NumPy Guide

By: Jose Marcial Portilla

jmportilla@pieriandata.com

Table of Contents

Introduction

The Python programming language has exploded in popularity and has rapidly become one of the standard programming languages for performing data analysis. It is object-oriented and is a higher-level language that promotes clean and readable code. Python also allows access to the use of legacy C, Fortran, or R code. A core part of any data analysis is the ability to perform complex numerical analysis on large datasets efficiently. Commercial tools such as MATLAB, Maple, and Mathematica exist, but are limited compared to a general-purpose programming language such as Python. Fortunately, the NumPy open source library for Python allows users to freely have access to powerful numerical analysis in Python.

NumPy (which stands for Numerical Python) is the primary open source library for scientific computing in Python. NumPy allows the construction of arrays and matrices and includes a large amount of built-in mathematical functions for linear algebra, Fourier transformations, random number generation, and much more. For linear algebra operations, NumPy utilizes the LAPACK linear algebra library (if already installed on your system otherwise it provides its own implementation). LAPACK is a recognized library that was originally written in Fortran.

Benefits of NumPy

A common question is why use NumPy instead of other open-source alternatives (such as Octave or SciLab). There are several benefits to using NumPy. NumPy code is designed to be clean and legible and compared to straight Python code requires fewer loops because operations are designed to work directly on arrays and matrices. The algorithms that NumPy is built on top of have been well-tested and have been designed for high performance.

Compared to standard Python Data Structures (such as lists or nested lists) Numpy arrays are stored more efficiently and array IO is much faster. Optimal reading and writing of data was kept in mind when creating the NumPy array data structure.

Significant portions of NumPy are written in C with Python serving only as a shell. This makes NumPy much faster than a pure Python implementation. Most importantly, NumPy is open-source and has a very strong developer community allowing users to use the software freely.

Brief History of NumPy

NumPy is based on Numeric, which was first released in 1995. For several reasons Numeric did not make it into the standard Python library, so it requires a separate installation (which we will cover in the next chapter).

In 2001 several people created SciPy, which is an open-source scientific computing library for Python, which aimed to provide functionality similar to MATLAB, Maple, and Mathematica. As SciPy was beginning to be developed further, developers were beginning to grow frustrated with Numeric and began developing Numarray (which is now deprecated).

In 2005 Travis Oliphant decided to integrate some of Numarray's features into Numeric, which eventually lead to an entire rewrite resulting in NumPy 1.0 in 2006. Originally, NumPy code was part of SciPy but it has now been separated, with SciPy using NumPy for array and matrix operations. For a more detailed history, visit NumPy's official documentation page.

Alright, now that we've learned about the benefits and history of NumPy, let's go ahead and get started with using it!

Chapter 1 NumPy Basics

In this chapter we will first cover on getting NumPy up and running on your System, as well as all its dependencies. However, it should be noted that this book assumes the reader already has experience with Python and feels comfortable installing packages on to their system. Then we will continue with an introduction of the basic functionality of NumPy.

Python Installation

In order to use NumPy, you will need to have Python installed. If you do not already have Python installed, it is highly recommended you use the Anaconda Distribution of Python. The Anaconda distribution is built and maintained by Continuum Analytics and provides a free distribution of Python and many libraries for Scientific Computing (including NumPy). To install the Anaconda Distribution of Python, go to https://www.continuum.io/downloads and follow the installation instructions for your Operating System.

This book assumes the use of Python 2.7 for the syntax used throughout.

Jupyter Notebooks

A quick note on the Jupyter Notebook environment, I highly suggest you try out the Jupyter Notebook Development Environment for learning Python and NumPy. The Jupyter Notebook is a web application that allows you to create and share documents that contain live code, equations, visualizations and explanatory text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, machine learning and much more. The Notebook has support for over 40 programming languages, including those popular in Data Science such as Python, R, Julia and Scala. Notebooks can be shared with others using email, Dropbox, and GitHub. Code can also produce rich output such as images, videos, LaTeX, and JavaScript. Interactive widgets can be used to manipulate and visualize data in real-time. The environment allows you to leverage big data tools, such as Apache Spark, from Python, R and Scala. You can then explore that same data with pandas, scikit-learn, ggplot2, dplyr, etc. You can try out Jupyter Notebooks with no installation at https://try.jupyter.org/. For installation instructions, visit http://jupyter.readthedocs.org/en/latest/install.html. Installation is usually very straight-forward, especially if you have the Anaconda Distribution installed on your computer, in which case you can simply type into your terminal: **conda install jupyter.**

NumPy Installation

The preferred and suggested method for installing NumPy on to your system is through the use of the Anaconda distribution with the use of the conda install. If you have downloaded the Anaconda Distribution, open your terminal (or any command prompt such as PowerShell for Windows) and type **conda install numpy**. This will install NumPy and all its dependencies on to your system. If you are using another installation of Python, you can use the PyPi package installer to install NumPy. Go to your command prompt or terminal and type: **pip install numpy**.

Now that NumPy and Python are successfully installed on your system, let's begin to learn about basic operations in NumPy! From now on we will refer to NumPy as Numpy (lowercase p) to refer to working with Numpy syntax and will use the NumPy notation when referring to NumPy in the context of a general library used with Python.

Code Syntax

A quick note on the code syntax and formatting that follows. All code is presented as if the code was being put directly into the Python IDLE interpreter. A use of **>>>** denotes in an input, with output being show directly underneath. By convention, for most cells it is assumed that numpy has been imported as np, for example:

Code Listing 1

```
>>> import numpy as np
```

Cells at the beginning of new sections will have import statement reminders.

NumPy Numerical Types

Before discussing array creation and numerical functions in NumPy, it is important to understand the various types of numbers available in NumPy versus standard Python. Standard Python has an integer, float, and complex type of numbers. For scientific computing however, this is limiting. NumPy addresses this issue by having a variety of numerical types at its disposal which are dependent on the memory requirements. The following table gives an overview of the various NumPy numerical types, with their name and description:

Table 1: NumPy Numerical Types

NumPy Numerical Types	
Type	Description
bool	Boolean (True or False) stored as a bit
inti	Platform integer (normally either int32 or int64)
int8	Byte (-128 to 127)

int16	Integer (-32768 to 32767)
int32	Integer (-2 ** 31 to 2 ** 31 -1)
int64	Integer (-2 ** 63 to 2 ** 63 -1)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 2 ** 32 - 1)
uint64	Unsigned integer (0 to 2 ** 64 - 1)
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64 or float	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex64	Complex number, represented by two 32-bit floats (real and imaginary components)
complex128 or complex	Complex number, represented by two 64-bit floats (real and imaginary components)

For many built-in functions in Numpy, the numerical data type can be specified when creating Numpy objects, such as an array. For each of the data types in the above table, there is a corresponding conversion function which shares the same name as the data type. For example:

Code Listing 2

```
>>> np.float16(12)

12.0
```

Arrays

For experience Python users, arrays will look and behave similarly to Python's built-in list object. However, NumPy arrays are more efficient for numerical operations and require less explicit loops than standard Python code.

NumPy Array Object

NumPy has a multidimensional array object known as the **ndarray**. The array object contains the actual data and can also contain metadata describing the data. There are 5 ways to create an array in NumPy:

Methods of Creating a NumPy array

1. Converting another Python data structure to an array (such as a list)
2. Built-in NumPy array creation objects (e.g. arrange, ones, zeros, etc.)
3. Reading an array from disk memory.
4. Creating arrays from raw bytes.
5. Using special library functions, such as random.

Let's now see some examples of these methods.

Converting Python Objects to Numpy arrays

For the most part, any numerical data arranges in an array-like structure in Python can then be converted directly in a Numpy array using the **array()** function. By convention, Numpy is imported as np. Let's see some example of converting Python objects into arrays.

Convert a list by passing it into the **array()** function.

Code Listing 3

```
>>> import numpy as np

>>> arr = np.array([1,2,3,4])
```

Convert nested lists to create higher dimensional matrices.

Code Listing 4

```
>>> arr = np.array([[1,2,3],[4,5,6],[7,8,9]])
```

The **array()** function is also string aware of complex number notation. For example:

Code Listing 5

```
>>> arr = np.array([[ 2.+0.j, 1.+0.j], [ 1.+0.j, 1.+0.j]])
```

Built-in Array Functions

Numpy has a variety of built-in functions for creating arrays without having to create a Python object first.

The **zeros()** function will create an array with filled 0 values in a specified shape. The default data type will be a floating point number. For example, to create a three by three element matrix of zeros:

Code Listing 6

```
>>> np.zeros((3,3))

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

Notice the use of brackets to denote the dimensions. Each dimension corresponds to a bracket pairing. In the above case we have a two-dimensional array that is 3 by 3.

A similar function **ones()** will create an array consisting of ones of a specified size. For example:

Code Listing 7

```
>>> np.ones((3,3))

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

Numpy also comes with a built-in function of r creating identity matrixes, called the **eye()** function. For example:

Code Listing 8

```
>>> np.eye((3,3))

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

For creating an array with a specified fill value, the function **full()** can be used to create the array. The first argument passed is the desired shape and the second argument is the fill value. For example:

Code Listing 9

```
>>> np.full((3,3),5)

array([[ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.]])
```

Another built-in function is **arange()**, which will create arrays with regularly incrementing values. It can be used with a single input (similar to the **range()** function in standard Python) or with multiple inputs to specify step sizes. The main parameters are start, finish, and step size. For all possible arguments, check the docstring of the function. For usage examples:

Code Listing 10

```
>>> np.arange(5)

array([0, 1, 2, 3, 4])

>>> np.arange(2, 5, dtype=np.float)

array([ 2., 3., 4.])

>>> np.arange(0,1,0.1)

array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9])
```

Another built-in function used is the **linspace()** function, which will create arrays with a specified number of elements, spaced equally between the given start and stop values. For example:

Code Listing 11

```
>>> np.linspace(0,100,5)
array([   0.,   25.,   50.,   75.,  100.])

>>> np.linspace(2,3,5)
array([ 2.  ,  2.25,  2.5 ,  2.75,  3.  ])
```

This function is useful when the number of elements in the array must meet a certain condition and **arange()** will generally not fulfill this need easily. It should also be noted that Numpy has a **logspace()** function which returns numbers spaced evenly on a logarithmic scale.

Another built-in function to create arrays is the **indices()** function which will create a set of arrays as a stacked "one-higher" dimensioned array, with each set representing the variation in that dimension. A code example serves as a clearer explanation:

Code Listing 12

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

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

Note how the output represents the indices of the grid. This function is more often used when attempting to evaluate other functions of multiple dimensions on a regular grid.

Reading Arrays from Disk

Numpy comes with several functions to create arrays by reading data from the computer disk. One of the most commonly used is the **genfromtxt** function. Simply put, the **genfromtxt** function performs two actions. It first loops through each line of an input file and converts each line into a sequence of strings. Secondly, it converts each of these strings into an appropriate data type. For most cases, **genfromtxt** is able to take missing data into account. There are simpler and faster functions in Numpy such as **loadtxt** which cannot take missing information into account.

Let’s start exploring **genfromtxt** through an example. First some imports:

Code Listing 13

```
>>> import numpy as np
>>> from StringIO import StringIO
```

We will first create an example text file with the use of **StringIO** then use **genfromtxt** to pass in two arguments, the text itself and the delimiter separating each element in the text.

Code Listing 14

```
>>> txt = "1,2,3,4,5\n6,7,8,9,10"
>>> np.genfromtxt(StringIO(txt),delimiter=",")
array([[  1.,   2.,   3.,   4.,   5.],
       [  6.,   7.,   8.,   9.,  10.]])
```

Keep in mind that while a comma is the most common separator for files (such as .csv files) other delimiters such as a tab space (‘\t’) can be used. If no delimiter is passed as a parameter then **genfromtxt** treats the **delimiter=None** so that the text is split along any white spaces, including consecutive white spaces.

If you are dealing with a fixed-width file then the columns are defined by a given number of characters. The delimiter argument in **genfromtxt** is flexible enough to handle this task by accepting an integer argument. For example:

Code Listing 15

```
>>> txt = " 1  1  1\n  2  2 10\n100333  3"
>>> np.genfromtxt(StringIO(txt),delimiter=3)
array([[   1.,    1.,    1.],
       [   2.,    2.,   10.],
       [ 100.,  333.,    3.]])
```

genfromtxt also comes with a convenient **autostrip** argument which allows individual entries to be stripped of leading or trailing white spaces. This is best shown through a comparison of two examples (note the use of the **dtype** parameter to work with strings):

Code Listing 16

```
>>> txt = "a, b ,    c   "
>>> np.genfromtxt(StringIO(txt),dtype='|S5',delimiter=',')
array(['a', ' b ', '    c'],
      dtype='|S5')

>>>np.genfromtxt(StringIO(txt),dtype='|S5',delimiter=',',autostrip=True)
array(['a', 'b', 'c'],
      dtype='|S5')
```

Comments can also be ignored through the use of a **comments** argument. Pass the comment indicator to ignore lines beginning as a comment. For example we can pass '#' to ignore any Python comments in a text:

Code Listing 17

```
>>> txt = """#

... # This is a comment

... # This is also a comment

... 1, 1

... 2, 2

... 3, 3 # A comment after the third line of data

... 4, 4

... # One last comment
... 5, 5
... """


>>> np.genfromtxt(StringIO(txt), comments="#", delimiter=",")
[[ 1.  1.]
 [ 2.  2.]
 [ 3.  3.]
 [ 4.  4.]
 [ 5.  5.]]
```

We can skip lines in a text with the **skip_header** and **skip_footer** arguments. You can pass integer values to these parameters to specify how many header or footer columns to skip. For example:

Code Listing 18

```
>>> txt = "\n".join(str(i) for i in range(5))
>>> np.genfromtxt(StringIO(txt))
array([ 0.,  1.,  2.,  3.,  4.])
>>> np.genfromtxt(StringIO(txt),skip_header=1, skip_footer=1)
array([1.,  2.,  3.])
```

By default, **skip_header**=0 and **skip_footer**=0, so that no lines are skipped.

Lastly, we can choose which columns to accept using the **usecols** argument. For example:

Code Listing 19

```
>>> txt = '1 2 3 4\n5 6 7 8'
>>> np.genfromtxt(StringIO(txt), usecols=(0, -1))

array([[ 1.,  4.],
       [ 5.,  8.]])
```

Now that we’ve seen the various ways to create array objects in Numpy, let’s begin to look at the basic attributes of arrays in Numpy.

Array Attributes

Ndim

The **ndim** attribute will return the number of dimensions of an array. For example:

Code Listing 20

```
>>> arr = np.array([[1,2],[3,4]])
>>> arr.ndim
2
```

Size

The **size** attribute will return the number of elements. For example:

Code Listing 21

```
>>> arr = np.arange(6)
>>> arr.size
6
```

Itemsize

The **itemsize** attribute returns the number of bytes for each element in the array. For example:

Code Listing 22

```
>>> arr = np.arange(6)
>>> arr.itemsize
8
```

Nbytes

The **nbytes** attribute will return the total number of bytes the array requires. This is the same as the product of **itemsize** and **size**:

Code Listing 23

```
>>> arr = np.arange(6)
>>> arr.nbytes
48
```

T

The **T** attribute will return a transposed version of the original array. It is important to note that this does not affect the original array permanently. For example:

Code Listing 24

```
>>> arr = np.arange(24).reshape(6,4)
>>> arr

array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
>>> arr.T

array([[ 0,  4,  8, 12, 16, 20],
       [ 1,  5,  9, 13, 17, 21],
       [ 2,  6, 10, 14, 18, 22],
       [ 3,  7, 11, 15, 19, 23]])
>>> arr

array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

Real

The **real** attribute will return the real part of an array, or the array itself if it only contains real numbers. For example:

Code Listing 25

```
>>> arr = np.array([1.j + 1, 2+ 3j])
>>> arr.real

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

Imag

The **imag** attribute will return the imaginary portion of an array. For example:

Code Listing 26

```
>>> arr = np.array([1.j + 1, 2+ 3j])
>>> arr.imag

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

Flat

The **flat** attribute allows you to iterate through an array as if it was a flat array. It does this by returning a **numpy.flatiter** object, which you can then iterate through. This is best shown through example:

Code Listing 27

```
>>> arr = np.arange(9).reshape(3,3)
>>> arr

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

>>> arr.flat

<numpy.flatiter at 0x1af5720>

>>> for ele in arr.flat:
    ... print ele

0
1
2
3
4
5
6
7
8
```

It should also be noted that you can directly index elements (we will learn about indexing in the next section) directly from the flattened object. For example:

Code Listing 28

```
>>> arr.flat[3]

3

>>>arr.flat[[1,3]]

array([1, 3])
```

Now that we've learned about basic array attributes, let's begin to look at how we can perform indexing on these arrays!

Array Indexing

Indexing arrays in Python is one of the most fundamental operations that can be performed on an array. Indexing arrays will feel very familiar in Python since they operate in a similar fashion to indexing lists. In Numpy there are several options for indexing which allows for greater flexibility and control. In this section we will get a brief overview of the most common indexing methods. Let's begin with single element indexing.

Single Element

Indexing for a single element will feel similar to indexing a Python list object. It is a 0-based index and also accepts negative index requests for indexing from the end of an array. Let's create an array and see a simple example:

Code Listing 29

```
>>> arr = np.arange(6)
>>> arr
array([0, 1, 2, 3, 4, 5])
```

Indexing single element:

Code Listing 30

```
>>> arr[3]
3

>>>arr[-1]
5
```

Now that we've seen a one-dimensional index situation, let's explore multidimensional array indexing for single elements. We'll start by reassigning the shape of the array object.

Code Listing 31

```
>>> arr.shape = (2,3)
>>> arr

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

Now we can index a multidimensional array through the use of commas instead of separate brackets. For example:

Code Listing 32

```
>>> arr[1,2]
5

>>>arr[1,-1] #With negative indexing
5
```

Note how we did not have to write two pairs of brackets. However, if we index a multidimensional array with fewer requested indices than dimensions, we get a back a sub-dimensional array (a subset of the original array). For example:

Code Listing 33

```
>>> arr[0]

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

It should be noted that the returned array is not a copy of the original but instead points to the same values in memory as the original array. Since in this case the first position of the array (index 0) is returned, we can actually continue to index that sub-array. For example:

Code Listing 34

```
>>> arr[0][0]
0
```

It should also be noted that in terms of syntax, **arr[0,1]** is the same as **arr[0][1]**, to show this:

Code Listing 35

```
>>> arr[0][1] == arr[0,1]
True
```

However the second case of two bracket pairs is more inefficient because a new temporary array is created after the first index so that it can then be indexed by the second bracket pair.

Slicing

Now that we've seen how to select individual elements let's take a look at taking slices of arrays. We'll start with some simple examples that will look familiar to Python users familiar with slicing lists.

Code Listing 36

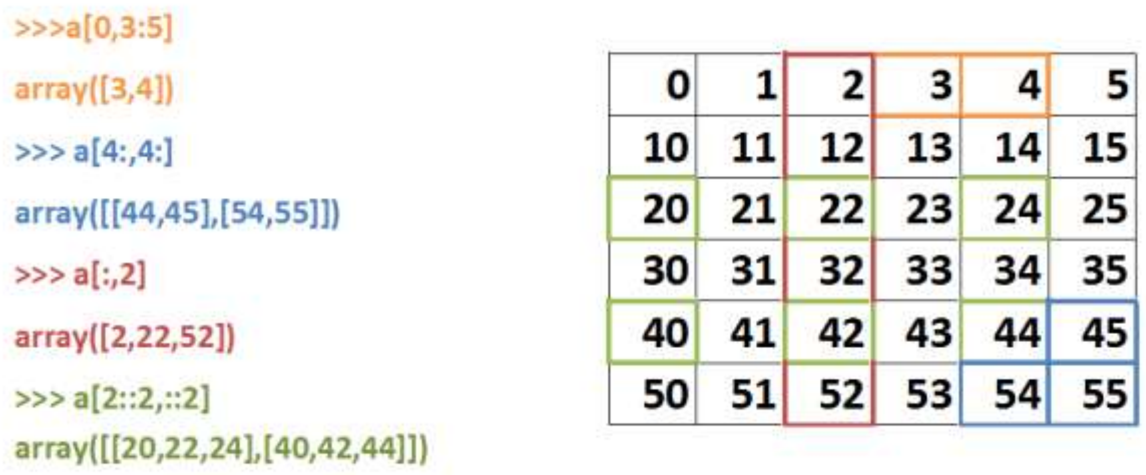
```
>>> arr = np.arange(10)
>>> arr

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

Now let's see several examples of slicing through index commands:

Code Listing 37

```
>>> arr[3:7]

array([3, 4, 5, 6])

>>> arr[:-1]

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


>>> arr[::-1]

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

>>> arr[0:10:2]

array([0, 2, 4, 6, 8])
```

For a visual demonstration of step size arguments in indexing, refer to Figure 1 below:



```
>>>a[0,3:5]
array([3,4])
>>> a[4:,4:]
array([[44,45],[54,55]])
>>> a[:,2]
array([2,22,52])
>>> a[2::2,::2]
array([[20,22,24],[40,42,44]])
```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

Figure 1: Array Slicing Examples

It should be noted that these slices of arrays do not copy the array data but only produce a new view of the original array data. There are some more advanced methods of indexing using arrays which we discuss in the next section.

Using arrays as an index

An interesting feature of Numpy arrays is the ability to use index arrays for selecting lists of values out of arrays to show a new array. Index arrays are one of the most useful features of Numpy as they allow the avoidance of using loops to go through individual elements, causing a great improvement to performance.

The Numpy arrays may be indexed with other arrays or lists. The use of index arrays can be useful for both simple and complex cases. It's important to note that when using index arrays what is returned is a copy of the original data, not a view as was the case for slicing. Index arrays must have a **dtype** of an integer. Each integer value of the index array then corresponds to an element index position. Let's see some examples to illustrate this process:

Code Listing 38

```
>>> arr = np.linspace(0,20,11)
>>> arr

array([  0.,   2.,   4.,   6.,   8.,  10.,  12.,  14.,  16.,
18.,  20.])

>>> ind = np.array([1,2,3])
>>> arr[ind]
```

```
array([ 2.,  4.,  6.])

>>> ind = np.array([1,0,1,1])
>>> arr[ind]

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

>>> ind = np.array([2,4,7,2])
>>> arr[ind]

array([  4.,   8.,  14.,   4.])
```

It should also be noted that negative values will also be accepted, for example:

Code Listing 39

```
>>> ind = np.array([-2,0,-1])
>>> arr[ind]

array([ 18.,   0.,  20.])
```

An error will occur if you attempt to retrieve a value out of the index bounds:

Code Listing 40

```
>>> ind = [100]
>>> arr[ind]

IndexError: index 100 is out of bounds for axis 1 with size 11
```

In most situations the array returned is the same shape as the index array but with the values specified by the elements of the index array.

Multidimensional array indexing

When working with multidimensional arrays indexing can become more complex. These use cases are generally not as common but are used in some scientific fields where data is described by many dimensions. We will start with a simple example of multidimensional indexing and gradually increase the complexity of the examples. To begin let's create a multidimensional array and pass two separate index arrays.

Code Listing 41

```
>>> arr = np.arange(36).reshape(6,6)
>>> arr

array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35]])
```

Now we will create two separate index arrays and use them to index the multidimensional array:

Code Listing 42

```
>>> ind1 = np.array([1,3])
>>> ind2 = np.array([2,4])

>>> arr[ind1,ind2]

array([ 8, 22])
```

In the case of indexing multidimensional arrays, if the index arrays have matching shapes (as is the case above) and there exists an index array for each dimension of the array which is being indexed, then the resulting array will have the same shape as the index arrays and the values will correspond to the index set for each of the positions in the index arrays. So in the example above, the array **[8,22]** is returned because **arr[1,2]** is 8 and **arr[3,4]** is 22.

If the index arrays do not have the same shape then Numpy will attempt to broadcast the index arrays to the same shape. If there is an exception a mismatch error is raised. For example:

Code Listing 43

```
>>> arr = np.arange(36).reshape(6,6)

>>> ind1 = np.array([1,3])
>>> ind2 = np.array([2,4,1])

>>> arr[ind1,ind2]

IndexError: shape mismatch: indexing arrays could not be broadcast
together with shapes (2,) (3,)
```

By broadcasting index arrays can be combined with scalars for the other indices. We will cover broadcasting in general later on, but for now we can see a quick example of a single scalar being broadcasted for multidimensional indexing:

Code Listing 44

```
>>> arr = np.arange(36).reshape(6,6)

>>> ind1 = np.array([1,3])
>>> arr[ind1,1]

array([ 7, 19])
```

Continuing on with increasing complexity of the indexing, we can partially index arrays. To show how this works let's start by explaining an example:

Code Listing 45

```
>>> arr = np.arange(36).reshape(6,6)
>>> arr

array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35]])



>>> arr[np.array([0,1,3])] #Partial Indexing

array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [18, 19, 20, 21, 22, 23]])
```

The result is a creation of a new array where the value of each element in the index array selects one row form the array being indexed and the resulting array has the shape seen (defined by the size of the row and number of index elements).

While this usage of index arrays is uncommon, it is used often in certain fields such as image analysis where mapping RGB values is necessary for multidimensional indexing. In a general case, the shape of the returned array is the concatenation of the shape of the index array with the shape of any non-indexed dimensions in the array being indexed.

Boolean index arrays

Boolean arrays can be used in Numpy to index arrays based on Boolean conditions. Boolean indexing (also referred to as "masking") must use arrays that are the same shape as the original array being indexed.

Let's begin with a simple example:

Code Listing 46

```
>>> arr = np.arange(5)
>>> arr

array([0, 1, 2, 3, 4])

>>> boo = arr > 2
>>> boo

array([False, False, False,  True,  True], dtype=bool)

>> arr[boo]

array([3, 4])
```

The result of **arr[boo]** is a one-dimensional array which contains all of the elements in the indexed array that correspond to the True elements of the Boolean array. Just like with normal index arrays, the array which is returned is a copy of the original data, not a view like in slicing.

Boolean array usage also works for multidimensional arrays. For example:

Code Listing 47

```
>>> arr = np.arange(25).reshape(5,5)
>>> arr

array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])

>>> boo = arr > 2
>>> boo

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

A Boolean array can also be used to index arrays which contain more dimensions than the Boolean array. This is best shown through an example:

Code Listing 48

```
>>> arr = np.arange(20).reshape(2,2,5)
>>> arr

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

       [[10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]]])

>>> boo = np.array([[True, False], [False, True]])
>>> arr[boo]

array([[ 0,  1,  2,  3,  4],
       [15, 16, 17, 18, 19]])
```

Note how the resulting shape of **arr[boo]** is (2,5). Using Boolean arrays in this context is best used in cases where the end result usage is clear to the user. The full Numpy documentation contains further details on this topic.

Index arrays combined with Slices

Another interesting feature of Numpy's indexing is the ability to combine index arrays with slices. For example:

Code Listing 49

```
>>> arr = np.arange(25).reshape(5,5)
>>> arr



array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])

>>> arr[np.array([0,2,4]),1:3]

array([[ 1,  2],
       [11, 12],
       [21, 22]])
```

In the above example the slice is converted into an index array that is then broadcasted. In the case above the rows with index 0,2 and 4 are returned that fit in the column slice of 1:3. Note the shape of the array that is returned is (3,2).

Index Structuring

Numpy contains an **np.newaxis** object which can be used for matching array shapes with expressions and assignments. The **np.newaxis** object can be used within array indices to add new dimensions with a size of one. This is best shown through example:

Code Listing 50

```
>>> arr = np.arange(25).reshape(5,5)
>>> arr

array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])

>>> arr.shape
(5,5)

>>> arr[:,np.newaxis,:].shape
(5,1,5)
```

It’s important to note that there are no new elements in the array, just an increase in the dimensionality of the array. This is useful when attempting to combine two arrays in a way that would otherwise require reshaping operations. For example:

Code Listing 51

```
>>> arr = np.arange(6)
>>> arr

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

>>> arr[:,np.newaxis] + arr[np.newaxis,:]

array([[ 0,  1,  2,  3,  4,  5],
       [ 1,  2,  3,  4,  5,  6],
       [ 2,  3,  4,  5,  6,  7],
       [ 3,  4,  5,  6,  7,  8],
       [ 4,  5,  6,  7,  8,  9],
       [ 5,  6,  7,  8,  9, 10]])
```

A useful feature in Numpy is the ellipses syntax which can be used to select and unspecified dimensions in full. For example:

Code Listing 52

```
>>> arr = np.arange(16).reshape(2,2,2,2)
>>> arr

array([[[[ 0,  1],
         [ 2,  3]],

        [[ 4,  5],
         [ 6,  7]]],


       [[[ 8,  9],
         [10, 11]],

        [[12, 13],
         [14, 15]]]])

>>> arr[0,...,1] # Use ellipses to grab unspecified dimension

array([[1, 3],
       [5, 7]])
```

Using the ellipses is equivalent to specifying to select any “in-between” dimensions. For example:

Code Listing 53

```
>>> arr[0,:,:,1]

array([[1, 3],
       [5, 7]])
```

Value Assignment through Indexing

As we’ve shown previously we can reassign elements in an array using single indexing, slices, and boolean and mask arrays. The only requirement here is that the value being assigned must be shape consistent with the section of the array selected. Let’s see some more examples of this:

Code Listing 54

```
>>> arr = np.arange(11)
>>> arr[3:6] = 100
>>> arr
```

```
array([  0,   1,   2, 100, 100, 100,   6,   7,   8,   9,  10])
```

We can also directly assign an entire array as long as it is the correct size:

Code Listing 55

```
>>> arr[3:6] = np.arange(3)
>>> arr

array([ 0,  1,  2,  0,  1,  2,  6,  7,  8,  9, 10])
```

It should be noted that these assignments are always made to the original array of data. There are some self-reference operations that may be counterintuitive. For example:

Code Listing 56

```
>>> arr = np.arange(0,100,10)
>>> arr

array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])

>>> arr[np.array([3,3,3,4])] += 100
>>> arr

array([  0,  10,  20, 130, 140,  50,  60,  70,  80,  90])
```

Sometimes people expect the element at the third index to have 100 added to it three times when in practice the element of 30 only turns into 130. This is because a new array is extracted from the original as a temporary copy which contains the values at the 3,3,3,4 indices. Then the value of 100 is added to the temporary array and then the temporary array is assigned back to the original. This way the array value at arr[3] + 100 is assigned three times, rather than being incremented three times.

That’s it for the basics of NumPy array indexing! Up next we will learn about broadcasting in NumPy and the rules of operations associated with it.

Broadcasting

Broadcasting is one of the most fundamental concepts to understand in Numpy and is a source of Numpy’s programming power. Broadcasting in Numpy allows you to do numeric operations in Python with the inherent speed of the C language. In a general sense, broadcasting lets you “broadcast” a smaller array or scalar across a larger array to perform operations without having to worry that the all the dimensions of you arrays line up. Once you use Numpy more you will begin to build an intuitive sense of broadcasting across arrays.

In most paired operations, Numpy performs the operation on an element by element basis. In a simple case, the arrays have the exact same shape. For example:

Code Listing 57

```
>>> arr = np.array([1,2,3,4,5])
>>> mul = np.array([2,2,2,2,2])
>>> arr * mul

array([ 2,  4,  6,  8, 10])
```

Due to broadcasting, Numpy can deal with similar operations where the shapes of the arrays do not match (under certain constraints). A simple example of this is broadcasting a single scalar across a larger array. For instance:

Code Listing 58

```
>>> arr * 2

array([ 2,  4,  6,  8, 10])
```

Here we can see that we get the same result as the previous example. The scalar, in this case 2, has been applied to each element in the array **arr** for the multiplication operation.

It should be noted that “under the hood” Numpy does not actually make an array version of the scalar, instead it uses the singular value to save on memory and be as computationally efficient as possible. In this way, the second example is more memory efficient than the first one.Broadcasting Rules

As previously mentioned there are certain conditions that must be met for broadcasting to work properly. When performing operations on two arrays, Numpy first compares their shapes element-wise. It begins with the trailing elements and then works its way forwards. The two dimensions achieve compatibility when they are equal or one of them is 1.

If neither of these conditions are met than an exception occurs: **ValueError: frames are not aligned**. This indicates that the arrays have incompatible shapes. However it should be noted that the arrays do not need to have the same number of dimensions. For example, an array representing some pixel image may have dimensions of 128x128x3. If you wanted to scale the every element in the last dimension (representing the simple RGB pixel color values) you would only need to line up the trailing dimension. For example, consider the pseudocode below:

Code Listing 59

```
Pixels  3d array): 128 x 128 x 3

Scale  (1d array):             3

Result (3d array): 128 x 128 x 3
```

When the dimensions are compared, if one of them is equal to 1, then that element is “stretched” to be broadcasted across the corresponding dimension of the other array.

This is best shown through several examples. Consider the various cases below in pseudocode and pay close attention to dimensions of the arrays.

Code Listing 60

```
A      (2d array):  2 x 3

B      (1d array):      1

Result (2d array):  2 x 3


A      (2d array):  2 x 3

B      (1d array):      3

Result (2d array):  2 x 3


A      (3d array):  12 x 3 x 4

B      (3d array):  12 x 1 x 4

Result (3d array):  12 x 3 x 4



A      (3d array):  12 x 3 x 4

B      (2d array):       3 x 4

Result (3d array):  12 x 3 x 4


A      (3d array):  12 x 3 x 4

B      (2d array):       3 x 1

Result (3d array):  12 x 3 x 4
```

Let’s see some pseudocode examples that would not broadcast:

Code Listing 61

```
A      (1d array):  5

B      (1d array):  6 # trailing dimensions do not match
```

```
A      (2d array):      3 x 2
B      (3d array):  8 x 4 x 3 # second from last dimensions mismatched
```

Finally, let's look at various examples of broadcasting in practice with some real code. We'll begin by defining some arrays and then attempting various arithmetic operations on them.

Code Listing 62

```
>>> a = np.arange(4)
>>> aa = x.reshape(4,1)
>>> b = np.ones(5)
>>> c = np.ones((3,4))

>>> a.shape
(4,)

>>> b.shape
(5,)

>>> a + b
<type 'exceptions.ValueError'>: shape mismatch: objects cannot be broadc
ast to a single shape

>>> aa.shape
(4, 1)

>>> b.shape
(5,)

>>> (aa + b).shape
(4, 5)
```

```
>>> aa + b
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.,  3.],
       [ 4.,  4.,  4.,  4.,  4.]])

>>> a.shape
(4,)

>>> c.shape
(3, 4)

>>> (a + c).shape
(3, 4)

>>> a + c
array([[ 1.,  2.,  3.,  4.],
       [ 1.,  2.,  3.,  4.],
       [ 1.,  2.,  3.,  4.]])
```

Chapter 2 Shaping Arrays

Now that we have an understanding of how to create arrays, their basic attributes, indexing and broadcasting, and how to perform some basic operations on them we can begin to learn about how to shape these arrays. We've already seen how **reshape** is used in a few examples in Chapter 1, but there are many more methods available in Numpy. Let's begin by learning about basic shaping of arrays.

Basic Shaping

Reshape

We've seen how reshape can accept a tuple of dimensions and reshape an existing array. For example:

Code Listing 63

```
>>> arr = np.arange(9).reshape(3,3)
>>> arr

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

Ravel

A common task is to flatten an array, we've seen how to produce an iterable "flat" object with the **flat** attribute, but we can return a full flattened array with **ravel.** For example:

Code Listing 64

```
>>> arr.ravel()

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

Flatten

The **flatten** method does the same thing as **ravel** but instead of returning just a view of the array, **flatten** will allocate new memory. Thus **flatten** is generally considered safer to use since changes to this object won't affect the original data.

Code Listing 65

```
>>> arr.flatten()
```

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

Transpose

As expected, **transpose** will return a transposed matrix of the original array.

Code Listing 66

```
>>> arr

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

>>> arr.transpose()

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

Resize

The **resize** method operates exactly the same as the **reshape** method, however if modifies the original data in-place. For example:

Code Listing 67

```
>>> arr.resize(1,9)
>>> arr

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

Stacking Arrays

Now that we've learned about reshaping arrays, let's discuss stacking multiple arrays together in various ways using some built-in Numpy functions.

Vstack

To vertically stack arrays, we can pass a tuple into the **vstack()** function. For example:

Code Listing 68

```
>>> import numpy as np
>>> x = np.arange(9).reshape(3,3)
>>> y = 2 * x

array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 0,  2,  4],
       [ 6,  8, 10],
       [12, 14, 16]])
```

Hstack

You can use the **hstack** method to perform horizontal stacking of arrays. A tuple argument is passed, similar to the **vstack** function.

Code Listing 69

```
>>> x = np.arange(9).reshape(3,3)
>>> x

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

>>> y = 2 * x
>>> y

array([[ 0,  2,  4],
       [ 6,  8, 10],
       [12, 14, 16]])

>>> np.hstack((x,y))

array([[ 0,  1,  2,  0,  2,  4],
       [ 3,  4,  5,  6,  8, 10],
       [ 6,  7,  8, 12, 14, 16]])
```

Dstack

We can also perform depth-wise stacking using the **dstack()** function. Depth-wise stacking stacks a list of arrays along the third axis (this is the depth). For example:

Code Listing 70

```
>>> np.dstack((x,y))

array([[[ 0,  0],
        [ 1,  2],
        [ 2,  4]],

       [[ 3,  6],
        [ 4,  8],
        [ 5, 10]],

       [[ 6, 12],
        [ 7, 14],
        [ 8, 16]]])
```

column_stack

Stacking one-dimensional arrays can be done directly with the **column_stack()** function. For example:

Code Listing 71

```
>>> np.column_stack((x,y))

array([[ 0,  1,  2,  0,  2,  4],
       [ 3,  4,  5,  6,  8, 10],
       [ 6,  7,  8, 12, 14, 16]])
```

For two dimensional arrays, **column_stack()** performs the same as **hstack()**.

row_stack

Similarly, we can also perform row-wise stacking. This is best shown through example:

Code Listing 72

```
>>> np.row_stack((x,y))

array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 0,  2,  4],
       [ 6,  8, 10],
       [12, 14, 16]])
```

Concatenate

Finally, the most robust way to combine arrays is to use the concatenate function. For example:

Code Listing 73

```
>>> np.concatenate((x,y))

array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 0,  2,  4],
       [ 6,  8, 10],
       [12, 14, 16]])
```

Splitting Arrays

Arrays can be split horizontally, vertically, and depth-wise. The functions used are **hsplit(), vsplit(), dsplit(),** and **split().** The arrays can be split into arrays of the same shape or indicate the position after which the split should occur.

Horizontal Splitting

We can use **hsplit()** to split an array along some horizontal axis as well as specifying the amount of final pieces. For example:

Code Listing 74

```
>>> np.hsplit(x,3)

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

This is similar to using the **split()** function with an extra parameter.

Vertical Splitting

As expected we can use **vsplit()** to split along the vertical axis. For example:

Code Listing 75

```
>>> np.vsplit(x,3)

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

Depth-wise Splitting

Finally, the **dsplit()** performs depth-wise splitting. For example:

Code Listing 76

```
>>> x = np.arange(8).reshape(2,2,2)
>>> np.dsplit(x,2)

[array([[[0],
         [2]],

        [[4],
         [6]]]),
array([[[1],
         [3]],

        [[5],
         [7]]])]
```

Next we will learn how to search, and sort elements in an array efficiently with built-in methods and functions in NumPy!

Chapter 3 Searching and Sorting

Some of the most important of the built-in Numpy functions are those that have to do with searching and sorting through matrices and arrays. Numpy's built-in functions are written to maximize efficiency in this category.

Searching

Numpy's ability to quickly search through arrays and matrices is one of its biggest strengths. Let's take a look at how to use some built-in functions to quickly find elements in an array.

Argmax and Argmin

If you are seeking the index of either the maximum or minimum elements of an array, you can use the built-in functions **argmin** and **argmax.** If there are multiple occurrences of the max or min value, then the indices return corresponds to the first occurrence. Let's see an example:

Code Listing 77

```
>>> arr = np.arange(9).reshape(3,3)
>>> arr

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

>>> np.argmax(arr)
8
>>> np.argmin(arr)
0

>>> np.argmax(arr, axis=0)
array([2, 2, 2])

>>> np.argmin(arr, axis=1)
array([0, 0, 0])

>>> arr = np.arange(9)
>>> arr[1] = 8
>>> arr
array([0, 8, 2, 3, 4, 5, 6, 7, 8])

>>>np.argmax(arr)
1
```

Nanargmax and Nanargmin

Numpy also provides NaA aware versions of **argmax** and **argmin** with the **nanargmax** and **nanargmin** functions, which ignore NaN values. For example:

Code Listing 78

```
>>> arr = np.array([np.nan,5,10])
>>> np.argmin(arr)
1

>>> np.nanargmin(arr)
0
```

Argwhere

By default the **argwhere** function will return all indices of elements greater than zero in an array. You can also pass a condition as you can with the **where** function. For more examples on what is meant by a condition and how to implement it, check the section covering the **where** function later on in this chapter. For examples of **argwhere**, see below:

Code Listing 79

```
>>> arr = np.arange(9).reshape(3,3)
>>> arr
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

>>> np.argwhere(arr) # Index of elements greater than zero
array([[0, 1],
       [0, 2],
       [1, 0],
       [1, 1],
       [1, 2],
       [2, 0],
       [2, 1],
       [2, 2]])

>>> np.argwhere(arr>5) # Index of elements greater than 5
array([[2, 0],
       [2, 1],
       [2, 2]])

>>> np.argwhere(arr<5)
```

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

Nonzero

The **nonzero** function will return all the indices of elements in an array which are not zero. This is easily seen through example:

Code Listing 80

```
>>> arr = np.eye(3)
>>> arr

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

>>> np.nonzero(arr)
(array([0, 1, 2]), array([0, 1, 2]))
```

Flatnonzero

Unlike the **nonzero** function, the **flatnonzero** function will return indices that are non-zero in the flattened version of an array. For example:

Code Listing 81

```
>>> arr = np.arange(-2, 3)
>>> arr
array([-2, -1,  0,  1,  2])

>>> np.flatnonzero(arr)
array([0, 1, 3, 4])
```

Where

The **where** function is an extremely useful function in Numpy and is commonly used in many situations. The **where** function will return the elements from an array that fit a condition. Conditions can be inserted as an entire Boolean array of matching size or a Boolean array that can be broadcast across the secondary input array. If inputting a Boolean sequence as a condition, two arrays must be placed as arguments along with the condition. The **where** statement will then pick values from the first array if the condition of the Boolean at the matching element index is True, otherwise it will return the element form the second input array. This is best shown through example:

Code Listing 82

```
>>> boo = [True,False],[False,True]
>>> arr = np.arange(4).reshape(2,2)
>>> arr2 = arr*10
>>> np.where(boo,arr,arr2)

array([[ 0, 10],
       [20,  3]])
```

The condition can also just be a comparison operator statement on the entire array, in this case the **where** function will return the elements where this condition is true. Let's see some examples of the various ways to pass a condition to an array:

Code Listing 83

```
>>> x = np.arange(9.).reshape(3, 3)
>>> np.where( x > 5 )

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

>>> x[np.where( x > 3.0 )]

array([ 4.,  5.,  6.,  7.,  8.])

>>> np.where(x < 5, x, -1)

array([[ 0.,  1.,  2.],
       [ 3.,  4., -1.],
       [-1., -1., -1.]])
```

Searchsorted

The **searchsorted** function is an interesting and useful tool to know about in Numpy. The function takes in an array and an array of values to be inserted into the first array. The function then finds indices where elements should be inserted to maintain order. For example:

Code Listing 84

```
>>> np.searchsorted([1,2,3,4,5,6], 3)
2

>>> np.searchsorted([1,2,3,4,5], [-10, 10, 2, 3])
array([0, 5, 1, 2])
```

Extract

The **extract** function will return the elements of an array that satisfy a specified condition. We can imagine this as calling **array[condition]** for when **condition** is a Boolean array. For example:

Code Listing 85

```
>>> arr = np.arange(16).reshape((4, 4))
>>> arr
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])


>>> condition = np.mod(arr, 3)==0

>>> np.extract(condition,arr)
array([ 0,  3,  6,  9, 12, 15])

>>> arr[condition]
array([ 0,  3,  6,  9, 12, 15])
```

Sorting

In this section we will learn about the various functions available in Numpy to sort an array.

Sort

The most basic function for sorting an array is the **sort** function. The function can take an input of an array and sort it along a specified axis. The function can also take in a **kind** argument which will define what sorting algorithm to use. The default sorting algorithm is quick sort, but ‘mergesort’ and ‘heapsort’ are also possible arguments to pass.

Numpy has three sorting algorithms available for the **sort** function. You can check Table 2 for a description of each from the documentation.

Table 2: Sorting Algorithms

Kind	Speed	Worst Case Order	Work Apace	Stable
‘quicksort’	1	O(n^2)	0	no
‘mergesort’	2	O(n*log(n))	~n/2	yes
‘heapsort’	3	O(n*log(n))	0	No
Source: http://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.sort.html				

Let’s see an example:

Code Listing 86

```
>>> arr = np.array([[1,5],[4,1]])
>>> np.sort(arr)           # Sort on last axis by default
array([[1, 5],
       [1, 4]])

>>> np.sort(arr,axis=None) # Sorting on the flattened array
array([1, 1, 4, 5])

>>> np.sort(arr,axis=0)    # Sorting by the first axis
array([[1, 1],
       [4, 5]])
```

Lexsort

We can use the **lexsort** function to perform an indirect sort using a sequence of keys. In order to use **lexsort** you need a **k** number of **N** shaped sequences.

The best way to explain this is by example, for instance, consider the code below:

Code Listing 87

```
>>> arr1 = [1,2,1,3,2,1] # First column
>>> arr2 = [1,2,3,4,5,6] # Second column
>>> ind = np.lexsort((b,a)) # Sort by arr1, then by arr2
>>> [(arr1[i],arr2[i]) for i in ind]

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

Note how we sort by **arr1** first, and then by **arr2**, as shown by calling the indices in the list comprehension. Note that the last tuple pair corresponds with the last sorted element of **arr1**, so that you get (3,4).

Argsort

Another interesting function in Numpy is **argsort** which returns the indices that would sort an array. So instead of getting the actual elements, you can later on use this index array for masking (which we will learn about in more detail later). For example:

Code Listing 88

```
>>> arr = np.array([99,55,77,11])
>>> np.argsort(arr)

array([3, 1, 2, 0]

>>> arr = np.array([[88,99,55],[66,77,11]])
>>> np.argsort(arr)

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

The general concept of using "arg" to return an index is a common technique that you will see appear in many functions in Numpy.

Sort_complex

This function will sort a complex array using the real part first, then the imaginary part of the array. For example:

Code Listing 89

```
>>> np.sort_complex([1 + 2j, 2 - 1j, 3 - 2j, 3 - 3j, 3 + 5j])

array([ 1.+2.j,  2.-1.j,  3.-3.j,  3.-2.j,  3.+5.j])
```

Partition

The **partition** function will take an input array and a parameter kth. The kth value of the element will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. For example:

Code Listing 90

```
>>> a = np.array([7, 6, 5, 1])
>>> np.partition(a, 3)
```

```
array([5, 1, 6, 7])
```

Argpartition

The **argpartition** takes in the same arguments as the **partition** function, however it will perform it returns the necessary indices instead of the values.

Code Listing 91

```
>>> a = np.array([7, 6, 5, 1])
>>> np.argpartition(a, 3)
array([2, 3, 1, 0])
```

That’s it for sorting and searching! Hopefully you found quite a few of these function potentially very useful! Let’s move on to learning about universal functions!

Chapter 4 Universal Functions (ufunc)

Universal functions, commonly known as **ufunc**, are functions that will operate on an array element by element. This means that universal functions are able to support array broadcasting and type casting. We can think of universal functions as a normal function that has been “vecotrized”. This means that the function will take in some fixed number of scalar inputs and then also output some fixed number of scalar outputs.

Universal functions range from basic arithmetic all the way to trigonmetric functions. They provide a nice way for you to perform operations and broadcast them through your array. There are currently over 60 universal functions in Numpy.

Many of these universal functions are actually called internally by Numpy. This means that when you type something like **arr1 + arr2**, NumPy is actually calling the universal function **add(arr1,arr2).** You might want to keep this in mind when writing larger pieces of code which utilize Numpy. The universal functions are also useful to call when creating code that will be flexible to various possible inputs.

In this section we will describe the most important universal functions to know with small examples of each of them. Many of these functions operate in a straight-forward manner.

Unless otherwise stated, the arrays **arr1** and **arr2** will be defined by the code below:

Code Listing 92

```
>>> arr1 = np.arange(9.0).reshape((3, 3))
>>> arr1
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.]])

>>> arr2 = np.arange(3.0)
>>> arr2
array([ 0.,  1.,  2.])
```

Math Functions

This section will cover the basic mathematic universal functions in Numpy. Many of which are used internally for array operations.

Addition, Subtraction, Multiplication

The basic mathematic operations are self-explanatory and can be seen by these basic examples (note to keep in mind the references to **arr1** and **arr2** that we stated previously):

Code Listing 93

```
>>> np.add(2.0, 3.0)

5.0

>>> np.add(arr1, arr2)
array([[  0.,   2.,   4.],
       [  3.,   5.,   7.],
       [  6.,   8.,  10.]])

>>> np.subtract(1.0, 4.0)
-3.0

>>> np.subtract(arr1, arr2)
array([[ 0.,  0.,  0.],
       [ 3.,  3.,  3.],
       [ 6.,  6.,  6.]])

>>> np.multiply(2.0, 3.0)
6.0

>>> np.multiply(arr1, arr2)
array([[  0.,   1.,   4.],
       [  0.,   4.,  10.],
       [  0.,   7.,  16.]])
```

Division

The division universal function must be discussed in a little more detail. For normal array operations, you can expect the appearance of NaN and Inf terms when division by zero (or zero being divided by a number). For example:

Code Listing 94

```
>>> np.divide(1.0, 2.0)
0.5

>>> np.divide(arr1, arr2)
array([[ NaN,  1. ,  1. ],
       [ Inf,  4. ,  2.5],
       [ Inf,  7. ,  4. ]])
```

It's also important to note that integer division will perform classic division, not true division. You must specify floats in order to perform true division. Keep in mind these examples may change depending on whether you are using Python 2 or Python 3. Python 3 performs true division always. For an example of np.divide with use of floating point numbers versus integers check the code below:

Code Listing 95

```
>>> np.divide(1, 2)
0

>>> np.divide(1, 2.)
0.5
```

You won't have to worry about division by zero raising a warning, it will always yield zero in integer arithmetic (note: that this is for Python 2).

Code Listing 96

```
>>> np.divide(np.array([0, 1], dtype=int), np.array([0, 0], dtype=int))

array([0, 0])
```

If you're writing a program where you would want to catch a division by zero, you can use the **seterr** function. This function will set how floating-point errors are handled in Numpy (for all code). It can be used as the following example where we try to divide 2 by 0:

Code Listing 97

```
>>> old_err_state = np.seterr(divide='raise')
>>> np.divide(2, 0)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FloatingPointError: divide by zero encountered in divide
```

Or to use the old error state and ignore the error:

Code Listing 98

```
>>> ignored_states = np.seterr(**old_err_state)
>>> np.divide(2, 0)
0
```

True_divide and Floor_divide

In situations where you want to specify the type of division being used, you can explicitly use the **true_divide** and **floor_divide** functions to control the type of division being done. It should be noted that depending on what version of Python you are using (2.x vs 3.x) floating point numbers in division may not be an issue to be concerned with. Let's see a quick example of both of these functions:

Code Listing 99

```
>>> arr = np.arange(10)
>>> np.true_divide(arr,2)
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5])

>>> np.floor_divide(arr,2)
array([0, 0, 1, 1, 2, 2, 3, 3, 4, 4])
```

Logaddexp

The **logaddexp** function will return the logarithm of the sum of exponentiations of the inputs. This is the same as **log(exp(arr1) + exp(arr2))**. You might use this function when trying to calculate small numbers. For example, if performing a probability calculation where a sequence of events is to occur and the individual probability of each event is low, then then probability of them all occurring will be a very small number. You could then use **logaddexp** to not have to worry about floating point numbers and then use the logarithm of the calculated probability.

For example:

Code Listing 100

```
>>> p_1 = np.log(2e-50)
>>> p_2 = np.log(3e-50)
>>> p_1and2 = np.logaddexp(p_1, p_2)
>>> p_1and2

-113.51981673726819
```

Remainder and Mod

If you want to get the remainder of a division operation, you can use either the **mod** or **remainder** functions. The both compute the same thing, so there is no difference between them. They will perform the operation element wise, for example:

Code Listing 101

```
>>> np.remainder(np.arange(10), 3)
array([0, 1, 2, 0, 1, 2, 0, 1, 2, 0])

>>> np.mod(np.arange(10), 3)
array([0, 1, 2, 0, 1, 2, 0, 1, 2, 0])
```

Absolute

In order to get the absolute value of all the elements in an array, you can use the **absolute** function. For example:

Code Listing 102

```
>>> arr = np.array([-1, 1])
>>> np.absolute(arr)
array([ 1,  1])
```

Rint

If you have an array of floating point numbers and want to round them to the nearest integer, you can use the **rint** function. For example, notice the rounding up of 5.5 below:

Code Listing 103

```
>>> arr = np.array([0.1,1.1,2.2,3.3,4.4,5.5,6.6])
>>> np.rint(arr)
array([ 0.,  1.,  2.,  3.,  4.,  6.,  7.])
```

Sign

If you want to check the sign (negative versus positive) of elements in your array, you can use the **sign** function. It will return -1 for negative numbers and 1 for positive numbers. Note that it returns 0 for zero. For example:

Code Listing 104

```
>>> np.sign([-2., 2.])
array([-1.,  1.])

>>> np.sign(0)
0
```

Exp

If you want to calculate the exponential for all the elements of an array, you can use the **exp** function. This will perform **exp(element)** for all the elements in the array (this is known as *e* or Euler’s number).

Code Listing 105

```
>>> np.exp(np.arange(10))
array([  1.00000000e+00,   2.71828183e+00,   7.38905610e+00,
         2.00855369e+01,   5.45981500e+01,   1.48413159e+02,
         4.03428793e+02,   1.09663316e+03,   2.98095799e+03,
         8.10308393e+03])
```

Exp2

The **exp2** function will calculate **2**x** for all **x** in the array, which is useful when describing objects that follow a **2**x** pattern, such as memory amounts. For example:

Code Listing 106

```
>>> np.exp2(np.arange(9))
array([   1.,    2.,    4.,    8.,   16.,   32.,   64.,  128.,  256.])
```

Log, log2, and log10

In case you want to calculate and broadcast logarithmic functions of your array, Numpy has universal functions for that. The **log** function calculates the natural logarithm, the **log2** function returns the base-2 logarithm, and the **log10** function returns the base 10 logarithm. All of these functions are performed element wise across the input array. Let’s see some examples:

Code Listing 107

```
>>> arr = np.arange(5)
>>> np.log(arr)
array([       -inf,  0.        ,  0.69314718,  1.09861229,  1.38629436])

>>> np.log2(arr)
array([      -inf,  0.       ,  1.       ,  1.5849625,  2.       ])

>>> np.log10(arr)
array([       -inf,  0.        ,  0.30103   ,  0.47712125,  0.60205999])
```

Keep in mind here that an array containing 0 will result in –inf, but Numpy will warn the user if this occurs. You will usually use these Numpy functions when dealing with mathematical functions that have *e* (Euler’s number) occurring in them.

Sqrt and Square

The **sqrt** and **square** functions can be used to compute square roots and squares. Note that you could also just broadcast ****2** or ****0.5.** Let's see an example of the square and square root functions:

Code Listing 108

```
>>> arr = np.arange(10)
>>> np.sqrt(arr)
array([ 0.        ,  1.        ,  1.41421356,  1.73205081,  2.        ,
        2.23606798,  2.44948974,  2.64575131,  2.82842712,  3.        ])

>>> np.square(arr)
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

Reciprocal

As you've probably guessed, the **reciprocal** function returns the reciprocal of each element in the input array (1/element). For example:

Code Listing 109

```
>>> np.reciprocal(10.)
0.1
```

Trigonometric Functions

Numpy's universal functions also come equipped with a variety of trigonometric use cases. These can come in handy when working with arrays describing real-life geometry. Let's get a quick review of the most common trigonometric universal functions in Numpy.

Sin, Cosine, and Tangent

The **sin,cos,** and **tan** functions can be used to return the trigonometric results of each element in the array. For example (Note the use of radians here):

Code Listing 110

```
>>> np.sin(np.array((0., 30., 45., 60., 90.)) * np.pi / 180. )
array([ 0.        ,  0.5       ,  0.70710678,  0.8660254 ,  1.        ])
```

```
>>> np.cos(np.array([0, np.pi/2, np.pi]))
array([  1.00000000e+00,   6.12303177e-17,  -1.00000000e+00])

>>> np.tan(np.array([-np.pi,np.pi/2,np.pi]))
array([  1.22460635e-16,   1.63317787e+16,  -1.22460635e-16])
```

Arcsin, Arcos, and Arctan

Similarly, there are inverse versions of the trigonometric functions as well, defined by the use of the prefix **arc**. For example:

Code Listing 111

```
>>> np.arcsin(1)     # pi/2
1.5707963267948966

>>> np.arccos([1, -1])
array([ 0.        ,  3.14159265])

>>> np.arctan([0, 1])
array([ 0.        ,  0.78539816])
```

Hypot

An interesting trigonometric function in Numpy is the **hypot** function. This function returns the hypotenuse of a right triangle and takes in the other two sides as an argument. For example, given a triangle with the sides 3 and 4:

Code Listing 112

```
>>> np.hypot(3,4)
5.0
```

Sinh, Cosh, and Tanh

If you want to calculate the hyperbolic trigonometric functions, you can do it easily with NumPy. The function names are the trigonometric functions with the attached suffix "**h**". For example:

Code Listing 113

```
>>> np.sinh(0)
0.0

>>> np.cosh(0)
1.0

>>> np.tanh(np.pi)
0.99627207622074998
```

Arcsinh, Arccosh, and Arctanh

The inverse hyperbolic trigonometric functions are also available. For example:

Code Listing 114

```
>>> np.arcsinh(np.array([np.e, 10.0]))
array([ 1.72538256,  2.99822295])

>>> np.arccosh([np.e, 10.0])
array([ 1.65745445,  2.99322285])

>>> np.arctanh([0, -0.5])
array([ 0.        , -0.54930614])
```

Angle Conversion

Numpy comes equipped to convert angles in an array from radians to degrees or vice versa. The two functions are **deg2rad** and **rad2deg.** Let's see a quick example:

Code Listing 115

```
>>> np.deg2rad(180)

3.1415926535897931

>>> np.rad2deg(np.pi/2)

90.0
```

Comparison Functions

In this section we will briefly go over the comparison operators that operate on an element by element basis in Numpy. It is important to note, do not use the Python keywords **and** and **or** to combine logical array expressions. These keywords will test the truth value of the entire array (not element-by-element as you might expect). Use the bitwise operators & and | instead. Let's see a variety of the built-in functions that serve as comparison operators:

Basic Comparison Operators

When comparing regular Python objects, you can use the basic operators such as **<,>,<=, >=.** However in Numpy, since the array objects can hold multiple elements it makes more sense to use the built-in comparison operators which return boolean arrays for element by element comparisons. Here are a few examples of the basic element by element comparison operators:

Code Listing 116

```
>>> np.greater([4,2],[2,2])
array([ True, False], dtype=bool)

>>> np.greater_equal([4, 2, 1], [2, 2, 2])
array([ True, True, False], dtype=bool)

>>> np.less([1, 2], [2, 2])
array([ True, False], dtype=bool)

>>> np.less_equal([4, 2, 1], [2, 2, 2])
array([False,  True,  True], dtype=bool)

>>> np.not_equal([1.,2.], [1., 3.])
array([False,  True], dtype=bool)

>>> np.equal([0, 1, 3], np.arange(3))
array([ True,  True, False], dtype=bool)
```

Next we'll look at the variety of floating functions in Numpy.

Floating Functions

Boolean Array Checks

Numpy also comes equipped with a variety of Boolean array floating functions for checking for number types. They begin with the letter **is** and the check if an array contains certain element types. The names of the functions are self-explanatory. Consider the examples below for a quick explanation of each one:

Code Listing 117

```
>>> >>> np.isreal([1+1j, 1+0j])
array([False,  True], dtype=bool)

>>> np.iscomplex([1+1j, 1+0j])
array([ True, False], dtype=bool)

>>> np.isinf([np.inf, -np.inf, 100.0])
array([ True,  True, False], dtype=bool)

>>> np.isnan(np.nan)
True
>>> np.isnan(np.inf)
False

>>> #Various examples of checking isfinite
>>> np.isfinite(1)
True
>>> np.isfinite(0)
True
>>> np.isfinite(np.nan)
False
>>> np.isfinite(np.inf)
False
>>> np.isfinite(np.NINF)
False
```

Chapter 5 Statistics

NumPy provides various functions and methods specifically designed for statistics. In this chapter we will describe the variety of functions available and show examples of them on arrays.

For the variance examples below, the array objects **arr1d** and **arr2d** will refer to the arrays described in the code below (unless stated otherwise).

Code Listing 118

```
>>> arr1d = np.arange(10)
>>> arr2d = np.arange(16).reshape(4,4)

>>> arr1d

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

>>> arr2d

array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Now let’s begin with a through breakdown of the various functions available in NumPy concerning statistics!

