2
Top-level SciPy

At the top level, SciPy is basically NumPy, since both the object creation and basic manipulation of these objects are performed by functions of the latter library. This assures much faster computations, since the memory handling is done internally in an optimal way. For instance, if an operation must be made on the elements of a big multidimensional array, a novice user might be tempted to go over columns and rows with as many `for` loops as necessary. Loops run much faster when they access each consecutive element in the same order in which they are stored in memory. We should not be bothered with considerations of this kind when coding. The NumPy/SciPy operations assure that this is the case. As an added advantage, the names of operations in NumPy/SciPy are intuitive and self explanatory. Code written in this fashion is extremely easy to understand and maintain; faster to correct or change in case of need. Let us illustrate this point with one introductory example.

The `scipy.misc` library contains a classical image used in the image processing community for testing and comparison purposes – `scipy.misc.lena`. This is the name given to a 512 x 512 pixel standard test image, which has been in use since 1973, and was originally cropped from the centerfold of November 1972 issue of Playboy magazine. It is a picture of Lena Söderberg, a Swedish model, shot by photographer Dwight Hooker. The image is probably the most widely used test image for all sorts of image processing algorithms (such as compression and noise reduction) and related scientific publications.

This image is stored as a two-dimensional array. The n^{th} column and m^{th} row entry of this array is a number that measures the grayscale value at the pixel in position ($n+1$, $m+1$) of the image. We access these numerical contents and store them in the `img` variable, by issuing the following command:


```
>>>img=scipy.misc.lena()
```

We may peek on some of these values, say the 7 x 3 upper corner of the image (7 columns, 3 rows). Instead of issuing a couple of `for` loops, we *slice* the corresponding portion of the image. The `img[0:3,0:7]` command gives us the following:

```
array([[162, 162, 162, 161, 162, 157, 163],
       [162, 162, 162, 161, 162, 157, 163],
       [162, 162, 162, 161, 162, 157, 163]])
```

We can use the same strategy to populate arrays, or change some of their values. For instance, in the next session, we change all entries of the second row of the previous array, between rows 2 and 6, to hold zeros, as follows:

```
>>>img[1,1:6]=0
>>> print img[0:3,0:7]
[[162 162 162 161 162 157 163]
 [162   0   0   0   0   0 163]
 [162 162 162 161 162 157 163]]
```


Object essentials

We have been introduced to the basic object – the multidimensional array (which in NumPy jargon is referred to as `ndarray`). All elements of the array are casted to the same datatype. We obtain this datatype by issuing the `dtype` command. We are able to access the value of any of its elements, as well as its dimension (shape), size, and many other properties of the array. The following session illustrates how to obtain some of that information:

```
>>>img.dtype, img.shape, img.size
```

```
(dtype('int64'), (512, 512), 262144)
>>>img[32,67]
87
```

Let us interpret the outputs. The entries of `img` are 64-bit integer values (`'int64'`). This is essentially different on different systems, and depends on both the Python installation and our computer specifications. The shape of the array (note it comes as a Python tuple) is 512 x 512, and consequently it has 262144 entries. The grayscale value of the image at the 33rd column and 68th row is 87 (note that in NumPy, as in Python or C, all indices are zero based).

We will now introduce the basic property and methods of NumPy/SciPy objects – datatype and indexing.

Datatype

There are several formulae to impose the datatype. For instance, if we want all entries of an already-created array to be 32-bit floating point values, we may cast it as follows:

```
>>> img=scipy.misc.lena().astype('float32')
```

A second way is done by using the optional argument, `dtype=` on any array creation command:

```
>>> scores = numpy.array([101,103,84], dtype='float32')
```

This can be simplified even further with a third clever method (although this practice offers codes that are not so easy to interpret):

```
>>> scores = numpy.float32([101,103,84])
array([ 101.,  103.,   84.], dtype=float32)
```


The choice of datatypes for NumPy arrays is extremely flexible; we may choose the basic Python types (including `bool`, `dict`, `list`, `set`, `tuple`, `str`, and `unicode`), although for numerical computations we mainly focus on `int`, `float`, `long`, and `complex`.

NumPy has its own datatypes optimized for using them with `ndarray` instances, with the same precision as the previously given native types. We distinguish them with a trailing underscore (_) after the name. For instance, `ndarray` of strings could be initialized, as follows:

```
>>> a=numpy.array(['Cleese', 'Idle', 'Gilliam'], dtype='str_')
>>>a.dtype
dtype('|S7')
```

Note two things; unlike its purely Python counterpart, the usage of the `'str_'` datatype requires the name to be quoted. We could use the longer unquoted version, `numpy.str_`, instead. Also, when prompted for datatype, the system returns its C-derived equivalent name instead; `'|S7'` (`'|S` for strings, and `7'` to indicate the largest size of any of its elements).

The most common way to address the usual numerical types is with the bit width nomenclature – `boolXX`, `intXX`, `uintXX`, `floatXX`, or `complexXX`, where `XX` indicates the bit size (for example, `uint32` for 32-bit unsigned integers).

It is also possible to design our own datatypes, and this is where the full potential of the flexibility of NumPy datatypes arise. For instance, a datatype to indicate the name and grades of a student could be created, as follows:


```
>>> dt=numpy.dtype([ ('name', numpy.str_, 16), 'grades', numpy.float64,
(2,)) ])
```

This means that the `dt` datatype has two parts – the first part is a name, that must be a 16 characters, `numpy.str_ string`. The second part, the grades, is a subarray of dimension 2 with scores as 64-bit floating point values. A valid array with elements in this datatype would then look like the following:

```
>>> MA141 = numpy.array([ ('Cleese', (7.0,8.0)), ('Gilliam', (9.0,10.0))
], dtype=dt)
```

Indexing

There are two basic methods to access the data in a NumPy array `A`, both of them with the same syntax, `A[obj]`, where `obj` is a Python object that performs the selection. We are already familiar with the basic method of record access for a single entry. The second method is the objective of this subsection, slicing. This concept is what makes NumPy and SciPy so incredibly easy to manage.

The basic slice is a Python object of the form `slice(start,stop,step)`, or in a more compact notation, `start:stop:step`. Initially, the three variables `start`, `stop`, and `step` are non-negative integer values, with `start` less than or equal to `stop`. This represents the sequence of indices *start + (k * step)*, for indices `k` from 0 to the largest integer smaller or equal to the value given by *(stop - start) / step*. When a slice is placed on any of the dimensions of `ndarray`, it selects all entries in that dimension indexed by the corresponding sequence of indices. The simple examples given next illustrate this point:

```
>>> A=numpy.array([[1,2,3,4,5,6,7,8],[2,4,6,8,10,12,14,16]])
>>> print A[0:2, 0:8:2]
```

```
[[ 1  3  5  7]
 [ 2  6 10 14]]
```

If `start` is greater than `stop`, a negative value of `step` is used to traverse the sequence backwards.

```
>>> print A[0:2, 8:0:-2]
[[ 8,  6,  4,  2]
 [16, 12,  8,  4]]
```

Negative values of `start` and `stop` are interpreted as `n-start` and `n-stop` (respectively), where `n` is the size of the corresponding dimension. The `A[0:2,-1:0:-2]` command gives exactly the same output as the previous example.

The slice objects can be shortened by absence of start (which implies a zero if `step` is positive, or the size of the dimension if `step` is negative), absence of `stop` (which implies the size of the corresponding dimension in case of positive `step`, or zero in case of negative `step`). Absence of `step` implies `step` is equal to 1. The `::` object can be shortened simply as `:`, for an easier syntax. The `A[:,::-2]` command then offers yet again the same output as the previous two.

The first nonbasic method of accessing data from an array is based on the idea of collecting several indices, and requesting the elements in array with those indices. For example, from our previous array `A` we would like to construct a new array with the elements on locations (0, 0), (0, 3), (1, 2), and (1, 5). We do so by gathering the x and y values of the indices in respective lists – `[0,0,1,1]`, `[0,3,2,5]`, and feeding these lists to `A` as an indexing object, as follows:

```
>>> print A[ [0,0,1,1], [0,3,2,5] ]
[ 1  4  6 12]
```

Note how the result loses the dimension of the primitive array, and offers a one-dimensional array. If we desire to capture a subarray of A with indices in the Cartesian product of two sets of indices, respecting the row and column choice and creating a new array with the dimensions of the Cartesian product, we use the comfortable `ix_` command. For instance, if in our previous array we would like to obtain the subarray of dimension 2 x 2 with indices in the Cartesian product of indices (0, 1) by (0,3) (these are the locations (0, 0), (0, 3), (1, 0), and (1, 3)), we do so as follows:


```
>>> print A[ numpy.ix_( [0,1], [0,3] )]
[[1 4]
 [2 8]]
```

The array object

At this point we are ready for a thorough study of all interesting attributes of `ndarray` for Scientific computing purposes. We have already covered a few, such as `dtype`, `shape`, and `size`. Other useful attributes are `ndim` (to compute the number of dimensions in the array), `real` and `imag` (to obtain the real and imaginary parts of the data, should this be formed by complex numbers), or `flat` (which creates a one-dimensional indexable iterator from the data).

For instance, if we desired to add all the values of an array together, we could use the `flat` attribute to run over all the elements sequentially, and accumulate all the values in a variable. A possible code to perform this task should look like the following code snippet (compare this code with the `ndarray.sum()` method explained in object calculation ahead):


```
>>> value=0; img=scipy.misc.lena()
>>> for item in img.flat: value+=item
>>> value
32518120
```

We have also explored some of the methods applied to arrays. These are the tools used to modify these objects; let it be their datatypes, their shape, or converting them to a different structure. We classify these methods in three big categories – array conversion, shape selection/manipulation, and object calculation.

Array conversion is used to cast data to different types (`astype`), copy arrays to store them under another variables (`copy`), fill whole arrays with scalar values (`fill`), or dump the array to a file, list, or string (`tofile`, `tolist`, `tostring`).

For instance, to write the contents of the `img` array to a text file, making sure that each entry of the array is printed as an integer, and that every two integers are separated by a white space, we could issue the following command:


```
>>> img.tofile("lena.txt",sep=" ",format="%i")
```

Note how the formatting string follows C conventions.

Shape selection/manipulation is usually employed when we require some kind of rearranging (`swapaxes`, `transpose`), including sorting (`argsort`, `sort`). We also use these methods when we need reshaping (`reshape`), resizing (`flatten`, `ravel`, `resize`, `squeeze`) or selecting (`choose`, `compress`, `diagonal`, `nonzero`, `searchsorted`, `take`). These methods are very powerful when used in cooperation with slicing operations; as a matter of fact, many of them can be used instead of slicing to offer our users more readable code.

We need to say a word about the differences between `flat`, `ravel`, and `flatten`, which offer very similar outputs, since they make a huge difference of usage in terms of memory management. The first one, `flat`, creates an iterator to the elements of the array. Once used, it disappears from memory. The second one, `ravel`, returns a view of the one-dimensional flattened array when it can, and copies of it when requested. The last one, `flatten`, creates a copy of the flattened one-dimensional array, and always allocates memory for it. We use it only when we need to change the values of flattened arrays.

Notice also the power of the sorting methods in the session given next. We create an array of integers. If these values were sorted, what would be the order of their indices? We may obtain this information with the `argsort` method. We may even impose the sorting algorithm to be used (rather than coding it ourselves) – `quicksort`, `mergesort`, or `heapsort`. We can even sort the array in place, using the `sort` method, as follows:


```
>>> A=numpy.array([11,13,15,17,19,18,16,14,12,10])
>>>A.argsort(kind='mergesort')
array([9, 0, 8, 1, 7, 2, 6, 3, 5, 4])
>>>A.sort()
>>> print A
[10 11 12 13 14 15 16 17 18 19]
```

Array calculation methods are used to perform computations or extract information about our data. We have a set of methods of statistical nature that help us compute, for instance, maximum or minimum values of the data (`max`, `min`), as well as their corresponding indices (`argmax`, `argmin`). We have methods to compute the sum, cumulative sums, product, or cumulative products (`sum`, `cumsum`, `prod`, `cumprod`). It is possible to extract the average (`mean`), point spread (`ptp`), variance (`var`), or standard deviation (`std`). Further nonstatistical calculation methods allow us to compute complex conjugate of complex-valued arrays (`conj`), the trace of the array (`trace`, the sum of the elements in the diagonal), or even clipping the matrix (`clip`) by forcing a minimum and maximum value below and above certain thresholds.

Note how most of these methods can act on the whole array, or over each of its dimensions:

```
>>> A=numpy.array([[1,1,1],[2,2,2],[3,3,3]])
>>>A.mean()
2
>>>A.mean(axis=0)
array([ 2.,  2.,  2.])
>>>A.mean(axis=1)
array([ 1.,  2.,  3.])
```

Let us also illustrate the `clip` command with an easy exercise based on the Lena image.

Compute the maximum and minimum values of Lena (`img`), and contrast them with the point spread (it should be equal to the difference between those two values). Create a new array `A` by clipping Lena so that the minimum is maintained, but the point spread is reduced to only 100 values.

```
>>>img.min(), img.max(), img.ptp()
(25, 245, 220)
>>> A=img.clip(img.min(),img.min()+100)
>>>A.min(), A.max(), A.ptp()
(25, 125, 100)
```


Array routines

In this section we will deal with most operations with arrays. We will classify them in four main categories, as follows:

- Routines for the creation of new arrays
- Routines for the manipulation of a single array
- Routines for the combination of two or more arrays
- Routines to extract information from arrays

The reader will surely realize that some operations of this kind can be carried out by methods, which once again shows the flexibility of Python and NumPy.

Routines for array creation

We have seen the basic command that brings an array to memory and stores it to a variable – `A=numpy.array([[1,2],[2,1]])`. The complete syntax is as follows:

```
array(object=,dtype=None,copy=True,order=None,subox=False,ndim=0)
```

Let us go over the options; `object` is simply the data we use to initialize the array. In the previous example, that object is a small 2 x 2 square matrix; we may impose a determinate datatype with the `dtype` option. The result is stored in the variable `A`; if `copy` is false, the returned object will be a copy of the array only if `dtype` is not equivalent to the datatype of `object`. The arrays are stored following a C-style ordering of rows and columns. If the user prefers to store the array following the memory style of Fortran, the `order='Fortran'` option should be used. The `subok` option is very subtle; if true, the array may be passed as a subclass of the object.

If false, then only `ndarray` arrays are passed. And finally, the `ndim` option indicates the smallest dimension returned by the array. If not offered, this is computed from `object`.

A set of special arrays can be obtained with the commands such as `zeros`, `ones`, `identity`, and `eye`. The names of these commands are quite informative, as mentioned next:

- `zeros` creates an array filled with zeros
- `ones` creates an array filled with ones
- The `identity` command creates a square matrix with dimension indicated by a single positive integer n. The entries are filled with zeros, except along the main diagonal ((k, k) for k from 0 to n-1), which is filled with ones.
- Very similar to `identity` is the `eye` command, which also constructs diagonal arrays. Unlike `identity`, `eye` allows specifying diagonals off the main one, and nonsquare arrays.

```
>>> Z=numpy.zeros((5,5), dtype=int)
>>> U=numpy.ones((2,2), dtype=int)
>>> I=numpy.identity(3, dtype=int)
```

In the first two cases, we indicated the shape of the array (as a Python tuple of positive integers), and the optional datatype imposition.

The syntax for `eye` is as follows:

```
numpy.eye(N,M=None,k=0,dtype=float)
```

The integers, `N` and `M` indicate the shape of the array, and the integer `k` indicates the index of the diagonal to populate. An index `k=0` (the default) points to the main diagonal, a positive index refers to upper diagonals, and negative value refer to lower diagonals.


```
>>> D=numpy.eye(4,k=1) + numpy.eye(4,k=-1)
>>> print D
[[ 0.  1.  0.  0.]
 [ 1.  0.  1.  0.]
 [ 0.  1.  0.  1.]
 [ 0.  0.  1.  0.]]
```

With the aid of only the previous four commands and basic slicing, it is possible to create more complex arrays in simple ways. We propose the following challenge.

Use exclusively the previous definitions of `U` and `I`, together with an `eye` array. How would the reader create a 5 x 5 array `A` of floating values with "fives" at the four entries (0, 0), (0, 1), (1, 0), (1, 1); "sixes" along the remaining entries of the diagonal; and "threes" in the two other corners?

```
>>> A=3.0*(numpy.eye(5,k=4) + numpy.eye(5,k=-4))
>>> A[0:2,0:2]=5*U; A[2:5,2:5]=6*I
>>> print A
[[ 5.  5.  0.  0.  3.]
 [ 5.  5.  0.  0.  0.]
 [ 0.  0.  6.  0.  0.]
 [ 0.  0.  0.  6.  0.]
 [ 3.  0.  0.  0.  6.]]
```


The flexibility of array creation in NumPy is even more apparent with the `fromfunction` command. For instance, if we require a 4 x 4 array where each entry reflects the product of its indices, we use the lambda function, `(lambda i,j: i*j)` in the `fromfunction` command, as follows:

```
>>> B=numpy.fromfunction( (lambda i,j: i*j), (4,4), dtype=int)
>>> print B
[[0 0 0 0]
 [0 1 2 3]
 [0 2 4 6]
 [0 3 6 9]]
```

Of great importance are the array creation commands that deal with the concept of masking. This is one of the most reliable methods to manipulate large arrays of data, and it is based on the idea of gathering those indices for which their corresponding entries satisfy a given condition. For example, in the array `B` shown in the preceding code snippet, we can mask all zero-valued entries with the `B==0` command, as follows:


```
>>> print B==0
[[ True  True  True  True]
 [ True False False False]
 [ True False False False]
 [ True False False False]]
```

How would the reader update `B` so that those zero entries can be replaced by the sum of the squares of their corresponding indices?

Multiplying a mask by a second array of the same shape offers a new array in which each entry is either zero (if the corresponding entry in the mask is false) or the entry of the second array (if the corresponding entry in the mask is true).

```
>>> B += numpy.fromfunction((lambda i,j:i*i+j*j), (4,4))*(B==0)
>>> print B
[[0 1 4 9]
 [1 1 2 3]
 [4 2 4 6]
 [9 3 6 9]]
```

But note that, in this process, we have created in each step a new array in memory with as many Boolean values as the size of the original array. In these toy examples it is not a big deal. But when handling large datasets, allocating too much memory could seriously slow down our computations and exhaust the memory of our system. Among the creation commands presented in the table, there are two in particular, such as `putmask` and `where`, which facilitate the management of resources internally, thus speeding up the process.

Note, for example, when we look for all odd-valued entries in `B`, the resulting mask has size of 16, although the interesting entries are only eight.

```
>>> print B%2!=0
[[False  True False  True]
 [ True  True False  True]
 [False False False False]
 [ True  True False  True]]
```

The `numpy.where()` command helps us gather precisely those entries in a more efficient way.

```
>>>numpy.where(B%2!=0)
(array([0, 0, 1, 1, 1, 3, 3, 3]), array([1, 3, 0, 1, 3, 0, 1, 3]))
```

If we desire to change those odd entries to, say their squares plus one, we can use the `numpy.putmask()` command instead, for a better management of memory.

```
>>>numpy.putmask( B, B%2!=0, B^2+1)
>>> print B
[[ 0  2  4 82]
 [ 2  2  2 10]
 [ 4  2  4  6]
 [82 10  6 82]]
```

Note how the `putmask` procedure does update the values of `B`, without the explicit need to make an assignment.

There are three more interesting commands that create arrays in the form of meshes. The `arange` and `linspace` commands create uniformly spaced values between two numbers. In `arange` we specify the spacing between elements; in `linspace` we specify the desired number of elements in the mesh. The `logspace` command creates uniformly spaced values in a logarithmic scale between the logarithm of two numbers to the base 10. The user could think of these outputs as the support of univariate functions.

```
>>> L1=numpy.arange(-1,1,0.3)
>>> print L1
[-1.  -0.7 -0.4 -0.1  0.2  0.5  0.8]
>>>L2=numpy.linspace(-1,1,4)
>>> print L2
[-1.          -0.33333333  0.33333333  1.        ]
>>>L3= numpy.logspace(-1,1,4)
>>> print L3
[  0.1           0.46415888   2.15443469  10.        ]
>>> L3
```

Finally, `meshgrid`, `mgrid`, and `ogrid` create two two-dimensional arrays of dimensions $n \times m$, containing the elements of two given one-dimensional arrays of dimensions n and m. It accomplished this by repeating the values of each array as necessary. The user could think of these outputs as the support of functions of two variables.

