Table of Contents

1
Mathematical Computations

In algo trading using Python, all mathematical computation and models require us to have a rigorous scientific computing library. The answer is NumPy (Numerical Python), an open-source library for numerical computing. As an extension to the Python programming language, NumPy adds support for multi-dimensional arrays and matrices, along with a library of high-level mathematical functions to operate on these arrays. Meanwhile, the SciPy package further extends the functionality of NumPy with a collection of applied mathematical algorithms. Together, NumPy and SciPy are add-on modules to Python that provide common mathematical and numerical routines.

In addition, Pandas offers efficient DataFrame objects with indexing. Together, NumPy and Pandas are building blocks of Python scientific computing. This chapter empowers readers to use them to perform essential analytical operations including mathematical functions, array, and matrix computation, file I/O, and data frame manipulation.

Before diving into specific Python libraries, we would like to review a number of environments in which you write and run your Python code. Here are several commonly-used environments for Python:

- Python IDLE
- Jupyter Notebook
- Spyder
- PyCharm
- IntelliJ IDEA

Each Python environment has its pros and cons, and it will be up to the reader to decide which Python environment to use for running the examples in this book.

We assume readers already being familiar with Python basics. Therefore we proceed

directly to NumPy, which provides Python scientific computing routines.

NumPy - The "Free MATLAB"

NumPy is a library offering support for large, multi-dimensional arrays and matrices as well as high-level math functions in Python. Prior to 2005, the research community often had to turn to expensive proprietary software such as MATLAB to perform quant computation on arrays and matrices. With Python becoming widely accepted, NumPy has been rapidly gaining popularity among the scientific and engineering research community. Hence NumPy has earned a nickname as "Free MATLAB."

In fact, both MATLAB and NumPy use BLAS and LAPACK under the hood for efficient linear algebra calculations.

A Short History

As an open-source library, NumPy has many contributors who have been working hard to make NumPy what it is today. There have been several milestones in the history of NumPy -

- When Python started gaining attention in the scientific and engineering community in the 1990s, there was a growing need to have a flexible package to allow researchers to perform scientific computing on arrays and matrices
- In 1995, a special interest group was founded with a goal to create an array computing package including Guido Van Rossum, who is the "Father of Python." Other major contributors include Fulton, Hugunin, Dubois, Ascher, and Hinsen
- In 2005, Travis Oliphant created NumPy by unifying features of several smaller packages
- In 2006, NumPy 1.0 was released.
- In 2011, NumPy 1.5.0 was released (with support for Python 3)

Some researchers have been arguing that low-level language such as C++ is a more efficient choice than NumPy for scientific computing. It is true that NumPy gives up a certain amount of efficiency for significant development productivity gain, compared with C++. In practice, developer productivity often outweighs everything else, especially in fast iteration

and pivot to prove a concept. More importantly, hardware has become increasingly faster and cheaper over time. As a result, the performance gap between C++ and NumPy has been shrinking fast. This explains why NumPy has become the de-facto library for scientific computing among most of the quantitative researchers.

To avoid installing the large SciPy package just to get an array object, NumPy package was kept separate from SciPy.

N-Dimensional Array

The core functionality of NumPy is a data structure representing an N-dimensional array called ndarray. It is a multidimensional container of elements of the same type. Please note that ndarray applies the zero-based index.

If you are familiar with Python's built-in list data structure, you would remember that the Python list could hold items of mixed data types. In contrast, NumPy's ndarrays are homogeneously typed. That is, all elements of a single array must be of the same type. In other words, each element in ndarray is an object of data-type object (called dtype).

Here is a comparison between Python lists and NumPy arrays-

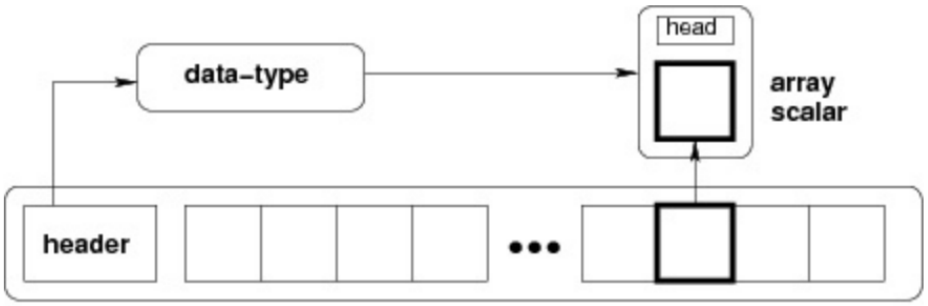
```
#Python lists allow for mixed data types
a = [1,2]
b = [[1,2],[3,4]]
c = [[1,2,'dog'],[3,[4]]]

#NumPy Arrays with all entries of the same data type
import numpy as np
x = np.array([1,2])
y = np.array([[1,2],[3,4]])
```

Just like other mutable objects in Python such as list, dictionary, and set, ndarrays are mutable objects. In other words, you can modify and manipulate a ndarray in place. By indexing or slicing the array, a ndarray's elements can be accessed or changed. The methods and attributes of the ndarray could also allow you to access or change its entries.

An ndarray can be a "view" to another ndarray. That is, different ndarrays can share the same data, and changes made in one ndarray are visible in another.

In addition to data stored, ndarray also contains meta data such as its shape, size, data type etc. The following diagram illustrates a ndarray. Every item in a ndarray occupies the same

size of block in the memory. If you slice to extract an item, you would get a Python object with one of the array scalar types.



(Source: https://www.tutorialspoint.com/numpy/numpy_ndarray_object)

Here are some ndarray examples -

A single-dimensional Array:

```
import numpy as np

x = np.array([1,2,3])
print(type(x))
print(x)
```

Output:

```
<class 'numpy.ndarray'>
[1 2 3]
```

A 2-dimensional array of size 2 x 3, composed of 4-byte integer elements:

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]], np.int32)
>>> type(x)
<type 'numpy.ndarray'>
>>> x.shape
(2, 3)
>>> x.dtype
dtype('int32')
```

To sum up, a NumPy array is a grid of values, all of the same type, and is indexed by a tuple of non-negative integers. There are several important attributes associated with a NumPy array -

- **rank** is defined by the number of dimensions of the array.

- **shape** is defined by a tuple of integers giving the size of the array along each dimension.
- **size** is the number of elements in the array
- **dtype** is the data types of the elements in the array

Numpy provides a number of numeric datatypes to construct arrays. By default, NumPy tries to guess a datatype when you create an array; however, you can also explicitly specify the datatype by including an optional argument. For instance,

```
import numpy as np

x = np.array([2.0, 3.0])  # Let NumPy select datatype
print(x.dtype)            # Prints "float64"x = np.array([2, 3],
dtype=np.int64)   # Specify a datatype
print(x.dtype)            # Prints "int64"
```

There are a number of ways to create a NumPy array. For instance, we initialized NumPy arrays from Python lists, and then access elements by indices:

```
import numpy as np

a = np.array([0, 1, 2]) # Create a rank 1 array from a list
print(type(a)) # Prints "<class 'numpy.ndarray'>"
print(a.size) # Prints 3
print(a.shape) # Prints "(3,)"
print(a[0], a[1], a[2]) # Prints "0 1 2 "

a[0] = 7 # Change an element of the array
print(a) # Prints "[7, 1, 2]"

b = np.array([[1,2,3,4],[5,6,7,8]]) # Create an array from lists with rank
2
print(b.size)  #Prints 8
print(b.shape) # Prints "(2, 4)"
print(b[0, 0], b[0, 1], b[1, 0]) # Prints "1 2 5"
```

Now, let us take a look at matrix. Matrix objects are a subclass of ndarray, so they inherit all the attributes and methods of ndarrays. Matrices are two-dimentional while ndarrays can be N-dimensional. For example, we can create a matrix and access elements by indices -

```
import numpy as np
# Create a new matrix
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])print(a)

# prints "array([[ 1,  2,  3],
#                [ 4,  5,  6],
#                [ 7,  8,  9],
```

```
#                [10, 11, 12]])"
# Create an array of indices
b = np.array([0, 2, 0, 1])#Obtain one entry from each row w/ indices in b
print(a[np.arange(4), b])

#Prints "[ 1  6  7 11]"
```

Slicing vs Fancy Indexing

The following use case shows how to slice a NumPy array. First, we use np.arange function to create a ndarray object containing a sequence. Then, we use a[2:] to slice items starting from the index. If you recall that a ndarray has zero-based index, you will see the slicing operation returns a view of the original array starting from the the third entry -

```
# slice items starting from index
import numpy as np
a = np.arange(10)
print(a)        #Prints [0 1 2 3 4 5 6 7 8 9]
print a[2:]     #Prints [2 3 4 5 6 7 8 9]
```

On the other hand, fancy indexing will return a newly created ndarray. Fancy indexing allows us to pass arrays of indices and quickly access / modify subsets of an array's values.

```
import numpy as np
x = np.arange(10)
i = np.array([1, 3, 5])
x[i] = -99
print(x)
```

output:

```
[0 -99 2 -99 4 -99 6 7 8 9]
```

Vectorization

There is an important concept called **vectorization**. It is highly recommended to use Numpy and Scipy methods instead of looping code. The reason is that vectorization in NumPy is much faster than the pure Python implementation.

When implementing a new algorithm, it is recommended to start implementing by using Numpy and Scipy. It is preferred using the vectorized idioms of those libraries instead of looping code. It is desirable to replace any nested "for loops" by calling equivalent Numpy

array methods and tools like map(), list comprehensions, etc. In this way, we avoid the CPU wasting time in the Python interpreter. See benchmarking performance below (comparing numpy.sum vs. summing over a loop) -

```
from timeit import Timer
import numpy as np

def timer(*funcs):
    # find the maximum function name length
    if len(funcs) > 1:
        maxlen = max(*[len(func) for func in funcs])
    elif len(funcs) == 1:
        maxlen = len(funcs[0])
    else:
        return

    # run each function 1000 times
    times = []
    for func in funcs:
        timerfunc = Timer("%s()" % func, "from __main__ import %s" % func)
        runtime = timerfunc.repeat(repeat=1000, number=1)
        mtime = np.mean(runtime)
        stime = np.std(runtime)
        dfunc = func + (" " * (maxlen - len(func) + 1))
        print "%s: %.7f +/- %.7f seconds" % (dfunc, mtime, stime)

#vectorization
def numpy_sum():
    total = np.sum(np.arange(1000))

#for loop
def loop_sum():
    total = 0
    for i in xrange(1000):
        total += i

>>> timer("vectorization numpy_sum", "loop_sum")
vectorization numpy_sum : 0.0000096 +/- 0.0000431 seconds
loop_sum : 0.0000417 +/- 0.0000449 seconds
```

As you can see from above, vectorization via NumPy sum function is much faster than summing over a loop as an algorithm implemented in "pure Python." On the other hand, vectorization may increase memory complexity of some operations from constant to linear, because temporary arrays must be created that are as large as the inputs.

NumPy Limitations

NumPy has a lot of strength, but it also has a few limitations -

- Unlike in Python lists, inserting or appending entries to an array is costly. For example, np.pad(...) for extending arrays creates new ones of the desired shape and padding values in stead of "padding" on the existing one. That is, this operation copies the given array into the new array and returns it.
- Similarly, np.concatenate([a1,a2]) operation does not join the two arrays but returns a new one instead,. The new array is populated with the entries from both.
- Reshaping the dimensionality of an array with np.reshape(...) is only possible when the number of entries remains the same.

Python developer community has been working on solutions to mitigate impact of those limitations.

SciPy - Extending Functionality of NumPy

The SciPy library is built upon NumPy and serves as one of the core packages of the NumPy/SciPy scientific computing stack. It provides efficient numerical routines such as those for numerical integration, interpolation, optimization, linear algebra, signal and image processing.

Like NumPy, the SciPy library development is supported by an open community of developers.

The SciPy package contains a list of sub-packages:

- constants: physical constants and conversion factors
- cluster: hierarchical clustering, vector quantization, K-means
- fftpack: Discrete Fourier Transform algorithms
- integrate: numerical integration routines
- interpolate: interpolation tools
- io: data input and output
- lib: Python wrappers to external libraries
- linalg: linear algebra routines
- misc: miscellaneous utilities (e.g. image reading/writing)
- ndimage: various functions for multi-dimensional image processing

- optimize: optimization algorithms including linear programming
- signal: signal processing tools
- sparse: sparse matrix and related algorithms
- spatial: KD-trees, nearest neighbors, distance functions
- special: special functions
- stats: statistical functions
- weave: tool for writing C/C++ code as Python multiline strings

Here is how we can import the SciPy subpackages separately:

```
>>> from scipy import linalg, optimize
```

Here is an example shows how to perform Singular Value Decomposition via scipy.linalg.svd, which factorizes the original matrix into two unitary matrices 'U' and 'Vh' and a 1-D array 's' of singular values:

```
#importing the scipy and numpy packages
 from scipy import linalg

import numpy as np

#Creating numpy array
a = np.random.randn(3, 2)

#Performing SVD
 U, s, Vh = linalg.svd(a)
```

For example, the original matrix 'a' could be -

```
[[-0.78505966  0.02923608]
 [ 0.42223307  0.71216619]
 [-0.16669502  0.87137861]]
```

The decomposed matrices would be -

```
#U
[[ 0.1577529   0.86255509 -0.48074186]
 [-0.69946385 -0.24604273 -0.67097935]
 [-0.6970397   0.44211049  0.56451215]]

#Vh
[[-0.26535147 -0.96415175]
 [-0.96415175  0.26535147]]

#s
[ 1.14184091  0.88652248]
```

You might have noticed that, by design, NumPy package was kept separate from SciPy. The rationale behind is to avoid installing SciPy just for the purpose of obtaining an array object.

Pandas for Data Analysis

For a data scientist/analyst, life is typically comprised of tasks such as cleaning data, merging/joining, analyzing, plotting or tabular displaying, and modeling. Often when we work with data, we would like to work intuitively on data in a "spreadsheet." The fast, flexible, and expressive Pandas data structures are designed to simplify working with structured (tabular and multi-dimensional) and time-series data. If you are already familiar to R and Excel concepts, you would find Pandas easy to pick up.

The Pandas library is built on top of NumPy. It is one of the most preferred tools for data analysts to do data manipulation and analysis.

Pandas is suitable for handling the following data -

- Tabular data with heterogeneously-typed columns
- Ordered and unordered time series data
- Arbitrary matrix data with row and column labels
- Any other form of observational/statistical data sets

The two primary data structures of Pandas, Series (1-dimensional) and DataFrame (2-dimensional), handle the vast majority of typical use cases in finance, statistics, and engineering.

Pandas makes dealing with data sets easy by -

- Easy handling of missing data
- Easy filtering of data by rows
- Columns can be inserted and deleted from DataFrame
- Automatic and explicit data alignment: objects can be explicitly aligned to a set of labels, or the user can simply ignore the labels and let Series, DataFrame, etc. automatically align the data for you in computations
- Powerful, flexible group by functionality
- Slicing, fancy indexing, and subsetting of large data sets

- Intuitive merging and joining data sets
- Flexible reshaping and pivoting of data sets
- Loading data from flat files (CSV and delimited), Excel files, databases
- Time series-specific functionality: date range generation and frequency conversion, moving window statistics, and lagging, etc.

Series

Pandas Series is a one-dimensional labeled array holding data of any type (integer, string, float, python objects, etc.). Pandas Series can be regarded as a column in a spreadsheet.

Pands allows you to set a series as an index for a data frame. An index is defined as the axis labels. Labels do not have to be unique but must be hashable.

DataFrame

The most powerful feature of Pandas is DataFrame. It is a 2D tabular data structure with row and column indexes.

To illustrate data frame capabilities, we use Pandas to read the daily stock quotes of ServiceNow into a dataFrame over a period of 2018/08 ~ 2019/08:

```
import pandas as pd
import numpy as np

df = pd.read_csv('NOW.csv')
df.head()
```

The output would display top rows of ServiceNow daily quote data -

```
         Date        Open        High         Low       Close   Adj_Close
\
0  2018-08-15  184.169998  187.440002  180.610001  182.729996  182.729996
1  2018-08-16  184.350006  184.520004  181.759995  182.029999  182.029999
2  2018-08-17  182.000000  183.160004  178.800003  180.830002  180.830002
3  2018-08-20  180.169998  181.919998  179.149994  181.360001  181.360001
4  2018-08-21  182.399994  184.830002  181.820007  182.100006  182.100006

    Volume
0  1261100
1   745700
2  1656000
3  1661500
```

```
4  1555100
```

If you would like to force specific data types for each column, you can use dtype option -

```
df = pd.read_csv('~/NOW.csv', dtype = {'Date': str,'Open': np.float64,\
    'High': np.float64, 'Low': np.float64,\
    'Close': np.float64, 'Adj_Close': np.float64})
```

By using df.info() function, You can obtain metadata info about this data frame -

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 252 entries, 0 to 251
Data columns (total 7 columns):
Date         252 non-null object
Open         252 non-null float64
High         252 non-null float64
Low          252 non-null float64
Close        252 non-null float64
Adj_Close    252 non-null float64
Volume       252 non-null int64
dtypes: float64(5), int64(1), object(1)
memory usage: 13.9+ KB
```

To select certain columns into a new data frame, we use the following -

```
df_selected = df[['Date','Volume', 'Close']]
```

Pandas also make it easy to compute summary statistics on data frames. For instance, to compute the daily average of trading volume for each month, we could do the following two steps -

First, we derive a new column called 'Month' based on the 'Date' column -

```
df['Month'] = df['Date'].apply(lambda x: x[:7])
```

Then we compute monthly mean values for numerical variable 'Volume' -

```
df2 = df.groupby(['Month'])['Volume'].mean()
```

As a result, the grouped means are -

```
Month
2018-08 1.702869e+06
2018-09 2.011468e+06
2018-10 2.746283e+06
2018-11 2.024719e+06
2018-12 2.194032e+06
2019-01 2.418271e+06
2019-02 1.920826e+06
```

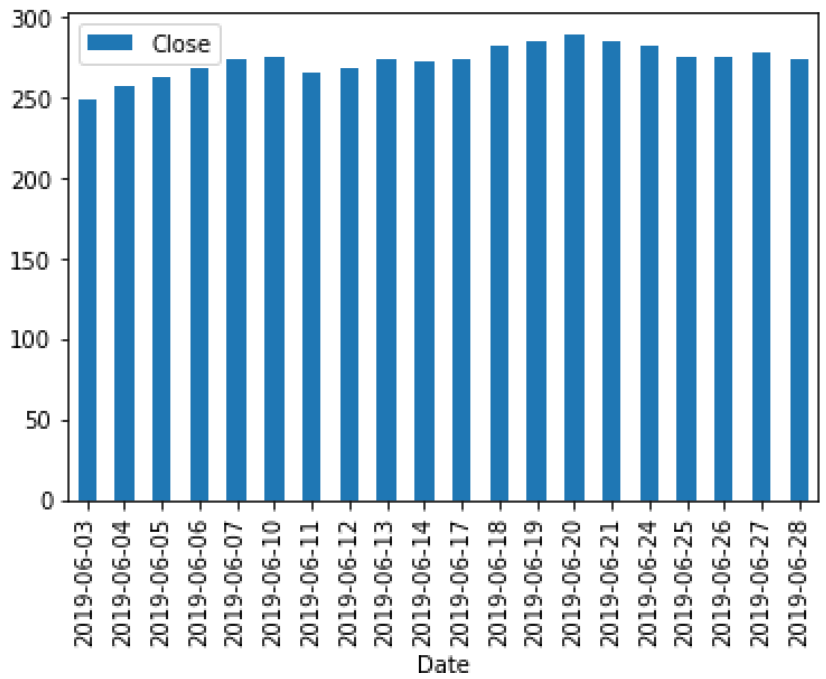
```
2019-03 1.837414e+06
2019-04 1.855890e+06
2019-05 1.726759e+06
2019-06 1.615410e+06
2019-07 1.661509e+06
2019-08 1.840095e+06
```

Moreover, Pandas makes it really easy to filter on rows based on values. To filter on rows by dates from the second quarter of the calendar year 2019, you can run the following -

```
df_q2 = df.loc[(df['Month'] >= '2019-04') & (df['Month'] <= '2019-06')]
```

Besides, you can directly plot data from our data frame. For example, we would like plot daily close price of ServiceNow over the entire month of June 2019 -

```
df_Jun2019.plot(kind='bar', x='Date', y='Close')
```



Regarding time series, Pandas has a powerful toolbox for working with dates, times, and time-indexed data. We will discuss the topic of times series in detail in a later chapter.

As another powerful feature of Pandas, we can easily merge data frames by performing SQL-like operations.

Just like NumPy/SciPy, Pandas package has great community support and rich online resources.

Summary

To sum up, NumPy enables us to carry out numerical operations with ndarrays to achieve optimized performance. If the operations on an array do not change the size of array, full compiled code speed can be achieved. Besides, Pandas gives us great power and flexibility to deal with data sets and make them useful for our analysis.

The three pillars of scientific Python are -

- NumPy for scientific computing
- Pandas for data manipulation
- Matplotlib for data visualization.

Equipped with a combination of NumPy, SciPy, Matplotlib, as well as IPython Notebook (now known as the Jupyter Notebook), you will have a powerful set of tools in hand for building algorithms for trading.

Index