Averages and Variance

Some of the most common statistical properties needed from an array revolve around averages and variance of the elements in an array. Let’s take a look at the various functions available.

Median

The median is the number separating the higher half of a data sample. The **median** function takes an array and an axis as arguments. It then returns the median of the array along that axis. It can accept both 1-d and 2-d arrays. For example:

Code Listing 119

```
>>> np.median(arr1d)
4.5

>>> np.median(arr2d,axis=0)
```

```
array([ 6.,  7.,  8.,  9.])

>>> np.median(arr2d,axis=1)
array([  1.5,   5.5,   9.5,  13.5])
```

Average

Just like the **median** function the **average** function takes in an array and an axis. However it can also take in an array of weights to compute a weighted average. The weights argument is passed as an array of weights associated with each element in the given array. For example:

Code Listing 120

```
>>> np.average(arr1d)
4.5

>>> np.average(arr1d,weights=np.arange(10))
6.33
```

Mean

As you probably suspected, the **mean** function will calculate the mean of a given array along a specified axis. Note that the previous function showed, **average** could take a weights argument, unlike **mean.** However, an axis can be specified for higher dimension arrays. Let's see a simple example:

Code Listing 121

```
>>> np.mean(arr1d)

4.5
```

It should be noted that for this particular arrange array the mean and median are the same.

Standard Deviation

The **std** function can take an array and an axis as arguments and will return the standard deviation along that axis of the array. It should be noted that if no axis is provided then the function will flatten the array before calculating the standard deviation. For example:

Code Listing 122

```
>>> np.std(arr1d)
2.8722813232690143

>>> np.std(arr2d,axis=1)
array([ 1.11803399,  1.11803399,  1.11803399,  1.11803399])
```

Variance

The **var** function will return the variance, which is equal to the square of the standard deviation. It operates in a similar manner as the **std** function.

Code Listing 123

```
>>> np.var(arr1d)
8.25

>>> np.var(arr2d,axis=1)
array([ 1.25,  1.25,  1.25,  1.25])
```

Dealing with NaN

If you find yourself dealing with arrays with Nan (or Null) values, Numpy has a series of “nan” statistics functions you can use which will ignore any Nan data points. They operate in the same fashion as their normal counterparts. Let’s create an array with a Nan element and use the nan statistics functions to obtain information from the array.:

Code Listing 124

```
>>> arr =np.array([1,np.nan,3])
```

Nanmean and Nanmedian

The **nan** versions of the mean and median functions:

Code Listing 125

```
>>> np.nanmean(arr)

2.0

>>> np.nanmedian(arr)

2.0
```

Nanstd

The **nan** version of the standard deviation function:

Code Listing 126

```
>>> np.std(arr)

1.0
```

Nanvar

The nan version of the variance function:

Code Listing 127

```
>>> np.nanvar(arr)

1.0
```

Correlation

Correlation allows us to see the dependence between two or more sets of data. Correlations are useful in their ability to indicate possible predictive relationships between sets of data.

Corrcoef

The **corrcoef** function returns the correlation coefficient. The correlation coefficient is a way of quantitatively measuring relationship between two or more random variables. In Numpy, the correlation coefficient for two arrays is calculated by Pearson correlation coefficient. For more info on the Pearson correlation coefficient, check out the Wikipedia article on the topic at: https://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient

For an example of calculating this in NumPy:

Code Listing 128

```
>>> arr1 = np.array([1,2,3,3,2,1])
>>> arr2 = np.array([4,5,6,4,5,6])
>>> np.corrcoef(arr1,arr2)

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

Correlate

The **correlate** function will perform a cross-correlation of two arrays, note that the arrays both ened to be one-dimensional. For more information on the formula used, check out: https://en.wikipedia.org/wiki/Cross-correlation.

For an example of how to use **correlate:**

Code Listing 129

```
>>> arr1 = np.array([1,2,3,3,2,1])

>>> arr2 = np.array([4,5,6,4,5,6])

>>> np.correlate(arr1,arr2)

array([60])
```

Cov

The **cov** function returns the covariance of the matrix given. Covariance indicates the level to which two variables vary together. If the greater values of one variable mainly correspond with the greater values of the other variable, and the same holds for the smaller values the covariance is positive. If the reverse of this is true, than the covariance of the arrays is negative. For example, determining the covariance of two 1-dimensional arrays with Numpy:

Code Listing 130

```
>>> arr1 = np.array([1,2,3,3,2,1])
>>> arr2 = np.array([4,5,6,4,5,6])
>>> np.cov(arr1,arr2)

array([[ 0.8,  0. ],
       [ 0. ,  0.8]])
```

Another important note is how the covariance matrix can appear when combining two arrays. For example:

Code Listing 131

```
>>> a1 = [-2,1,4]
>>> a2 = [3,1,0.5]
>>> A = np.vstack((a1,a2))
>>> np.cov(A)

array([[ 9.   , -3.75],
       [-3.75,  1.75]])

>>> np.cov(a1, a2)
```

```

array([[ 9.  , -3.75],
       [-3.75,  1.75]])

>>>np.cov(a1)

array(9.0)
```

Calling a covariance function on two arrays is the same as calling on the vertical stack of both of those arrays. Next let’s learn about order statistics!

Order Statistics

Order statistics are functions that have to do with the ranking of the elements in an array, such as finding maximum or minimum of an array. There are a few key functions in this category in Numpy.

Amin and Amax

The **amin** and **amax** functions take an array and an optional axis. They then return the minimum and maximum element of the array, respectively. For example, without and with axis calls:

Code Listing 132

```
>>> arr = np.array([[1,2],[3,4]])
>>> arr
array([[1, 2],
       [3, 4]])

>>> np.amin(arr)
1

>>> np.amax(arr)
4

>>> np.amax(arr,axis=0)
array([3,4])
```

Nanmin and nanmax

The **nanmin** and **nanmax** functions will ignore and NaN values that occur in the array when taking the minimum and maximum. For example, an array with **np.nan** without using **amin** and then using **nanmin**:

Code Listing 133

```
>>> arr = np.array([[1,2,np.nan],[np.nan,3,4]])
>>> np.amin(arr)
Nan

>>> np.nanmin(arr)
1.0
```

Peak to Peak (ptp)

The peak to peak function for Numpy is **ptp**. Peak to peak is the range of values of an array, meaning the maximum element minus the minimum element. The function can also take a specified axis argument. If a multidimensional array is put in, then it returns an array of the peak to peak values. For example:

Code Listing 134

```
>>> arr = np.arange(4).reshape((2,2))
>>> arr

array([[0, 1],
       [2, 3]])

>>> np.ptp(arr, axis=0)
array([2, 2])

>>> np.ptp(arr, axis=1)
array([1, 1])
```

Percentile

A percentile is a measure in statistics indicating the value below which a given percentage of observations in a group of observations fall. For example the 10th percentile is the value below which 10 percent of the observations may be found. The percentile function in Numpy is **percentile** and accepts an array and a parameter **q** which defines what percentile to compute (it must be between 0 and 100 inclusive). This function can also take an axis argument to specify along which axis to perform the percentile operation. Let's see it in action:

Code Listing 135

```
>>> arr = np.array([[10,9,6], [5,3,1]])
>>> arr

array([[10,  9,  6],
       [ 5,  3,  1]])

>>> np.percentile(arr, 50)
5.5

>>> np.percentile(arr, 50, axis=0)
array([ 7.5,  6. ,  3.5])

>>> np.percentile(arr, 50, axis=1)
array([ 9.,  3.])
```

Histograms

Histogram

The **histogram** function in Numpy takes in an array argument as well as a **bins, range,** and **weights** argument. The histogram is computed over the flattened array. The bins argument is a list of where all but the last bin is half-open. For example if a bin is **[0,1,2,3]** then the first bin will be **[0,1)** (which means including 0, but excluding 1). The second bin would be **[1,2)**. The last bin is then **[2,3]**, which includes 3. Keep this in mind when passing your bins argument; otherwise you may unexpectedly cut off your last bin.

The range argument can be used to set a lower and upper range of the bins, the default is simply the minimum and maximum of the array.

Code Listing 136

```
>>> np.histogram([0,1,1,2,2,2,2],bins=[0,1,2,3])

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

Bincount

The **bincount** function in Numpy will return an array of count the number of occurrences of each value in an array of (non-negative) integers. The bin counts in the returned array will correspond to the index value. This means that the array returned will have a minimum length of the largest value of the input array. Let's see a quick example:

Code Listing 137

```
>>> np.bincount(np.arange(10))
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])

>>> np.bincount(np.array([0,1,2,1,1,3,4,4,5,7,8]))
array([1, 3, 1, 1, 2, 1, 0, 1, 1])
```

So to further explain, note that in the second **bincount** operation since the integer 1 showed up 3 times, then the output array has the integer 3 at index 1. This is a nice format to keep as your counts can now be directly called through indexing.

Chapter 6 Linear Algebra

One of the main reasons for NumPy's creation was for the ability to perform efficient linear algebra techniques. The linear algebra library for NumPy (**numpy.linalg**) is an extremely powerful tool for performing efficient analysis on arrays. In this section we will go over the key functions of this library. For a full list of functions available in the linear algebra library, you can check out: http://docs.scipy.org/doc/numpy-1.10.0/reference/routines.linalg.html.

Matrix and Vector Products

To start off, Numpy comes equipped to handle a variety of matrix product operations. Let's see a few examples of the key functions in this category:

Dot and Vdot

Let's begin with the **dot** function, which returns the dot product of two arrays. The dot product will take two equal-length sequences of numbers (in this case Numpy arrays) and returns a single scalar value. We can define the dot product either algebraically or geometrically. Algebraically, it is the sum of the products of the corresponding entries of the two sequences of numbers. Geometrically, it is the product of the Euclidean magnitudes of the two vectors and the cosine of the angle between them.

In terms of NumPy array operations, the dot product is equal to the following code (as defined in the NumPy documentation):

Code Listing 138

```
dot(arr1, arr2)[x,y,u,v] = sum(arr1[x,y,:] * arr2[u,:,v])
```

Now let's see various examples of using the **dot** function:

Code Listing 139

```
>>> np.dot(2, 4)
8

>>> arr1 = [[2, 0], [0, 2]]
>>> arr2 = [[10, 1], [4, 4]]
>>> np.dot(arr1, arr2)
array([[20,  2],
       [ 8,  8]])

>>> arr1 = np.arange(12).reshape((3,4))
>>> arr2 = np.arange(12)[::-1].reshape((4,3))
>>> np.dot(arr1, arr2)
array([[ 24,  18,  12],
       [128, 106,  84],
       [232, 194, 156]])
```

Let's now discuss the **vdot** function. We use **vdot** when checking for the dot products of two vectors. It's important to note that the **vdot** function will handle complex numbers differently than the **dot** function. If the first argument is complex, the complex conjugate of the first argument is used for the calculation of the dot product.

More importantly, you should take into account that the **vdot** function will flatten multi-dimensional inputs before performing the dot product. Because of this, **vdot** should only be used for 1-dimensional vectors. Let's see an example of this:

Code Listing 140

```
>>> arr1 = np.array([[1, 2], [3, 4]])
>>> arr2 = np.array([[5, 6], [7, 8]])
>>> np.vdot(arr1, arr2)
70
```

Inner and Outer

We also have options for specifying inner and outer dot products for 1-dimenstional arrays. Using the **inner** function is the same as performing the following code:

Code Listing 141

```
>>> np.inner(arr1, arr2) == sum(arr1[:]*arr2[:])
True
```

Now let's see an example of the inner product with some 1-dimensional arrays:

Code Listing 142

```
>>> arr1 = np.array([1,2,3])
>>> arr2 = np.array([0,1,0])
>>> np.inner(arr1, arr2)
2
```

The outer product of two vectors is best explained by an example utilizing strings to show the nature of the multiplication:

Code Listing 143

```
>>> arr1 = np.array(['x', 'y', 'z'], dtype=object)
>>> np.outer(arr1, [1, 2, 3])

array([['x', 'xx', 'xxx'],
       ['y', 'yy', 'yyy'],
       ['z', 'zz', 'zzz']], dtype=object)
```

Tensordot

A tensor is a multidimensional array and the **tensordot** function can compute the tensor dot product of two tensors. The mathematics behind this sort of operation are beyond the scope of this book, but you can find more resources on this topic at Wikipedia here: https://en.wikipedia.org/wiki/Tensor and here: https://en.wikipedia.org/wiki/Dyadics.

The **tensordot** function will take in two arrays as arguments and then an **axes** parameter. The tensor product function then returns the sum of the products of the respective elements in each array over the specified axes argument. These common use cases of **tensordot** are best shown through example.

Code Listing 144

```
>>> x = np.arange(18.).reshape(2,3,3)
>>> y = np.arange(24.).reshape(3,2,4)
>>> z = np.tensordot(x,y, axes=([1,0],[0,1]))
>>> z.shape

(3, 4)

>>> z

array([[ 600.,  645.,  690.,  735.],
       [ 660.,  711.,  762.,  813.],
       [ 720.,  777.,  834.,  891.]])
```

Kron

More advanced users of linear algebra may find the need to perform a Kronecker product, which results in a composite array made of blocks of the second array scaled to the first. Unfamiliar users can refer to the Wikipedia article on this topic for more information on the mathematics behind it, available here: https://en.wikipedia.org/wiki/Kronecker_product. In a general sense, the function assumes that the number of dimensions of **arr1** and **arr2** are the same, and if necessary prepending with the smallest ones. For example, if **arr1.shape = (a0,a1,…aN)** and **arr2.shape = (b0,b1,…bN)** then the Kronecker product has a shape of **(a0*b,a1*b1,…,aN*bN).** The elements of the resulting array are then products formed by the following method (as defined in the Numpy documentation):

Code Listing 145

```
kron(arr1,arr2)[k0,k1,...,kN] = arr1[i0,i1,...,iN] * arr2[j0,j1,...,jN]

where

kt = it * st + jt,  t = 0,...,N

In the common 2-D case (N=1), the block structure can be visualized:

[[ arr1[0,0]*arr2,   arr1[0,1]*arr2,  ... , arr1[0,-1]*arr2  ],
 [  ...                              ...   ],
 [ arr1[-1,0]*arr2,  arr1[-1,1]*arr2, ... , arr1[-1,-1]*arr2 ]]
```

Let’s see some examples of the function in action:

Code Listing 146

```
>>> np.kron([1,10,100], [4,6,8])

array([  4,   6,   8,  40,  60,  80, 400, 600, 800])

>>> np.kron( [4,6,8],[1,10,100])

array([  4,  40, 400,   6,  60, 600,   8,  80, 800])

>>> np.kron(np.eye(2), np.ones((2,2)))
array([[ 1.,  1.,  0.,  0.],
       [ 1.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  1.],
       [ 0.,  0.,  1.,  1.]])
```

Now let’s move on to a fundamental topic in linear algebra, Eigenvalues!

