

2.1 NumPy Arrays

NumPy is the fundamental Python package for scientific computing. It adds the capabilities of N -dimensional arrays, element-by-element operations (broadcasting), core mathematical operations like linear algebra, and the ability to wrap C/C++/Fortran code. We will cover most of these aspects in this chapter by first covering what NumPy arrays are, and their advantages versus Python lists and dictionaries.

Python stores data in several different ways, but the most popular methods are *lists* and *dictionaries*. The Python `list` object can store nearly any type of Python object as an element. But operating on the elements in a list can only be done through iterative loops, which is computationally inefficient in Python. The NumPy package enables users to overcome the shortcomings of the Python lists by providing a data storage object called `ndarray`.

The `ndarray` is similar to lists, but rather than being highly flexible by storing different types of objects in one list, only the same type of element can be stored in each column. For example, with a Python list, you could make the first element a list and the second another list or dictionary. With NumPy arrays, you can only store the same type of element, e.g., all elements must be floats, integers, or strings. Despite this limitation, `ndarray` wins hands down when it comes to operation times, as the operations are sped up significantly. Using the `%timeit` magic command in IPython, we compare the power of NumPy `ndarray` versus Python lists in terms of speed.

```
import numpy as np

# Create an array with 10^7 elements.
arr = np.arange(1e7)

# Converting ndarray to list
larr = arr.tolist()

# Lists cannot by default broadcast,
# so a function is coded to emulate
# what an ndarray can do.
```

```
def list_times(alist, scalar):
    for i, val in enumerate(alist):
        alist[i] = val * scalar
    return alist

# Using IPython's magic timeit command
timeit arr * 1.1
>>> 1 loops, best of 3: 76.9 ms per loop

timeit list_times(larr, 1.1)
>>> 1 loops, best of 3: 2.03 s per loop
```

The `ndarray` operation is ~ 25 faster than the Python loop in this example. Are you convinced that the NumPy `ndarray` is the way to go? From this point on, we will be working with the array objects instead of lists when possible.

Should we need linear algebra operations, we can use the `matrix` object, which does not use the default broadcast operation from `ndarray`. For example, when you multiply two equally sized `ndarrays`, which we will denote as A and B , the $n_{i,j}$ element of A is only multiplied by the $n_{i,j}$ element of B . When multiplying two `matrix` objects, the usual matrix multiplication operation is executed.

Unlike the `ndarray` objects, `matrix` objects can and only will be two dimensional. This means that trying to construct a third or higher dimension is not possible. Here's an example.

```
import numpy as np

# Creating a 3D numpy array
arr = np.zeros((3,3,3))

# Trying to convert array to a matrix, which will not work
mat = np.matrix(arr)

# "ValueError: shape too large to be a matrix."
```

If you are working with matrices, keep this in mind.

2.1.1 Array Creation and Data Typing

There are many ways to create an array in NumPy, and here we will discuss the ones that are most useful.

```
# First we create a list and then
# wrap it with the np.array() function.
alist = [1, 2, 3]
arr = np.array(alist)

# Creating an array of zeros with five elements
arr = np.zeros(5)

# What if we want to create an array going from 0 to 100?
arr = np.arange(100)
```

```

# Or 10 to 100?
arr = np.arange(10,100)

# If you want 100 steps from 0 to 1...
arr = np.linspace(0, 1, 100)

# Or if you want to generate an array from 1 to 10
# in log10 space in 100 steps...
arr = np.logspace(0, 1, 100, base=10.0)

# Creating a 5x5 array of zeros (an image)
image = np.zeros((5,5))

# Creating a 5x5x5 cube of 1's
# The astype() method sets the array with integer elements.
cube = np.zeros((5,5,5)).astype(int) + 1

# Or even simpler with 16-bit floating-point precision...
cube = np.ones((5, 5, 5)).astype(np.float16)

```

When generating arrays, NumPy will default to the bit depth of the Python environment. If you are working with 64-bit Python, then your elements in the arrays will default to 64-bit precision. This precision takes a fair chunk memory and is not always necessary. You can specify the bit depth when creating arrays by setting the data type parameter (`dtype`) to `int`, `numpy.float16`, `numpy.float32`, or `numpy.float64`. Here's an example how to do it.

```

# Array of zero integers
arr = np.zeros(2, dtype=int)

# Array of zero floats
arr = np.zeros(2, dtype=np.float32)

```

Now that we have created arrays, we can reshape them in many other ways. If we have a 25-element array, we can make it a 5×5 array, or we could make a 3-dimensional array from a flat array.

```

# Creating an array with elements from 0 to 999
arr1d = np.arange(1000)

# Now reshaping the array to a 10x10x10 3D array
arr3d = arr1d.reshape((10,10,10))

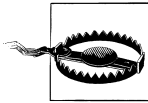
# The reshape command can alternatively be called this way
arr3d = np.reshape(arr1s, (10, 10, 10))

# Inversely, we can flatten arrays
arr4d = np.zeros((10, 10, 10, 10))
arr1d = arr4d.ravel()

print arr1d.shape
(1000,)

```

The possibilities for restructuring the arrays are large and, most importantly, easy.



Keep in mind that the restructured arrays above are just different views of the same data in memory. This means that if you modify one of the arrays, it will modify the others. For example, if you set the first element of `arr1d` from the example above to `1`, then the first element of `arr3d` will also become `1`. If you don't want this to happen, then use the `numpy.copy` function to separate the arrays memory-wise.

2.1.2 Record Arrays

Arrays are generally collections of integers or floats, but sometimes it is useful to store more complex data structures where columns are composed of different data types. In research journal publications, tables are commonly structured so that some columns may have string characters for identification and floats for numerical quantities. Being able to store this type of information is very beneficial. In NumPy there is the `numpy.recarray`. Constructing a `recarray` for the first time can be a bit confusing, so we will go over the basics below. The first example comes from the NumPy documentation on record arrays.

```
# Creating an array of zeros and defining column types
recarr = np.zeros((2,), dtype=('i4,f4,a10'))
toadd = [(1,2., 'Hello'), (2,3., "World")]
recarr[:] = toadd
```

The `dtype` optional argument is defining the types designated for the first to third columns, where `i4` corresponds to a 32-bit integer, `f4` corresponds to a 32-bit float, and `a10` corresponds to a string 10 characters long. Details on how to define more types can be found in the NumPy documentation.¹ This example illustrates what the `recarray` looks like, but it is hard to see how we could populate such an array easily. Thankfully, in Python there is a global function called `zip` that will create a list of tuples like we see above for the `toadd` object. So we show how to use `zip` to populate the same `recarray`.

```
# Creating an array of zeros and defining column types
recarr = np.zeros((2,), dtype=('i4,f4,a10'))

# Now creating the columns we want to put
# in the recarray
col1 = np.arange(2) + 1
col2 = np.arange(2, dtype=np.float32)
col3 = ['Hello', 'World']

# Here we create a list of tuples that is
# identical to the previous toadd list.
toadd = zip(col1, col2, col3)

# Assigning values to recarr
recarr[:] = toadd
```

¹ <http://docs.scipy.org/doc/numpy/user/basics.rec.html>

```
# Assigning names to each column, which
# are now by default called 'f0', 'f1', and 'f2'.

recarr.dtype.names = ('Integers' , 'Floats', 'Strings')

# If we want to access one of the columns by its name, we
# can do the following.

recarr('Integers')
# array([1, 2], dtype=int32)
```

The *recarray* structure may appear a bit tedious to work with, but this will become more important later on, when we cover how to read in complex data with NumPy in the *Read and Write* section.



If you are doing research in astronomy or astrophysics and you commonly work with data tables, there is a high-level package called ATpy² that would be of interest. It allows the user to read, write, and convert data tables from/to FITS, ASCII, HDF5, and SQL formats.

2.1.3 Indexing and Slicing

Python index lists begin at zero and the NumPy arrays follow suit. When indexing lists in Python, we normally do the following for a 2×2 object:

```
alist=[[1,2],[3,4]]

# To return the (0,1) element we must index as shown below.
alist[0][1]
```

If we want to return the right-hand column, there is no trivial way to do so with Python lists. In NumPy, indexing follows a more convenient syntax.

```
# Converting the list defined above into an array
arr = np.array(alist)

# To return the (0,1) element we use ...
arr[0,1]

# Now to access the last column, we simply use ...
arr[:,1]

# Accessing the columns is achieved in the same way,
# which is the bottom row.
arr[1,:]
```

Sometimes there are more complex indexing schemes required, such as conditional indexing. The most commonly used type is `numpy.where()`. With this function you can return the desired indices from an array, regardless of its dimensions, based on some conditions(s).

² <http://atpy.github.com>

```
# Creating an array
arr = np.arange(5)

# Creating the index array
index = np.where(arr > 2)
print(index)
(array([3, 4]),)

# Creating the desired array
new_arr = arr[index]
```

However, you may want to remove specific indices instead. To do this you can use `numpy.delete()`. The required input variables are the array and indices that you want to remove.

```
# We use the previous array
new_arr = np.delete(arr, index)
```

Instead of using the `numpy.where` function, we can use a simple boolean array to return specific elements.

```
index = arr > 2
print(index)
[False False True True True]
new_arr = arr[index]
```

Which method is better and when should we use one over the other? If speed is important, the boolean indexing is faster for a large number of elements. Additionally, you can easily invert True and False objects in an array by using `~ index`, a technique that is far faster than redoing the `numpy.where` function.

2.2 Boolean Statements and NumPy Arrays

Boolean statements are commonly used in combination with the `and` operator and the `or` operator. These operators are useful when comparing single boolean values to one another, but when using NumPy arrays, you can only use `&` and `|` as this allows fast comparisons of boolean values. Anyone familiar with formal logic will see that what we can do with NumPy is a natural extension to working with arrays. Below is an example of indexing using compound boolean statements, which are visualized in three subplots (see Figure 2-1) for context.

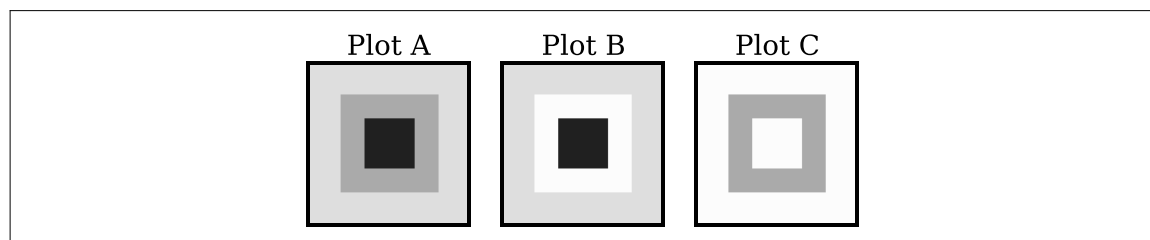


Figure 2-1. Three plots showing how indexing with NumPy works.

```

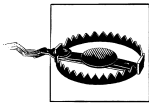
# Creating an image
img1 = np.zeros((20, 20)) + 3
img1[4:-4, 4:-4] = 6
img1[7:-7, 7:-7] = 9
# See Plot A

# Let's filter out all values larger than 2 and less than 6.
index1 = img1 > 2
index2 = img1 < 6
compound_index = index1 & index2

# The compound statement can alternatively be written as
compound_index = (img1 > 3) & (img1 < 7)
img2 = np.copy(img1)
img2[compound_index] = 0
# See Plot B.

# Making the boolean arrays even more complex
index3 = img1 == 9
index4 = (index1 & index2) | index3
img3 = np.copy(img1)
img3[index4] = 0
# See Plot C.

```



When constructing complex boolean arguments, it is important to use parentheses. Just as with the order of operations in math (PEMDAS), you need to organize the boolean arguments contained to construct the right logical statements.

Alternatively, in a special case where you only want to operate on specific elements in an array, doing so is quite simple.

```

import numpy as np
import numpy.random as rand

# Creating a 100-element array with random values
# from a standard normal distribution or, in other
# words, a Gaussian distribution.
# The sigma is 1 and the mean is 0.
a = rand.randn(100)

# Here we generate an index for filtering
# out undesired elements.
index = a > 0.2
b = a[index]

# We execute some operation on the desired elements.
b = b ** 2 - 2

# Then we put the modified elements back into the
# original array.
a[index] = b

```

2.3 Read and Write

Reading and writing information from data files, be it in text or binary format, is crucial for scientific computing. It provides the ability to save, share, and read data that is computed by any language. Fortunately, Python is quite capable of reading and writing data.

2.3.1 Text Files

In terms of text files, Python is one of the most capable programming languages. Not only is the parsing robust and flexible, but it is also fast compared to other languages like C. Here's an example of how Python opens and parses text information.

```
# Opening the text file with the 'r' option,
# which only allows reading capability
f = open('somefile.txt', 'r')

# Parsing the file and splitting each line,
# which creates a list where each element of
# it is one line
alist = f.readlines()

# Closing file
f.close()
:

# After a few operations, we open a new text file
# to write the data with the 'w' option. If there
# was data already existing in the file, it will be overwritten.
f = open('newtextfile.txt', 'w')

# Writing data to file
f.writelines(newdata)

# Closing file
f.close()
```

Accessing and recording data this way can be very flexible and fast, but there is one downside: if the file is large, then accessing or modulating the data will be cumbersome and slow. Getting the data directly into a `numpy.ndarray` would be the best option. We can do this by using a NumPy function called `loadtxt`. If the data is structured with rows and columns, then the `loadtxt` command will work very well as long as all the data is of a similar type, i.e., integers or floats. We can save the data through `numpy.savetxt` as easily and quickly as with `numpy.loadtxt`.

```
import numpy as np

arr = np.loadtxt('somefile.txt')

np.savetxt('somenewfile.txt')
```

If each column is different in terms of formatting, `loadtxt` can still read the data, but the column types need to be predefined. The final construct from reading the data will

be a `recarray`. Here we run through a simple example to get an idea of how NumPy deals with this more complex data structure.

```
# example.txt file looks like the following
#
# XR21 32.789 1
# XR22 33.091 2

table = np.loadtxt('example.txt',
                  dtype='names': ('ID', 'Result', 'Type'),
                  'formats': ('S4', 'f4', 'i2'))

# array([('XR21', 32.78900146484375, 1),
#       ('XR22', 33.090999603271484, 2)],
#      dtype=[('ID', '<S4'), ('Result', '<f4'), ('Type', '<i2')])
```

Just as in the earlier material covering `recarray` objects, we can access each column by its name, e.g., `table['Result']`. Accessing each row is done the same as with normal `numpy.array` objects.

There is one downside to `recarray` objects, though: as of version NumPy 1.8, there is no dependable and automated way to save `numpy.recarray` data structures in text format. If saving `recarray` structures is important, it is best to use the `matplotlib.mlab`³ tools.



There is a highly generalized and fast text parsing/writing package called `AsciiTable`.⁴ If reading and writing data in ASCII format is frequently needed for your work, this is a must-have package to use with NumPy.

2.3.2 Binary Files

Text files are an excellent way to read, transfer, and store data due to their built-in portability and user friendliness for viewing. Binary files in retrospect are harder to deal with, as formatting, readability, and portability are trickier. Yet they have two notable advantages over text-based files: file size and read/write speeds. This is especially important when working with big data.

In NumPy, files can be accessed in binary format using `numpy.save` and `numpy.load`. The primary limitation is that the binary format is only readable to other systems that are using NumPy. If you want to read and write files in a more portable format, then `scipy.io` will do the job. This will be covered in the next chapter. For the time being, let us review NumPy's capabilities.

```
import numpy as np

# Creating a large array
data = np.empty((1000, 1000))
```

³ http://matplotlib.sourceforge.net/api/mlab_api.html

⁴ <http://cxc.harvard.edu/contrib/asciitable/>

```
# Saving the array with numpy.save
np.save('test.npy', data)

# If space is an issue for large files, then
# use numpy.savez instead. It is slower than
# numpy.save because it compresses the binary
# file.
np.savez('test.npz', data)

# Loading the data array
newdata = np.load('test.npy')
```

Fortunately, `numpy.save` and `numpy.savez` have no issues saving `numpy.recarray` objects. Hence, working with complex and structured arrays is no issue if portability beyond the Python environment is not of concern.

2.4 Math

Python comes with its own `math` module that works on Python native objects. Unfortunately, if you try to use `math.cos` on a NumPy array, it will not work, as the `math` functions are meant to operate on elements and not on lists or arrays. Hence, NumPy comes with its own set of math tools. These are optimized to work with NumPy array objects and operate at fast speeds. When importing NumPy, most of the math tools are automatically included, from simple trigonometric and logarithmic functions to the more complex, such as fast Fourier transform (FFT) and linear algebraic operations.

2.4.1 Linear Algebra

NumPy arrays do not behave like matrices in linear algebra by default. Instead, the operations are mapped from each element in one array onto the next. This is quite a useful feature, as loop operations can be done away with for efficiency. But what about when transposing or a dot multiplication are needed? Without invoking other classes, you can use the built-in `numpy.dot` and `numpy.transpose` to do such operations. The syntax is Pythonic, so it is intuitive to program. Or the math purist can use the `numpy.matrix` object instead. We will go over both examples below to illustrate the differences and similarities between the two options. More importantly, we will compare some of the advantages and disadvantages between the `numpy.array` and the `numpy.matrix` objects.

Some operations are easy and quick to do in linear algebra. A classic example is solving a system of equations that we can express in matrix form:

$$\begin{aligned} 3x + 6y - 5z &= 12 \\ x - 3y + 2z &= -2 \\ 5x - y + 4z &= 10 \end{aligned} \tag{2.1}$$

$$\begin{bmatrix} 3 & 6 & -5 \\ 1 & -3 & 2 \\ 5 & -1 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 12 \\ -2 \\ 10 \end{bmatrix} \tag{2.2}$$

Now let us represent the matrix system as $\mathbf{AX} = \mathbf{B}$, and solve for the variables. This means we should try to obtain $\mathbf{X} = \mathbf{A}^{-1}\mathbf{B}$. Here is how we would do this with NumPy.

```
import numpy as np

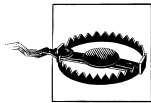
# Defining the matrices
A = np.matrix([[3, 6, -5],
               [1, -3, 2],
               [5, -1, 4]])

B = np.matrix([[12],
               [-2],
               [10]])

# Solving for the variables, where we invert A
X = A ** (-1) * B
print(X)

# matrix([[ 1.75],
#        [ 1.75],
#        [ 0.75]])
```

The solutions for the variables are $x = 1.75$, $y = 1.75$, and $z = 0.75$. You can easily check this by executing \mathbf{AX} , which should produce the same elements defined in \mathbf{B} . Doing this sort of operation with NumPy is easy, as such a system can be expanded to much larger 2D matrices.



Not all matrices are invertible, so this method of solving for solutions in a system does not always work. You can sidestep this problem by using `numpy.linalg.svd`,⁵ which usually works well inverting poorly conditioned matrices.

Now that we understand how NumPy matrices work, we can show how to do the same operations without specifically using the `numpy.matrix` subclass. (The `numpy.matrix` subclass is contained within the `numpy.array` class, which means that we can do the same example as that above without directly invoking the `numpy.matrix` class.)

```
import numpy as np

a = np.array([[3, 6, -5],
              [1, -3, 2],
              [5, -1, 4]])

# Defining the array
b = np.array([12, -2, 10])

# Solving for the variables, where we invert A
x = np.linalg.inv(a).dot(b)
print(x)

# array([ 1.75,  1.75,  0.75])
```

⁵ <http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.svd.html>

Both methods of approaching linear algebra operations are viable, but which one is the best? The `numpy.matrix` method is syntactically the simplest. However, `numpy.array` is the most practical. First, the NumPy array is the standard for using nearly anything in the scientific Python environment, so bugs pertaining to the linear algebra operations will be less frequent than with `numpy.matrix` operations. Furthermore, in examples such as the two shown above, the `numpy.array` method is computationally faster.

Passing data structures from one class to another can become cumbersome and lead to unexpected results when not done correctly. This would likely happen if one were to use `numpy.matrix` and then pass it to `numpy.array` for further operations. Sticking with one data structure will lead to fewer headaches and less worry than switching between matrices and arrays. It is advisable, then, to use `numpy.array` whenever possible.