The first of these routines, `meshgrid`, accepts only arrays as input. The other two routines, `mgrid` and `ogrid`, accept only indexing objects (for example, slices). The difference between these last two is a matter of memory allocation; while `mgrid` allocates full arrays with all the data, `ogrid` only creates enough sets so that the corresponding `mgrid` command could be obtained by a proper Cartesian product, as follows:

```
>>> print numpy.meshgrid(L2,L3)
(array([[-1.        , -0.33333333,  0.33333333,  1.        ],
       [-1.        , -0.33333333,  0.33333333,  1.        ],
       [-1.        , -0.33333333,  0.33333333,  1.        ],
       [-1.        , -0.33333333,  0.33333333,  1.        ]]), array([[
0.1       ,  0.1       ,  0.1       ,  0.1       ],
```

```
       [  0.46415888,   0.46415888,   0.46415888,   0.46415888],
       [  2.15443469,   2.15443469,   2.15443469,   2.15443469],
       [ 10.        ,  10.        ,  10.        ,  10.        ]]))
>>> print numpy.mgrid[0:5,0:5]
[[[0 0 0 0 0]
  [1 1 1 1 1]
  [2 2 2 2 2]
  [3 3 3 3 3]
  [4 4 4 4 4]]

 [[0 1 2 3 4]
  [0 1 2 3 4]
  [0 1 2 3 4]
  [0 1 2 3 4]
  [0 1 2 3 4]]]
>>> print numpy.ogrid[0:5,0:5]
[array([[0],
       [1],
       [2],
       [3],
       [4]]), array([[0, 1, 2, 3, 4]])]
```


We would like to finish the subsection on array creation by showing one of the most useful routines for image processing and differential equations – the `tile` command. Its syntax is very simple, and is shown as follows:

```
tile(A, reps)
```


This routine presents a very effective way of tiling an array `A` following some repetition pattern `reps` (a tuple, a list, or another array) to create larger arrays. The following checkerboards exercise shows its potential.

Start with two small binary arrays – `B=numpy.ones((3,3))` and `checker2by2=numpy.zeros((6,6))`, and create a checkerboard using `tile` and as few operations as possible.

The following is a possible solution:

```
>>>checker2by2[0:3,0:3]=checker2by2[3:6,3:6]=B
>>> numpy.tile(checker2by2,(4,4))
```

Routines for the combination of two or more arrays

On occasion we need to combine the data of two or more arrays together to solve a specific problem. The core NumPy libraries contain extremely efficient routines to carry out these computations, and we urge the reader to get familiar with them. They are constructed with state-of-the-art algorithms, and they make sure that usage of memory is minimum and complexity is optimal. The most relevant in this set of routines are those that operate on arrays as if they were matrices. We then have matrix products (`outer`, `inner`, `dot`, `vdot`, `tensordot`, `cross`, and `kron`), array correlations (`correlate`, `convolve`), array stacking (`concatenate`, `vstack`, `hstack`, `column_stack`, `row_stack`, and `dstack`), and array comparison (`allclose`).

The reader versed in linear algebra will surely enjoy the matrix products included in NumPy. We postpone their usage and analysis until we cover the SciPy module on linear algebra in *Chapter 3, SciPy for Linear Algebra*.

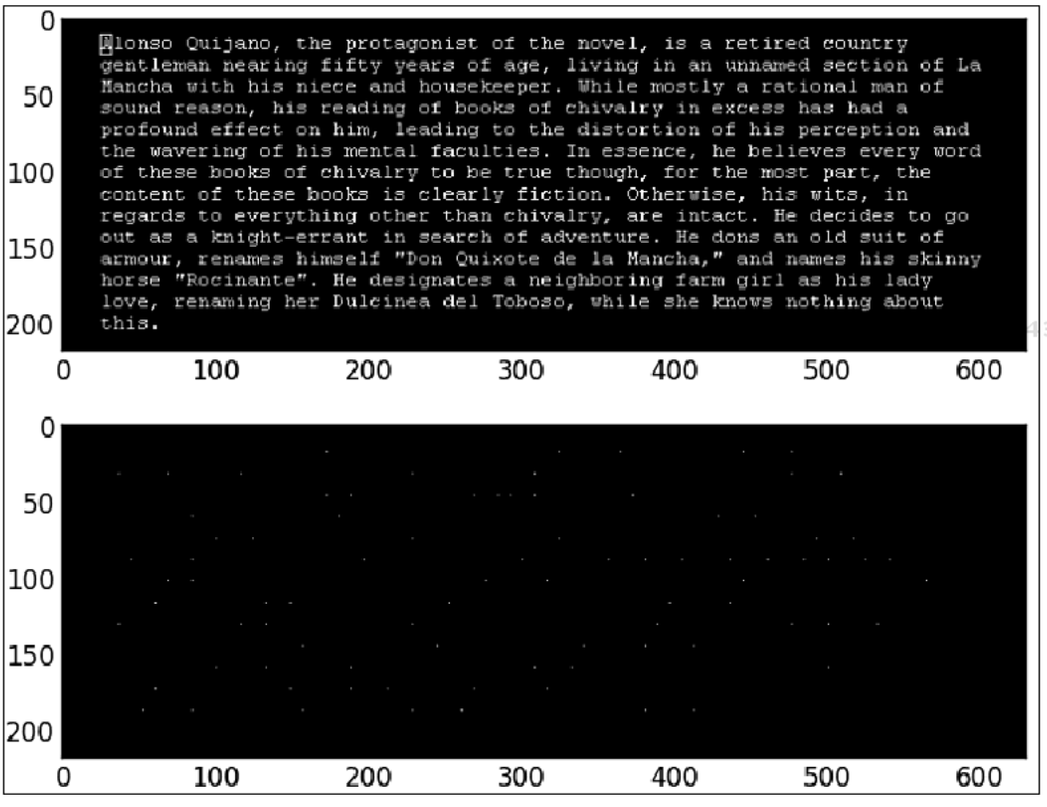
An excellent use for correlation of arrays is, for example, for basic pattern matching. For instance, the image in the following example represents a binary array (it contains only ones and zeros). We visualize it by assigning to each location in the array a white pixel if the corresponding value is one, and a black pixel to zero values. The first array, `text`, contains an image of a paragraph extracted from the wikipedia page on Don Quixote, while a second array, `letterE`, contains an image of the letter "e". This `letterE` array is actually a subarray of dimension 6 x 6 obtained from the `text` array:

```
>>>letterE=text[14:20,169:175]
```