Eigenvalues

For the following topics it will be assumed that the linear algebra library in Numpy has been imported with the following syntax:

Code Listing 147

```
>>> from numpy import linalg as LA
```

Let's begin with a quick review of what an eigenvalue is. A number **w** is an eigenvalue of **a** if there exists a vector **v** such that **dot(a,v) = w *v**. In this way, the arrays, **a,w,** and **v** all satisfy the equations **dot(a[:,:],v[:,i] = w[i] * v[:,i]**.

For a full explanation of Eigenvalues and Eigenvectors, I recommend you check out the Wikipedia article on this topic, available here: https://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors.

Alright now let's see some example of how to use the linear algebra library in Numpy to find eigenvalues and eigenvectors.

Square and Symmetric Matrices

Let's see a simple use case of using the **linalg.eig** function, which returns the computed eigenvalues and right eigenvectors of a square array. For example:

Code Listing 148

```
>>> w, v = LA.eig(np.diag((2, 4, 8)))
>>> w
array([ 2.,  4.,  8.])


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

We can use the linear algebra function **linalg.eigh** to get the eigenvalues and eigenvectors of a symmetric matrix. This function returns two objects, a 1-dimensional array containing the eigenvalues of the input matrix, and a 2-dimensional square array (or matrix depending on input) of the corresponding eigenvectors (in columns). For example:

Code Listing 149

```
>>> arr = np.array([[1, -2j], [2j, 5]])
>>> arr
array([[ 1.+0.j,  0.-2.j],
       [ 0.+2.j,  5.+0.j]])
>>> w, v = LA.eigh(arr)
>>> w
array([ 0.17157288,  5.82842712])

>>> v
array([[-0.92387953+0.j        , -0.38268343+0.j        ],
       [ 0.00000000+0.38268343j,  0.00000000-0.92387953j]])
```

Matrix Properties

In this section we will discuss built-in functions for obtaining various properties of a matrix.

Matrix or Vector Norm

The built-in function **linalg.norm** in Numpy is able to return one of eight different norms, or one of an infinite number of vector norms (depending on the parameter **ord**). The following matrix norms can be calculated:

Table 3: Ord Parameter Options

ord	norm for matrices	norm for vectors
None	Frobenius norm	2-norm
'fro'	Frobenius norm	–
'nuc'	nuclear norm	–
inf	max(sum(abs(x), axis=1))	max(abs(x))
-inf	min(sum(abs(x), axis=1))	min(abs(x))
0	–	sum(x != 0)
1	max(sum(abs(x), axis=0))	as below

ord	**norm for matrices**	**norm for vectors**
-1	min(sum(abs(x), axis=0))	as below
2	2-norm (largest sing. value)	as below
-2	smallest singular value	as below
other	–	sum(abs(x)**ord)**(1./ord)
Source: http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.linalg.norm.html#numpy.linalg.norm		

Let's now look at various examples form the NumPy documentation (http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.linalg.norm.html#numpy.linalg.norm) on the utilizing the **norm** function:

Code Listing 150

```
>>> arr1 = np.arange(9) - 4
>>> arr1
array([-4, -3, -2, -1,  0,  1,  2,  3,  4])

>>> arr2 = arr1.reshape((3, 3))
>>> arr2
array([[-4, -3, -2],
       [-1,  0,  1],
       [ 2,  3,  4]])

>>> LA.norm(arr1)
7.745966692414834

>>> LA.norm(arr2)
7.745966692414834

>>> LA.norm(arr2, 'fro')
7.745966692414834

>>> LA.norm(arr1, np.inf)
4

>>> LA.norm(arr2, np.inf)
9

>>> LA.norm(arr1, -np.inf)
0

>>> LA.norm(arr2, -np.inf)
```

```
2

>>> LA.norm(arr1, 1)
20

>>> LA.norm(arr2, 1)
7

>>> LA.norm(arr1, -1)
-4.6566128774142013e-010

>>> LA.norm(arr2, -1)
6

>>> LA.norm(arr1, 2)
7.745966692414834

>>> LA.norm(arr2, 2)
7.3484692283495345

>>> LA.norm(arr1, -2)
nan

>>> LA.norm(arr2, -2)
1.8570331885190563e-016

>>> LA.norm(arr1, 3)
5.8480354764257312

>>> LA.norm(arr1, -3)
nan
```

Condition

Similar to the norm function, the **linalg.condition** function is able to return the condition number of a matrix, using one of seven norms. The condition number of a matrix X is defined as the norm of X times the norm of the inverse of X. We can also think of this as the ratio of the largest to smallest singular value in the singular value decomposition of a matrix. For more information on condition numbers and how they apply to matrices, check out the Wikipedia article on the topic here: https://en.wikipedia.org/wiki/Condition_number. For the **linalg.condition** function, the parameter p defines the norm used. The table below shows the various options for **p** and the resulting norm:

Table 4: p parameter options

p	norm for matrices
None	2-norm, computed directly using the SVD
'fro'	Frobenius norm
inf	max(sum(abs(x), axis=1))
-inf	min(sum(abs(x), axis=1))
1	max(sum(abs(x), axis=0))
-1	min(sum(abs(x), axis=0))
2	2-norm (largest sing. value)
-2	smallest singular value
Source: http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.linalg.cond.html#numpy.linalg.cond	

Let's now see some examples of the **linalg.cond** function in use:

Code Listing 151

```
>>> arr1 = np.array([[1, 0, -1], [0, 1, 0], [1, 0, 1]])
>>> arr1
array([[ 1,  0, -1],
       [ 0,  1,  0],
       [ 1,  0,  1]])

>>> LA.cond(arr1)
1.4142135623730951

>>> LA.cond(arr1, 'fro')
3.1622776601683795

>>> LA.cond(arr1, np.inf)
2.0

>>> LA.cond(arr1, -np.inf)
1.0

>>> LA.cond(arr1, 1)
2.0

>>> LA.cond(arr1, -1)
```

```
1.0

>>> LA.cond(arr1, 2)
1.4142135623730951

>>> LA.cond(arr1, -2)
0.70710678118654746
```

Determinant

A determinant is a useful value that can be computed from the elements of a square matrix. It is often useful when a matrix is used to represent the coefficients in a system of linear equations; the determinant can be used to solve those equations (although there are more efficient ways). Let's see a simple example of the **det** function used to determine a determinant:

```
>>> arr1 = np.array([[2, 4], [6, 8]])
>>> np.linalg.det(arr1)
-8.0
```

Rank

To determine the rank of a matrix, Numpy uses the singular value decomposition method (SVD). This method is the most time consuming, but the most reliable method. For more information on calculating matrix rank, check out the Wikipedia article here: https://en.wikipedia.org/wiki/Rank_(linear_algebra). Let's see some simple examples of using **matrix_rank** form the **linalg** library:

Code Listing 152

```
>>> from numpy.linalg import matrix_rank
>>> matrix_rank(np.eye(4)) # Full rank matrix
4

>>> I=np.eye(4); I[-1,-1] = 0. # rank deficient matrix
>>> matrix_rank(I)
3

>>> matrix_rank(np.ones((4,))) # 1 dimension - rank 1 unless all 0
1

>>> matrix_rank(np.zeros((4,)))
0
```

Trace

The **trace** function is a useful tool for linear algebra calculations. This function returns the sum along the diagonals of the array. If the input array is two dimensional the sum along its diagonal is returned (there is an optional offset parameter). Let's see a simple example of **trace:**

Code Listing 153

```
>>> np.trace(np.eye(4))
4.0
```

It's important to note here that the **trace** function is actually not part of the **linalg** library.

Matrix Equations and Inversions

In this section we will discuss using Numpy to solve systems of equations set in matrix form, as well as functions that can help with inversing matrices (a significant part of solving these equations).

Solve

The **solve** function in Numpy will allow us to solve a linear matrix equation, or a system of linear scalar equations. This functions computes and "exact" solution **X** of the well-determined (full rank) linear matrix equation **AX=B**. Let's see a simple example to solve the following system of equations **6 * x0 + 2*x1 = 18** and **2*x0 + 4 * x1 = 16**:

Code Listing 154

```
>>> eq1 = np.array([[6,2], [2,4]])
>>> eq2 = np.array([18,16])
>>> ans = np.linalg.solve(eq1, eq2)
>>> ans

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

Tensor Solve

Similarly to the **solve** function, there is a tensor equivalent. Note the use of ***** in the code below to create the multi-dimensional tensors. For example:

Code Listing 155

```
>>> ten1 = np.eye(2*3*4)
>>> ten1.shape = (2*3, 4, 2, 3, 4)
>>> ten2 = np.random.randn(2*3, 4)
>>> ans = np.linalg.tensorsolve(ten1, ten2)
>>> ans.shape
(2, 3, 4)
```

Least Squares

The least squares method is a common method used to solve a linear matrix equation. It is especially use din simple linear regression models. For example we can attempt to fit a line **y=mx+b** through some data points. For example:

Code Listing 156

```
>>> x = np.array([0, 1, 2, 3])
>>> y = np.array([-1, 0.2, 0.9, 2.1])
```

By examining the coefficients, we see that the line should have a slope of approximately 1 and have a y-intercept at around -1.

We can rewrite the line equation as **y = Ap**, where **A = [[x 1]]** and **p = [[m], [c]].** Now we can use lstsq to solve for *p*:

Code Listing 157

```
>>> A = np.vstack([x, np.ones(len(x))]).T
>>> A
array([[ 0.,  1.],
       [ 1.,  1.],
       [ 2.,  1.],
       [ 3.,  1.]])

>>> m, b = np.linalg.lstsq(A, y)[0]
>>> print m, b
1.0 -0.95
```

Inverse

To compute the inverse of an array (the multiplicative inverse), we can use the **linalg.inv** function. By definition an inverse of a matrix produces an identity matrix when multiplied by the original matrix. For example:

Code Listing 158

```
>>> arr = np.array([[2,3],[4,5]])
>>> arr
array([[2, 3],
       [4, 5]])


>>> arrinv= np.linalg.inv(arr)
>>> arrinv
array([[-2.5,  1.5],
       [ 2. , -1. ]])

>>> np.dot(arr,arrinv)
array([[ 1.,  0.],
       [ 0.,  1.]])
```

Up next, we will learn about using Numpy for testing!

Chapter 7 Testing

One of the most important developments in software engineering has been the rise of test driven development (TDD). TDD utilizes automated unit testing in order to automatically test new code and make sure functionality is preserved. In this chapter we will be reviewing the built-in methods for unit testing in NumPy. Let's begin by looking at **assert** statement functions.

Assert Functions

The unit tests used by TDD usually use assert statements to make sure that the results of particular functions have not changed. A common issue with assert statements is the use of floating point numbers. Due to round off error, floating point number results may not always be the same (e.g. 0.000000003 != 0. 000000002). Numpy's **testing** package comes with assert functions which are prepared for this issue.

Throughout this section it will be important to take note of the AssertionErrors that arise and check that they match your understanding of what the functions are checking for.

Assert Functions

Let's get a quick overview of the variety of **assert** functions in Numpy's testing package.

Table 5: Assert Functions

Function	**Description**
assert_almost_equal()	Raises an exception if two numbers are not equal up to a specified precision
assert_approx_equal()	Raises an exception if two numbers are not equal up to a certain significance
assert_array_almost_ equal()	Raise an exception if two arrays are not equal up to a specified precision
assert_array_equal()	Raises an exception if two arrays are not equal.
assert_array_less()	Raises an exception if two arrays do not have the same shape, and the elements of the first array are strictly less than the elements of the second array
assert_equal()	Raises an exception if two objects are not equal

assert_raises()	Fails if a specified exception is not raised by a callable invoked with defined arguments
assert_warns()	Fails if a specified warning is not thrown
assert_string_equal()	Asserts that two strings are equal
assert_allclose()	Raises an assertion if two objects are not equal up to desired tolerance
Source: http://docs.scipy.org/doc/numpy-1.10.0/reference/routines.testing.html	

Let's now explore some of these functions in greater depth.

Asserting almost equal

Let's call the **assert_almost_equal** function on two floating point numbers with the accuracy up to 7 decimal places:

Code Listing 159

```
>>> print np.testing.assert_almost_equal(0.123456789,0.123456780, decima
l=7)
None
```

Note how no exception is raised and a None is returned. If we call it with higher precision:

Code Listing 160

```
>>> print np.testing.assert_almost_equal(0.123456789,0.123456780, decima
l=8)

--------------------------------------------------------------------
AssertionError                      Traceback (most recent call last)
<ipython-input-28-4b4a64ebf6f0> in <module>()
----> 1 print np.testing.assert_almost_equal(0.123456789,0.123456780, de
cimal=8)
      2 None

/opt/conda/envs/python2/lib/python2.7/site-packages/numpy/testing/utils.
pyc in assert_almost_equal(actual, desired, decimal, err_msg, verbose)
    488         pass
    489     if round(abs(desired - actual), decimal) != 0 :
--> 490         raise AssertionError(_build_err_msg())
    491
    492

AssertionError:
Arrays are not almost equal to 8 decimals
```

```
 ACTUAL: 0.123456789
 DESIRED: 0.12345678
```

Now we get an exception raised we can act upon in a testing situation.

