NumPy Cookbook Second Edition

NumPy has the ability to give you speed and high productivity. High performance calculations can be done easily with clean and efficient code, and it allows you to execute complex algebraic and mathematical computations in no time.

This book will give you a solid foundation in NumPy arrays and universal functions. Starting with the installation and configuration of IPython, you'll learn about advanced indexing and array concepts along with commonly used yet effective functions. You will then cover practical concepts such as image processing, special arrays, and universal functions. You will also learn about plotting with Matplotlib and the related SciPy project with the help of examples. At the end of the book, you will study how to explore atmospheric pressure and its related techniques. By the time you finish this book, you'll be able to write clean and fast code with NumPy.

What this book will do for you...

- Learn advanced indexing and linear algebra
- Deal with missing stock price data using masked arrays
- Explore everything you need to know about image processing
- Dive into broadcasting and histograms
- Profile NumPy code and visualize the results
- Speed up your code with Cython
- Use universal functions and interoperability features
- Analyze your performance using Quality Assurance
- Learn about exploratory and predictive data analysis with NumPy

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions for solving the problem efficiently
- Clear explanations of what you did
- Apply the solution to other situations

Quick answers to common problems

NumPy has the ability to give you speed and high productivity. High performance calculations can be done easily with clean and efficient code, and it allows you to execute complex algebraic and mathematical computations in no time.

This book will give you a solid foundation in NumPy arrays and universal functions. Starting with the installation and configuration of IPython, you'll learn about advanced indexing and array concepts along with commonly used yet effective functions. You will then cover practical concepts such as image processing, special arrays, and universal functions. You will also learn about plotting with Matplotlib and the related SciPy project with the help of examples. At the end of the book, you will study how to explore atmospheric pressure and its related techniques. By the time you finish this book, you'll be able to write clean and fast code with NumPy.

What this book will do for you...

- Learn advanced indexing and linear algebra
- Deal with missing stock price data using masked arrays
- Explore everything you need to know about image processing
- Dive into broadcasting and histograms
- Profile NumPy code and visualize the results
- Speed up your code with Cython
- Use universal functions and interoperability features
- Analyze your performance using Quality Assurance
- Learn about exploratory and predictive data analysis with NumPy

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions for solving the problem efficiently
- Clear explanations of what you did
- Apply the solution to other situations

Quick answers to common problems

NumPy has the ability to give you speed and high productivity. High performance calculations can be done easily with clean and efficient code, and it allows you to execute complex algebraic and mathematical computations in no time.

This book will give you a solid foundation in NumPy arrays and universal functions. Starting with the installation and configuration of IPython, you'll learn about advanced indexing and array concepts along with commonly used yet effective functions. You will then cover practical concepts such as image processing, special arrays, and universal functions. You will also learn about plotting with Matplotlib and the related SciPy project with the help of examples. At the end of the book, you will study how to explore atmospheric pressure and its related techniques. By the time you finish this book, you'll be able to write clean and fast code with NumPy.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Getting to Grips with Commonly Used Functions'
- A synopsis of the book’s content
- More information on NumPy Cookbook Second Edition

About the Author

Ivan Idris has an MSc in experimental physics. His graduation thesis had a strong emphasis on applied computer science. After graduating, he worked for several companies as a Java developer, data warehouse developer, and QA analyst. His main professional interests are business intelligence, big data, and cloud computing. Ivan enjoys writing clean, testable code and interesting technical articles. He is the author of NumPy Beginner's Guide, NumPy Cookbook, Python Data Analysis, and Learning NumPy, all by Packt Publishing. You can find more information about him and a few NumPy examples at http://ivanidris.net/wordpress/.
This second edition adds two new chapters on the new NumPy functionality and data analysis. We NumPy users live in exciting times. New NumPy-related developments seem to come to our attention every week, or maybe even daily. At the time of the first edition, the NumFocus, short for NumPy Foundation for Open Code for Usable Science, was created. The Numba project—a NumPy-aware dynamic Python compiler using LLVM—was also announced. Further, Google added support to their cloud product called Google App Engine.

In the future, we can expect improved concurrency support for clusters of GPUs and CPUs. OLAP-like queries will be possible with NumPy arrays. This is wonderful news, but we have to keep reminding ourselves that NumPy is not alone in the scientific (Python) software ecosystem. There is SciPy, matplotlib (a very useful Python plotting library), IPython (an interactive shell), and Scikits. Outside the Python ecosystem, languages such as R, C, and Fortran are pretty popular. We will cover the details of exchanging data with these environments.

What This Book Covers

Chapter 1, Winding Along with IPython, introduces IPython, a toolkit mostly known for its shell. The web-based notebook is an exciting feature covered in detail here. Think of MATLAB and Mathematica, but in your browser, it's open source and free.

Chapter 2, Advanced Indexing and Array Concepts, shows that NumPy has very efficient arrays that are easy to use due to the powerful indexing mechanism. This chapter describes some of the more advanced and tricky indexing techniques.

Chapter 3, Getting to Grips with Commonly Used Functions, makes an attempt to document the most essential functions that every NumPy user should know. NumPy has many functions—too many to even mention in this book!

Chapter 4, Connecting NumPy with the Rest of the World, the number of programming languages, libraries, and tools one encounters in the real world is mind-boggling. Some of the software runs on the cloud, while some of it lives on your local machine or a remote server. Being able to fit and connect NumPy with such an environment is just as important as being able to write standalone NumPy code.

Chapter 5, Audio and Image Processing, assumes that when you think of NumPy, you probably don't think of sounds or images. This will change after reading this chapter.
Chapter 6, *Special Arrays and Universal Functions*, introduces pretty technical topics. This chapter explains how to perform string operations, ignore illegal values, and store heterogeneous data.

Chapter 7, *Profiling and Debugging*, shows the skills necessary to produce good software. We demonstrate several convenient profiling and debugging tools.

Chapter 8, *Quality Assurance*, deserves a lot of attention because it’s about quality. We discuss common methods and techniques, such as unit testing, mocking, and BDD, using the NumPy testing utilities.

Chapter 9, *Speeding Up Code with Cython*, introduces Cython, which tries to combine the speed of C and the strengths of Python. We show you how Cython works from the NumPy perspective.

Chapter 10, *Fun with Scikits*, covers Scikits, which are a yet another part of the fascinating scientific Python ecosystem. A quick tour guides you through some of the most useful Scikits projects.


Chapter 12, *Exploratory and Predictive Data Analysis with NumPy*, presents real-world analysis of meteorological data. I've added this chapter in the second edition.
In this chapter, we will cover a number of commonly used functions:

- \( \text{sqrt}() \), \( \log() \), \( \text{arange}() \), \( \text{astype}() \), and \( \text{sum}() \)
- \( \text{ceil}() \), \( \text{modf}() \), \( \text{where}() \), \( \text{ravel}() \), and \( \text{take}() \)
- \( \text{sort}() \) and \( \text{outer}() \)
- \( \text{diff}() \), \( \text{sign}() \), and \( \text{eig}() \)
- \( \text{histogram}() \) and \( \text{polyfit}() \)
- \( \text{compress}() \) and \( \text{randint}() \)

We will be discussing these functions in the following recipes:

- Summing Fibonacci numbers
- Finding prime factors
- Finding palindromic numbers
- The steady state vector
- Discovering a power law
- Trading periodically on dips
- Simulating trading at random
- Sieving integers with the Sieve of Eratosthenes
Introduction

This chapter is about the commonly used NumPy functions. These are the functions that you will be using on a daily basis. Obviously, the usage may differ for you. There are so many NumPy functions that it is virtually impossible to know all of them, but the functions in this chapter are the bare minimum with which we should be familiar.

Summing Fibonacci numbers

In this recipe, we will sum the even-valued terms in the Fibonacci sequence whose values do not exceed 4 million. The Fibonacci series is a sequence of integers starting with zero, where each number is the sum of the previous two, except (of course) the first two numbers, zero and one (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 ...).

The sequence was published by Fibonacci in 1202 and originally did not include zero. Actually, it was already known to Indian mathematicians in earlier centuries. