )
array([[ 0.20894554,  0.72381952,  0.44493734,  0.52981483,  0.17825647],
       [ 0.08718544,  0.9803592 ,  0.57896141,  0.26381222,  0.13927053],
       [ 0.51509276,  0.72995728,  0.15169618,  0.51958984,  0.69107468]])
```

Code 2-24

Finally, a useful function is **indices** that will return two (or more) arrays with the indices indicated as rows or columns. The example in Code 2-25 considers arrays that are 4×3. The indices returns two matrices (in the form of a tensor) with the rows or columns incrementing.

```
>>> indices( (4,3) )
array([[[0, 0, 0],
        [1, 1, 1],
        [2, 2, 2],
        [3, 3, 3]],

       [[0, 1, 2],
        [0, 1, 2],
        [0, 1, 2],
        [0, 1, 2]]])
```

Code 2-25

2.5.5 Random Numbers

NumPy contains a random module that can be imported with *from numpy import random*. This module contains many routines for generating random numbers. A vector of *N* random numbers with values between 0 and 1 is obtained by *random.rand(N)*. A matrix of random numbers uses a different function *random.ranf((V,H))* where *V* and *H* are the vertical and horizontal dimensions of the array. One case that is used later in the book is a set of random numbers based on a Gaussian distribution. Code 2-26 shows the **random.normal** function with no argument, which returns a number from a standard distribution. By adding two arguments, the user defines the mean and standard deviation of the distribution. The third argument returns an array of values along the specified distribution. The *random* module has several other types of random number generators.

```
>>> random.normal()
0.9299042168312539
>>> random.normal(0.5, 0.1)
0.54425343582683983
>>> random.normal(0.5, 0.1, 10)
array([ 0.41583017,  0.47527431,  0.62681263,  0.52641006,  0.55507713,
        0.64093938,  0.46411351,  0.61719065,  0.55209002,  0.5232665 ])
```

Code 2-26

Another useful function is **random.shuffle**, which rearranges items in an array in a random order. In Code 2-27 an array that can consist of any type of items is created and shuffled.

© Jones and Bartlett Publishers, LLC. NOT FOR SALE OR DISTRIBUTION

```
>>> a = [1,2,3,4,5]
>>> random.shuffle( a )
>>> a
[4, 3, 2, 5, 1]
>>> a = ['t', (3,4), 45, 6.7]
>>> random.shuffle( a )
>>> a
[(3, 4), 45, 6.7, 't']
```

Code 2-27

2.5.6 Statistical Methods

Arrays are *objects* and therefore contain functions. Code 2-28 shows how to extract information about an array. Without any argument, these functions will compute the values for the entire array. An integer argument determines over which axis the functions are computed. In a two-dimensional array the axes are vertical and horizontal. If the argument to a function is 0, then the function is performed over the vertical axis; if the argument is 1, it is performed over the horizontal axis. Arrays with higher dimensions can receive arguments with larger integer values.

```
>>> d = random.ranf( (4,3) )
>>> d.max( )  # max value in entire array
>>> d.max( 0 ) # max values in each column
>>> d.max( 1 ) # max values in each row
>>> d.min( )   # min value in entire array
>>> d.sum( )   # sum of entire array
>>> d.sum( 0 ) # sum of columns
>>> d.mean( 1 ) # average of rows
>>> d.std( )  # standard deviation of entire array
```

Code 2-28

2.6 Thinking About Problems

It is possible to use *for* loops to perform the addition of two arrays, as shown in Code 2-29. Note that the use of a direct command (>>> d = a + b) is much faster than using loops.

```
>>> d = zeros( 3, float )
>>> for i in range( 3 ):
d[i] = a[i] + b[i]

>>> d
array([ 0.1,  5. ,  8.3])
```

Code 2-29

Python is an interpreted language, which means that each step of the program is first converted to machine language and then executed. Thus, a command inside of a *for* loop will be interpreted several times, which is time-consuming. When a command such as *c=a+b* can be used to perform the same function, a significant amount of time

© Jones and Bartlett Publishers, LLC. NOT FOR SALE OR DISTRIBUTION

will be saved. In actuality there is still a *for* loop that is performing all of the additions necessary to compute $c=a+b$, but these are performed "beneath the Python" in the compiled language in which Python was written. Functions such as **dot** and **add.outer** are much faster than writing the steps out, as shown in Code 2-29.

Programmers who have experience in C, Fortran, and similar languages are used to thinking in terms of *for* loops in order to solve problems. In order to make Python efficient, this thinking needs to change a little. Problems with data arrays need to be conceived in a more parallel fashion in which operations are performed over the entire array rather than for individual elements. The basic rule is to avoid *for* loops. If a program is being developed that contains nested *for* loops, then there is a good chance that it can be rewritten using faster and more efficient array commands.

2.7 Array Conversions

Arrays often need to be converted to data types such as lists and strings. The **array** command has already been used to convert a list to an array (see the first line in Code 2-1). Code 2-30 shows the **tolist** command, which converts an array to a list.