The maximum value of the correlation of both arrays offers the location of all the "e" letters contained in the array text:

```
>>> print letterE
[[0 1 1 1 1 0]
 [1 0 0 0 0 1]
 [1 1 1 1 1 1]
 [1 0 0 0 0 0]
 [1 0 0 0 0 0]
 [0 1 1 1 1 1]]
>>>corr = scipy.ndimage.correlate(text,letterE)
>>> eLocations = (corr == corr.max())
```

This results in the following screenshot:



A few words about stacking operations; we have a basic concatenation routine, `concatenate`, which joins a sequence of arrays together along a pre-determined axis. Of course, all arrays in the sequence must have the same dimensions, otherwise it doesn't work. The rest of the stack operations are syntactic sugar for special cases of `concatenate` – `vstack` to glue arrays vertically, `hstack` to glue arrays horizontally, `dstack` to glue arrays in the third dimension, and so on.

Another impressive set of routines for array combination are the set operations. They allow the user to handle one-dimensional arrays as if they were sets, and perform with easiness, the Boolean operations of intersection (`intersect1d`), union (`union1d`), set difference (`setdiff1d`), or set exclusive or (`setxor1d`). The results of any of these set operations on arrays always return sorted arrays. It is also possible to test whether all the elements in one array belong to a second array (`in1d`).

Routines for array manipulation

There is a sequence of splitting routines, designed to break up arrays into smaller arrays, in any given dimension – `array_split`, `split` (both basic splitting in the indicated axis), `hsplit` (horizontal split), `vsplit` (vertical split), and `dsplit` (in the third axis). Let us illustrate these with a simple example:

```
>>> print checker2by2
[[ 1.  1.  1.  0.  0.  0.]
 [ 1.  1.  1.  0.  0.  0.]
 [ 1.  1.  1.  0.  0.  0.]
 [ 0.  0.  0.  1.  1.  1.]
 [ 0.  0.  0.  1.  1.  1.]
 [ 0.  0.  0.  1.  1.  1.]]
>>>numpy.vsplit(checker2by2,3)
[array([[ 1.,  1.,  1.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  0.,  0.,  0.]]),
 array([[ 1.,  1.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  1.,  1.]]),
 array([[ 0.,  0.,  0.,  1.,  1.,  1.],
       [ 0.,  0.,  0.,  1.,  1.,  1.]])]
```


The behavior of a Python function on an array is *usually* the application of the function to each of the elements of the array. Note for example how the NumPy function `sin` works on an array:

```
>>> a=numpy.array([-numpy.pi, numpy.pi])
>>> print numpy.vstack((a, numpy.sin(a)))
[[ -3.14159265e+00   3.14159265e+00]
 [ -1.22464680e-16   1.22464680e-16]]
```


This happens provided the function has been properly vectorized (which is the case with `numpy.sin`). Notice the behavior with nonvectorized Python functions. Let us define one that computes, for each value of `x`, the maximum between `x` and 100 without using any routine from the NumPy libraries.

```
# function max100
defmax100(x):
    return max(x,100)
```

If we try to apply this function to the preceding array, the system raises an error, as follows:

```
>>> max100(a)
ValueError: The truth value of an array with more than one element is
ambiguous. Use a.any() or a.all()
```

We need to explicitly indicate to the system when we desire to apply one of our functions to arrays, as well as scalars. We do that with the `vectorize` routine, as follows:

```
>>>numpy.vectorize(max100)(a)
array([100, 100])
```


For our benefit, the NumPy libraries provide a great deal of already-vectorized mathematical functions. Some examples are `round_`, `fix` (to round the elements of an array to a desired number of decimal places), `angle` (to provide the angle of the elements of an array, provided they are complex numbers), any basic trigonometric (`sin`, `cos`, `tan`, `sic`), exponential (`exp`, `exp2`, `sinh`, `cosh`), and logarithmic functions (`log`, `log10`, `log2`).

We also have mathematical functions that treat the array as output of multidimensional functions, and offer relevant computations. Some useful examples are `diff` (to emulate differentiation along any specified dimension, by performing discrete differences), `gradient` (to compute the gradient of the corresponding function), or `cov` (for the covariance of the array). Sorting the whole array according to the values of the first axis is also possible with the `msort` and `sort_complex` routines.

Routines to extract information from arrays

Most of the routines to extract information are statistical in nature, which include `average` (which acts exactly as the `mean` method), `median` (to compute the statistical median of the array on any of its dimensions, or the array as a whole), and computation of histograms (`histogram`, `histogram2d`, and `histogramdd`, depending on the dimensions of the array). The other important set of routines in this category deal with the concept of bins for arrays of dimension one. This is more easily explained by means of examples. Take the array `A=numpy.array([5,1,1,2,1,1,2,2,10,3,3,4,5])`. The `unique` command finds the different values in any array, and presents them as sorted:


```
>>>numpy.unique(A)
array([ 1, 2, 3, 4, 5, 10])
```

For arrays such as `A`, in which all the entries are nonnegative integers, we can visualize the array `A` as a sequence of eleven bins labeled with numbers from 0 to 10 (the maximum value in the array). Each bin with label n contains the number of n's in the array:

```
>>>numpy.bincount(A)
array([0, 4, 3, 2, 1, 2, 0, 0, 0, 0, 1])
```

For arrays where some of the elements are not numbers (`nan`), NumPy has a set of routines that mimic methods to extract information, but disregard the conflicting elements – `nanmax`, `nanmin`, `nanargmax`, `nanargmin`, `nansum`, and so on.

```
>>> A=numpy.fromfunction((lambda i,j: (i+1)*(-1)**(i*j)),(4,4))
>>> print A
[[ 1.  1.  1.  1.]
 [ 2. -2.  2. -2.]
 [ 3.  3.  3.  3.]
 [ 4. -4.  4. -4.]]
>>> B=numpy.log2(A)
>>> print B
[[ 0.          0.          0.          0.        ]
 [ 1.                nan  1.                nan]
 [ 1.5849625  1.5849625  1.5849625  1.5849625]
 [ 2.                nan  2.                nan]]
>>>numpy.sum(B), numpy.nansum(B)
(nan, 12.339850002884624)
```

Summary

In this chapter we have explored in depth the creation and basic manipulation of the object array used by SciPy, as an overview of the NumPy libraries. In particular, we have seen the principles of slicing and masking, which simplify the coding of algorithms to the point of transforming an otherwise unreadable sequence of loops and primitive commands, into an intuitive and self-explanatory set of object calls and methods. We have also learned that the nonbasic modules in NumPy are replicated as modules in SciPy itself. The chapter roughly followed the same structure as the official NumPy reference (which the reader can access at the SciPy pages at `docs.scipy.org/doc/numpy/reference`). There are other good sources that cover NumPy with rigor, and we refer you to any of that other material for a more detailed coverage of this topic.

In the next five chapters we will be accessing the commands that make SciPy a powerful tool in numerical computing. The structure of those chapters is basically a reflection of the different SciPy modules, structured in an order that allows building applications on top of each other.