Approximately equal arrays

The **assert_approx_equal** function will raise an exception if two numbers are not equal up to a certain number of significant digits. Let's use the function on the previous numbers:

Code Listing 161

```
>>> print np.testing.assert_approx_equal(0.123456789, 0.123456780,significa
nt=8)
None

>>> np.testing.assert_approx_equal(0.123456789, 0.123456780,significant=9)
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-32-9f1c39f24b6f> in <module>()
----> 1 np.testing.assert_approx_equal(0.123456789, 0.123456780,significant
=9)

/opt/conda/envs/python2/lib/python2.7/site-packages/numpy/testing/utils.pyc
in assert_approx_equal(actual, desired, significant, err_msg, verbose)
    585         pass
    586     if np.abs(sc_desired - sc_actual) >= np.power(10., -(significan
t-1)) :
--> 587         raise AssertionError(msg)
    588
    589 def assert_array_compare(comparison, x, y, err_msg='', verbose=True
,

AssertionError:
Items are not equal to 9 significant digits:
 ACTUAL: 0.123456789
 DESIRED: 0.12345678
```

Almost equal arrays

The **assert_equal_almost_equal** function will raise an exception if two arrays are not equal up to a specified precision. The function checks if the two arrays have the same shape first. Then the values of the arrays are compared element by element. For example:

Code Listing 162

```
>>> arr1 = np.array([0,0.123456789])
>>> arr2 = np.array([0, 0.123456780])
>>> print np.testing.assert_array_almost_equal(arr1,arr2 , decimal=8)
None

>>> np.testing.assert_array_almost_equal(arr1,arr2 , decimal=9)

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-37-f3fd68af8971> in <module>()
----> 1 np.testing.assert_array_almost_equal(arr1,arr2 , decimal=9)

/opt/conda/envs/python2/lib/python2.7/site-packages/numpy/testing/utils.pyc
in assert_array_almost_equal(x, y, decimal, err_msg, verbose)
    840     assert_array_compare(compare, x, y, err_msg=err_msg, verbose=ve
rbose,
    841              header=('Arrays are not almost equal to %d decimals' %
decimal),
--> 842              precision=decimal)
    843
    844

/opt/conda/envs/python2/lib/python2.7/site-packages/numpy/testing/utils.pyc
in assert_array_compare(comparison, x, y, err_msg, verbose, header, precisi
on)
    663                                 names=('x', 'y'), precision=precisi
on)
    664             if not cond :
--> 665                 raise AssertionError(msg)
    666     except ValueError as e:
    667         import traceback

AssertionError:
Arrays are not almost equal to 9 decimals

(mismatch 50.0%)
 x: array([ 0.         ,  0.123456789])
 y: array([ 0.        ,  0.12345678])
```

Equal Arrays

The **assert_array_equal** function will raise an exception if two arrays are not equal. This means that the shapes must be equal and the elements of each array must be equal. It should be noted that with this function NaN values are allowed. For example:

Code Listing 163

```
>>> arr1 = np.array([0,0.123456789])
>>> arr2 = np.array([0, 0.123456780])
>>> print np.testing.assert_array_equal(arr1,arr2)

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-38-602eed749eaf> in <module>()
----> 1 np.testing.assert_array_equal(arr1,arr2)

/opt/conda/envs/python2/lib/python2.7/site-packages/numpy/testing/utils.pyc
in assert_array_equal(x, y, err_msg, verbose)
    737     """
    738     assert_array_compare(operator.__eq__, x, y, err_msg=err_msg,
--> 739                          verbose=verbose, header='Arrays are not eq
ual')
    740
    741 def assert_array_almost_equal(x, y, decimal=6, err_msg='', verbose=
True):

/opt/conda/envs/python2/lib/python2.7/site-packages/numpy/testing/utils.pyc
in assert_array_compare(comparison, x, y, err_msg, verbose, header, precisi
on)
    663                                 names=('x', 'y'), precision=precisi
on)
    664             if not cond :
--> 665                 raise AssertionError(msg)
    666     except ValueError as e:
    667         import traceback

AssertionError:
Arrays are not equal

(mismatch 50.0%)
 x: array([ 0.      ,  0.123457])
 y: array([ 0.      ,  0.123457])
```

Ordering Arrays

An interesting function for testing arrays is the **assert_array_less** function which will raise an exception if the two arrays do not have the same shape, and the elements of the first array are strictly less than the elements of the second array. For example:

Code Listing 164

```
>>> np.testing.assert_array_less([0, 0.123456789,np.nan], [1, 0.23456780, n
p.nan])
None

>>> np.testing.assert_array_less([0, 0.123456789,np.nan], [0, 0.123456780,
np.nan])

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-41-9230a9086666> in <module>()
----> 1 np.testing.assert_array_less([0, 0.123456789,np.nan], [0, 0.1234567
80, np.nan])

/opt/conda/envs/python2/lib/python2.7/site-packages/numpy/testing/utils.pyc
in assert_array_less(x, y, err_msg, verbose)
    911     assert_array_compare(operator.__lt__, x, y, err_msg=err_msg,
    912                          verbose=verbose,
--> 913                          header='Arrays are not less-ordered')
    914 
    915 def runstring(astr, dict):

/opt/conda/envs/python2/lib/python2.7/site-packages/numpy/testing/utils.pyc
in assert_array_compare(comparison, x, y, err_msg, verbose, header, precisi
on)
    663                                 names=('x', 'y'), precision=precisi
on)
    664             if not cond :
--> 665                 raise AssertionError(msg)
    666     except ValueError as e:
    667         import traceback

AssertionError: 
Arrays are not less-ordered

(mismatch 100.0%)
 x: array([ 0.      ,  0.123457,       nan])
 y: array([ 0.      ,  0.123457,       nan])
```

Comparing Objects

For testing outside of Numpy objects we can use the **assert_equal** function, which will raise an exception if two Python objects are not equal. These objects can be Numpy arrays, as well as lists, tuples, or dictionaries. For example:

Code Listing 165

```
>>> np.testing.assert_equal((1, 2), (1, 1))

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-42-d1a18b66d5c7> in <module>()
----> 1 np.testing.assert_equal((1, 2), (1, 1))

/opt/conda/envs/python2/lib/python2.7/site-packages/numpy/testing/utils.pyc
in assert_equal(actual, desired, err_msg, verbose)
    268         assert_equal(len(actual), len(desired), err_msg, verbose)
    269         for k in range(len(desired)):
--> 270             assert_equal(actual[k], desired[k], 'item=%r\n%s' % (k,
err_msg), verbose)
    271         return
    272     from numpy.core import ndarray, isscalar, signbit

/opt/conda/envs/python2/lib/python2.7/site-packages/numpy/testing/utils.pyc
in assert_equal(actual, desired, err_msg, verbose)
    332     # Explicitly use __eq__ for comparison, ticket #2552
    333     if not (desired == actual):
--> 334         raise AssertionError(msg)
    335 
    336 def print_assert_equal(test_string, actual, desired):

AssertionError:
Items are not equal:
item=1

 ACTUAL: 2
 DESIRED: 1
```

Comparing Strings

The **assert_string_equal** function will check if two strings are equal, an interesting note is that the exception will return the difference between the two strings. For example:

Code Listing 166

```
>>> np.testing.assert_string_equal("Num", "Numpy")

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-49-4f42ea07990f> in <module>()
----> 1 np.testing.assert_string_equal("Num", "Numpy")

/opt/conda/envs/python2/lib/python2.7/site-packages/numpy/testing/utils.pyc
in assert_string_equal(actual, desired)
```

```
    981     msg = 'Differences in strings:\n%s' % (''.join(diff_list)).rstr
ip()
    982     if actual != desired :
--> 983         raise AssertionError(msg)
    984
    985

AssertionError: Differences in strings:
- Num+ Numpy?    ++
```

Comparing with Floating-points

As previously mentioned, the full representation of floating point numbers in a computer is not exact due to round off error. This could possibly lead to issues when trying to compare floating point numbers for testing purposes. There are two functions in the NumPy's testing library designed to deal with this: **assert_array_almost_equal_nulp** and **assert_array_max_ulp**. The use of ULP stands for Unit of Least Precision. This is related to the IEEE specification for floating point numbers, where half ULP precision is required for arithmetic operations.

Another important term is the machine epsilon, which is the largest relative rounding error in floating-point arithmetic. The Numpy function **finfo** allows a user to determine the machine epsilon. Let's see some examples of these functions in action:

Code Listing 167

```
>>> mac_eps = np.finfo(float).eps
>>> print mac_eps
2.220446049250313e-16

>>> np.testing.assert_array_almost_equal_nulp(1.0, 1.0 + mac_eps)
None

>>> np.testing.assert_array_almost_equal_nulp(1.0, 1.0 + 2 * mac_eps)

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-56-9fb7521598c3> in <module>()
      1 mac_eps = np.finfo(float).eps
----> 2 print np.testing.assert_array_almost_equal_nulp(1.0, 1.0 + 2 * mac_
eps)

/opt/conda/envs/python2/lib/python2.7/site-packages/numpy/testing/utils.pyc
in assert_array_almost_equal_nulp(x, y, nulp)
   1356             max_nulp = np.max(nulp_diff(x, y))
   1357             msg = "X and Y are not equal to %d ULP (max is %g)" % (
nulp, max_nulp)
-> 1358         raise AssertionError(msg)
```

```
   1359
   1360 def assert_array_max_ulp(a, b, maxulp=1, dtype=None):

AssertionError: X and Y are not equal to 1 ULP (max is 2)
```

Comparing floats with more ULPs

The **assert_array_max_ulp** function will allow the user to specify the upper bound for the number of ULPs allowed. The **maxulp** parameter accepts an integer value for the limit.

Code Listing 168

```
>>> mac_eps = np.finfo(float).eps
>>> print mac_eps
2.2204460492503131e-16

>>> np.testing.assert_array_max_ulp(1.0, 1.0 + mac_eps)
1.0
>>> np.testing.assert_array_max_ulp(1.0, 1 + 2 * mac_eps,maxulp=2)
2.0
```

Nose Test Decorators

The **nose** framework for Python allows for easier unit-testing and maintains organization of your testing code. The framework utilizes decorators extensively. The **numpy.testing** module actually has a number of decorators built-in for use with nose tests. This section will simply list and briefly describe the available decorators and assume knowledge on how to implement with nose tests. For more information on the **nose** framework, visit: https://nose.readthedocs.org/en/latest/.

Table 6: Decorator Descriptions for Testing

Decorator	**Description**
numpy.testing.decorators.deprecated	This function filters deprecation warnings when running tests
numpy.testing.decorators. knownfailureif	This function raises KnownFailureTest exception based on a condition
numpy.testing.decorators.setastest	This decorator marks a function as being a test or not being a test
numpy.testing.decorators.skipif	This function raises a SkipTest exception based on a condition
numpy.testing.decorators.slow	This function labels test functions or methods as slow
Source: http://docs.scipy.org/doc/numpy-1.10.0/reference/routines.testing.html	

This concludes the testing section of the book. Let's move on by combining everything we've learned with a project analyzing stocks from the stock market!

Chapter 8 Stock Analysis Project

In this section we will perform a basic analysis of a technology stock while reviewing the various techniques covered throughout the book. In many of the following sections the necessary code will be too extensive to put in line by line into IDLE, thus most of the code will assume it has been saved in a .py file for execution. This formatting will be clear by the lack of "**>>>**" notation. If you notice that there is a presence of **>>>** then the code can be run simply in IDLE assuming that the previous .py files have been run and the variable names have stayed in memory. It will be clear in the code blocks what is intended to be a larger .py file and what are simple function commands in IDLE. Let's now begin analyzing stock data!

Reading the Data

To begin our analysis, we will first need some stock data to analyze. We will be using the Pandas library to grab the stock data directly with Python. The Pandas library is a fantastic data analysis tool built on top of NumPy and I highly encourage you to check it out if you've enjoyed what you have learned about NumPy. We will be analyzing the Ford Motor Company, stock ticker **F.** You can also download a .csv file from Yahoo finance here: http://finance.yahoo.com/q?s=F. Here is a sample script which will download the stock information using python (assuming Pandas has been installed).

Code Listing 169

```
import pandas.io.data as web
import datetime

start = datetime.datetime(2010, 1, 1)
end = datetime.datetime(2013, 1, 27)
f = web.DataReader("F", 'yahoo', start, end) #Ford Motor Company
f.to_csv('Ford Stock Data')
```

The script will then save the data as a .csv file called "Ford Stock Data.csv". Let's now load the closing price and volume data into Numpy using the following:

Code Listing 170

```
import numpy as np
c,v = np.loadtxt('Ford Stock Data',delimiter=',',usecols=(4,5),
unpack=True,skiprows=1)
```

Since we were dealing with a CSV file we note the delimiter as a comma, and we also specify which columns to use, in this case the closing price and volume. We also set the unpack parameter equal to True, which allows us to use tuple unpacking to simultaneously assign the objects c and v. Finally, we specify to skip the starting row, which is the column name.

Averaging the Stock Price

Let’s begin our analysis by looking at various stock average price metrics. We will begin by looking at the Volume Weighted Price Average, known as the VWAP. This metric characterizes an average price for the option. Typically, the higher the volume of shares traded, the more significant a price move is. Let’s go ahead and calculate the VWAP for the shares. We can do this by utilizing the **average** function in Numpy and by defining the weights parameter by the volume of shares traded that day. For example:

Code Listing 171

```
>>> vwap = np.average(c, weights=v)
>>> print vwap
12.701641737336889
```

We can also simply use the **mean** function to calculate a standard mean of the closing price. For example:

Code Listing 172

```
>>> print np.mean(c)
12.3967963671
```

Another common average price metric is the time-weighted average price (TWAP). The reasoning behind this metric is that more recent price quotes are more important than older prices. There are a few ways of creating the weights based on a time series of a stock, one of the simpler ways would be to simply use **arange()** to create a weight array. For example:

```
>>> time = np.arange(len(c))
>>> np.average(c,weights=time)
11.911782235278858
```

It should be noted that this method of creating the weights array is an oversimplification of a time-weighted average.

Range of Values

When performing an analysis, it’s important to understand the full range of values, not just the averages. Since we already have information of the high and low prices of the stock, we can use this to find the highest and lowest price of that stock ever (for the given time period). Let’s see how we can do this with Numpy:

Code Listing 173

```
high,low=np.loadtxt('Ford Stock Data', delimiter=',', usecols=(2,3),
skiprows=1,unpack=True)
print 'Highest price ever was: ',np.max(high)
print 'Lowest price ever was: ',np.min(low)
```

```
#OUTPUT
Highest price ever was:  18.969999
Lowest price ever was:  8.82
```

We could also check the spread of the values with the peak to peak function, **ptp**. For example:

Code Listing 174

```
print "High price spread: ", np.ptp(high)
print "Low price spread: ", np.ptp(low)

#OUTPUT

High price spread:  9.939999
Low price spread:  9.790001
```

Let's now explore developing some more statistics around the stock price.

Stock Statistics

When evaluating stocks, the most interesting aspect is the closing price and an analyst's ability to attempt to predict that price. Intuitively we can suspect that the closing price should be somewhere around the mean price. However, outliers can occur which can skew the averages and diminish their ability to be accurate predictors. One way to attempt at finding some "average" without worrying about outliers is to find the median value instead of the mean. We can do this with Numpy, for example:

Code Listing 175

```
>>> print 'Median of closing price was: ',np.median(c)

Median of closing price was:  11.99
```

We can also take a look at the variance of the stock, to see how much it has fluctuated over the years. For example:

Code Listing 176

```
>>> print 'Variance of stock was: ', np.var(c)

Variance of stock was:  4.5029055565
```

Similarly, we can check the stock's standard deviation:

Code Listing 177

```
>>> print 'STD of stock was: ', np.std(c)

STD of stock was:  2.12200507928
```

Now let's explore the stock returns. A common analysis technique for stocks is to analyze returns and the logarithmic returns of the closing price. The logarithmic returns are determined by taking the log of all the prices and then calculating the differences between them. They measure the rate of change. The formula can be described by the following equation: **log(price2)-log(price1) = log(price2/price1).**

We can use the built-in Numpy function **diff** to return an array which calculates the n-th order discrete difference along a given axis. Using this on the stock price will be similar to finding the derivative of the price with respect to time. We can then get the returns by dividing by the value from the previous day. For example:

Code Listing 178

```
>>> returns = np.diff(c)/c[:-1]
```

Now we can compute the standard deviation of the average returns:

Code Listing 179

```
>>> np.std(returns)

0.021916143537167306
```

We can also compare the number of days that had a positive return versus those that had a negative return. We can do this with the **where** function. For example:

Code Listing 180

```
>>> pos = len(np.where(returns > 0)[0])
>>> neg = len(np.where(returns < 0)[0])

>>> print 'Number of days with positive return: ',pos
>>> print 'Number of days with negative return: ',neg


Number of days with positive return:  382
Number of days with negative return:  370
```

Looks like positive returns had a slightly higher number of occurrences than negative return days.

Let's now continue with the logarithmic returns:

Code Listing 181

```
logret = np.diff(np.log(c))
```

Note that because stock prices cannot be negative, we do not have to check or remove any negative or zero values (otherwise the stock would not be listed). We'll use the logarithmic returns to calculate volatility.

Volatility measures the variation in price of the stock option. We can calculate historical volatility from the historical price data of a stock. Let's take a look at annual and monthly volatility. Annual volatility can be calculated by dividing the ratio of the standard deviation of the logarithmic returns and the mean of those returns by the square root of the reciprocal of business days in a year (typically 252). Let's see an example for clarity:

Code Listing 182

```
>>> annual_vol = (np.std(logret)/np.mean(logret) )/ np.sqrt(1./252.)
>>> print annual_vol

940.00130794
```

We can calculate monthly volatility in a similar way:

Code Listing 183

```
>>> month_vol = (np.std(logret)/np.mean(logret) )/ np.sqrt(1./12.)
>>> print month_vol

205.125102238
```

In the next section we will take a look at how to manage data with timestamps.

Time Stamps

Our stock data had timestamped information regarding the date of the stock price. This tiem series information can prove very useful for analysis. Let's use Numpy to grab the data:

Code Listing 184

```
>>> date,close = np.loadtxt('Ford Stock Data',delimiter=',',
usecols=(0,4),unpack=True,skiprows=1)

-----------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-48-0b22cd359dd6> in <module>()
----> 1 date,close = np.loadtxt('Ford Stock Data',delimiter=',',usecols=
(0,4),unpack=True,skiprows=1)

C:\Users\User\Anaconda\lib\site-packages\numpy\lib\npyio.pyc in loadtxt(
fname, dtype, comments, delimiter, converters, skiprows, usecols, unpack
, ndmin)
    858
    859                 # Convert each value according to its column and sto
re
--> 860                 items = [conv(val) for (conv, val) in zip(converters
, vals)]
    861                 # Then pack it according to the dtype's nesting
    862                 items = pack_items(items, packing)

ValueError: invalid literal for float(): 2010-01-04
```

Notice how we get an error regarding the date. This is because Numpy is designed to handle floating-point operations, not interpreting strings! In order to deal with this we will have to utilize the **converters** parameter in the **loadtxt** function. This parameter allows us to create user-defined functions for interpreting values in a text document. We will start by creating a function which parses the dates with the use of the datetime library. For example:

Code Listing 185

```
def date_convert(d):
    return datetime.datetime.strptime(d, "%Y-%m-%d").date().weekday()
```

This function will read in the string form of the date and convert it to a weekday using the **datetime** library included in Python. So for example:

Code Listing 186

```
>>> date_convert('2010-01-04')

2 # Monday = 0, Tuesday = 1, etc...
```

Let’s now use this function in conjunction with the loadtxt function:

Code Listing 187

```
date,close = np.loadtxt('Ford Stock Data',delimiter=',',usecols=(0,4),
unpack=True,skiprows=1,converters={0:date_convert})
```

Note how we use a dictionary like syntax for assigning the column number to a function for conversion. Let’s check what date looks like:

Code Listing 188

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

Let's now use this new information to examine stock price averages by day of the week. We'll start by creating an empty array to hold the price averages of each stock by the day of the week. Then we will create a function which cycles through the possible days and uses **where** to allocate the matching days and average their closing price.

Code Listing 189

```
day_avg = np.zeros(5) # empty matrix for each day of week

for i in range(5):
    index = np.where(date == i)
    prices = np.take(close, index)
    day_avg[i] = np.mean(prices)


print day_avg #print averages for the day

#OUTPUT
[ 12.40112676  12.39464968  12.42213834  12.43063698  12.33512819]
```

Based on the results it appears that the daily analysis is pretty uniform. Let's take a deeper look with some different average types.

Stock Averages

Average true range (known as ATR) is another indicator of volatility of a stock price. Let's now get a breakdown how to calculate it.

First, the ATR is based on the high and low price of a certain period, usually 20 days. Then for each day we calculate the daily range (the difference between the high and low price), the difference between the high and previous close, and the difference between the previous close and the low price. These three ranges allow us to know how much the stock price moved and how volatile it is.

We then need to find the maximum value for each day, we can use the **maximum** function to do this. After we have the maximum value for that day we can proceed with computing the average true range of values.

Code Listing 190

```
period = 20
```

```
h = high[-period:]
l = low[-period:]

prev_close = c[-period -1: -1]

true_range = np.maximum(h - l, h - prev_close, prev_close - l)

ATR = np.zeros(period)

ATR[0] = np.mean(true_range)

for i in range(1, period):
    ATR[i] = (period - 1) * ATR[i - 1] + true_range[i]
    ATR[i] /= period

print 'The ATR is: ', ATR

#OUTPUT




The ATR is:
[ 0.292       0.2954      0.29663     0.2982985   0.3158835  0.31308948
  0.30893493  0.30498818  0.30223877  0.31062683  0.30709549  0.30274072
  0.30810368  0.3046985   0.30096357  0.29391539  0.28871962  0.28578364
  0.27999446  0.27599474]
```

A common method of analyzing time-series data is through the use of a simple moving average (SMA). To calculate an SMA, a period of time is chosen to define a window of time, then we can calculate the mean of the data as we move that window of time through the entire time series.

A function that will be of great use in this scenario is the **convolve** function. This function returns a discrete, linear convolution. This is useful since in probability the sum of two independent random variables is distributed according to the convolution of their individual distributions. It will become clearer how to use the **convolve** function later on in the code example.

Code Listing 191

```
# Num of days in week
week = 5

# Get weights (0.2)
weights = np.ones(week) / week

# call convolve
```

```
close = np.loadtxt('Ford Stock Data',delimiter=',',usecols=(4,),unpack=T
rue,skiprows=1)

SMA = np.convolve(weights,close)[week-1:-week+1]
```

We can then plot this data against the closing price using the **Matplotlib** library. We won't go into explaining the **matplotlib** library into too much detail here, just know that it has been imported with the following statement:

Code Listing 192

```
>>> import matplotlib.pyplot as plt
```

For more information on the capabilities of **matplotlib**, please reference: http://matplotlib.org. Let's see an example piece of code which calculates the SMA:

Code Listing 193

```
# Plot the data against the normal closing price
time = np.arange(week - 1, len(close))
plt.figure(figsize = (10,8))
plt.plot(time, close[week-1:], lw=1.0, label="Closing Price")
plt.plot(time, SMA, 'r--', lw=2.0, label="SMA")

plt.title("Work Week Moving Average")
plt.xlabel("Days")
plt.ylabel("Price in Dollars")
plt.legend()
```

This code outputs the following figure:

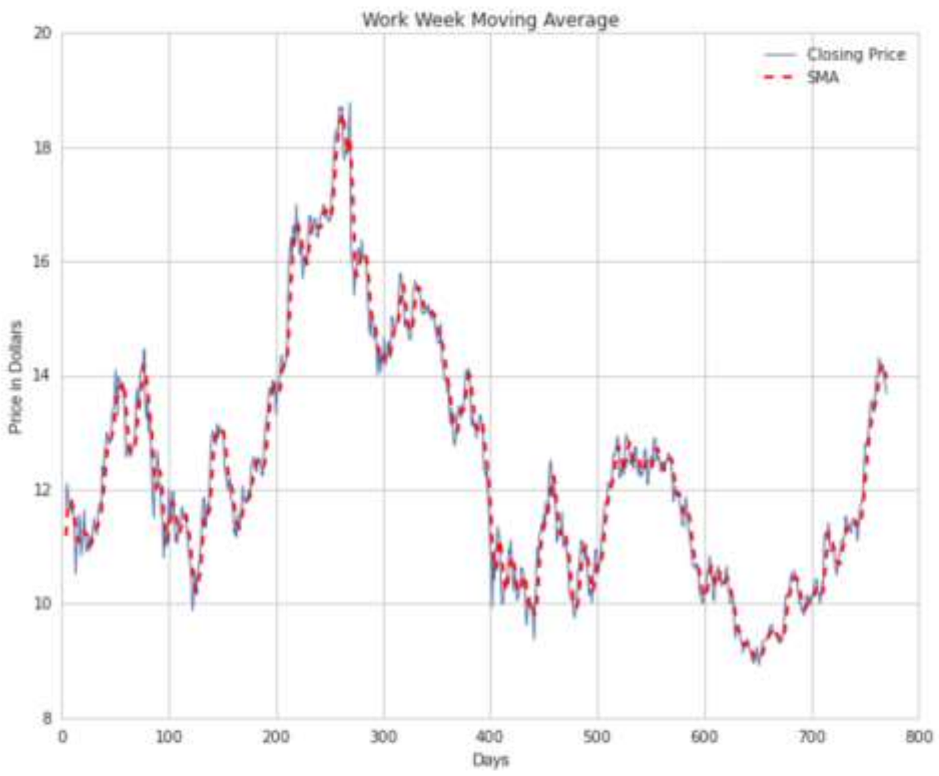


Figure 2: SMA for 800 Days

The effect of the SMA is too small to see on such a large time scale, so let's zoom in to a 3 week period:

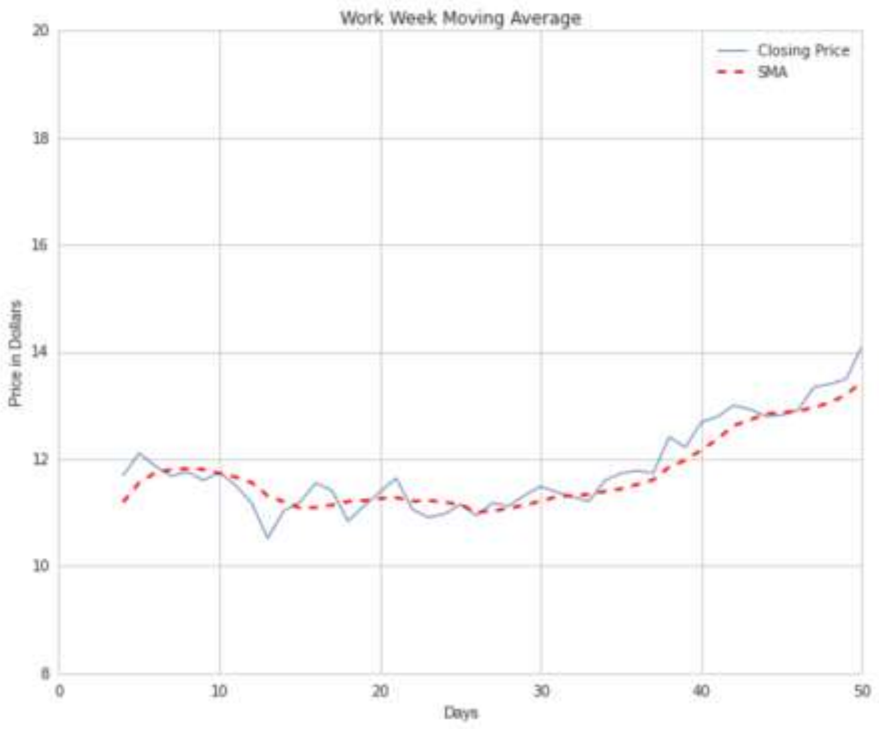


Figure 3: SMA for 50 days

Let’s now learn about the exponential moving average

The exponential moving average (EMA) is another alternative to the SMA. This method is distinguished by using exponentially decreasing weights. Let’s breakdown how to calculate the EMA:

Code Listing 194

```
days = 5

# Exponential weighting
weights = np.exp(np.linspace(-1,0,days))

#Normalize
weights /= weights.sum()

# call convolve
close = np.loadtxt('Ford Stock Data',delimiter=',',usecols=(4,),unpack=T
rue,skiprows=1)

EMA = np.convolve(weights,close)[days-1:-days+1]

# Plot the data against the normal closing price
time = np.arange(days - 1, len(close))
plt.figure(figsize = (10,8))
plt.plot(time, close[days-1:], lw=1.0, label="Closing Price")
plt.plot(time, EMA, 'r--', lw=2.0, label="EMA")
plt.xlim(0,100)
plt.title("Work Week Exponential Moving Average")
plt.xlabel("Days")
plt.ylabel("Price in Dollars")
plt.legend()
```

This produces the following plot:

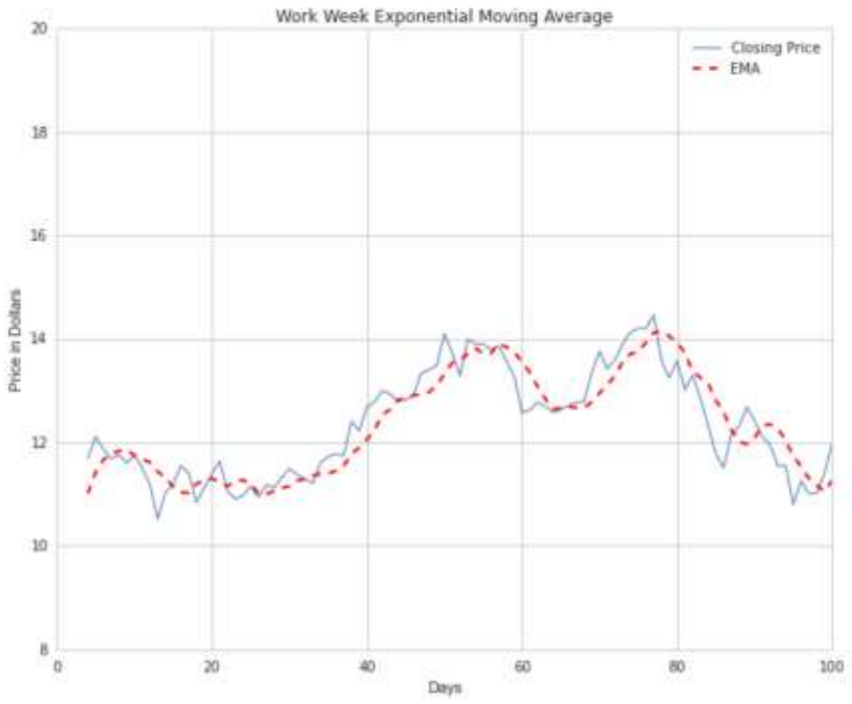


Figure 4: EMA for 100 days

Hopefully you've found this analysis interesting, and more importantly seen the incredible power of Numpy to quickly analyze and parse through complex data! That's it for all the code, practice, and definitions in this book! I hope you've enjoyed it and have a greater understanding of the amazing NumPy library!

Conclusion

Hopefully you've seen the amazing power of NumPy and have already made use of it for your own projects! Remember to use this book as a handy reference for examples of tools and functions within NumPy.

It's important to take note that many other libraries owe their success to Numpy, including the popular pandas library (which is built directly on top of Numpy). The Python programming language has begun to blossom as the de-facto programming language of data scientists and the Numpy library will continue to serve as a cornerstone of this Pydata ecosystem.