```
>>> a = array([ 0., 0.25, 0.5, -5., 1., 1.25, -1., -4., 2.,  -2., 2.5, 2.75,3.,
3.25,-3.])
>>> a.tolist()
[0.0, 0.25, 0.5, -5.0, 1.0, 1.25, -1.0, -4.0, 2.0, -2.0, 2.5, 2.75, 3.0, 3.25,
-3.0]
```

Code 2-30

Conversion from an array to a string is a little more complicated. The command **str** will convert the characters from a **print** statement into a string, as shown in Code 2-31. However, this includes newline characters. To convert the elements of an array into a string, each element must be sent to the **str** function, which can easily be accomplished by employing the **map** function.

```
>>> str( a )
'[ 0.    0.25  0.5  -5.    1.    1.25 -1.   -4.    2.   -2.    2.5   2.75\n  3.
3.25 -3.  ]'
>>> map( str, a )
['0.0', '0.25', '0.5', '-5.0', '1.0', '1.25', '-1.0', '-4.0', '2.0', '-2.0',
'2.5', '2.75', '3.0', '3.25', '-3.0']
```

Code 2-31

Another type of string that can be used is a binary string in which the elements of the array are converted to their binary equivalents. Consider an array of integers. A 32-bit processor will encode an integer in four bytes. When applied to an array of integers, the command **tostring** will convert each integer to its four-byte representation. Code 2-32 displays this conversion. The first four bytes (shown as \x00 \x00 \x00 \x00) are the bytes that encode the first integer (which is 0) in the array *a*.

© Jones and Bartlett Publishers, LLC. NOT FOR SALE OR DISTRIBUTION

```
>>> a = arange( 4 )
>>> a
array([0, 1, 2, 3])
>>> b = a.tostring()
>>> b
'\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
```

Code 2-32

Conversion from a byte string is performed by **fromstring**, which may need an argument to indicate what type of conversion is taking place. The first command in Code 2-33 converts the byte-string to integers (which is the default conversion). The second command converts the same byte string to short integers that use only two bytes instead of four. The third conversion produces two floats (eight bytes apiece).

```
>>> fromstring( b )
array([0, 1, 2, 3])
>>> fromstring( b, Int8 )
array([0, 0, 0, 0, 1, 0, 0, 0, 2, 0, 0, 0, 3, 0, 0, 0], dtype=int8)
>>> fromstring( b, float )
array([  2.12199579e-314,   6.36598737e-314])
```

Code 2-33

These conversions can be very useful when reading and writing to binary files. Some bioinformatics applications use files produced by machines. The data files may contain data stored in binary format that needs to be converted to arrays for processing in Python. The **fromstring** command will convert data into an array in a single command. In the early days of DNA sequencing, many of the sequencing machines were attached to MACs rather than PCs. The problem for PC users was that MACs stored information differently than PCs and so a conversion was necessary to read the data properly.

A MAC stores information in a *big endian* format whereas a PC uses the *little endian* format. Consider a two-byte (short) integer. The least significant byte can store numbers from −128 to +127. Any integer bigger than that uses the most significant byte. The big endian format will store the most significant byte first, whereas the little endian format will store the least significant byte first.

Code 2-34 shows an example that begins with array *a*, which contains four elements that are two bytes each (shorts). The byte string for this is printed for a PC. Now consider a case in which the same data is stored on a MAC. String *c* is the byte string in this example. When this data is read in on a PC, the conversion will be incorrect (the first **fromstring** command). The **byteswap** command will reverse the bytes and produce the correct data.

```
>>> a = array([0, 1, 2, 3]).astype(short)
>>> a.tostring()
'\x00\x00\x01\x00\x02\x00\x03\x00'

>>> c = '\x00\x00\x00\x01\x00\x02\x00\x03'
```

© Jones and Bartlett Publishers, LLC. NOT FOR SALE OR DISTRIBUTION


```
>>> fromstring( c )
array([16777216, 50332160])
>>> fromstring(c,short).byteswap()
array([0, 1, 2, 3], dtype=int16)
```

Code 2-34

2.8 SciPy

SciPy is a large library of scientific routines for Fourier transforms. Solving equations, integrations, differential equations, wavelets, and so on are functions available in this package. Certainly, before creating a scientific program it would be prudent to see if SciPy contains the needed routines. In this book only the Fourier transform module will be used.

2.9 Summary

There are many third-party modules available for Python. This chapter reviewed the NumPy and SciPy packages, which provide a plethora of vector and matrix tools. SciPy is a large package that provides many scientific functions. This text uses only a small fraction of these tools, and readers are encouraged to explore their offerings.

Bibliography

SciPy. Retrieved from http://scipy.org.

Problems

1. What is the difference between the **range** and **arange** functions?
2. Using the **numpy.random** module, create a vector of random numbers.
3. Without using the **for** or **while** commands, compute the average of a vector of 1 million random numbers.
4. Compute the sum of the columns of a large matrix of random numbers.
5. Create a vector of random numbers having a range of -1 to $+1$.
6. Create a matrix of random numbers. Multiply this matrix by scalar in a single Python command.
7. Create a 5×3 matrix of random numbers. Create a vector of three random numbers. What happens when you run *matrix * vector*?
8. Compute the transpose of a matrix.
9. Create a random DNA string and write a Python script to convert the letters to an array of numbers (A=1, C=2, G=3, and T=4). Do not use loops (*for, while*) to accomplish this conversion. (This can be accomplished in five commands, including the creation of the DNA string.)

© Jones and Bartlett Publishers, LLC. NOT FOR SALE OR DISTRIBUTION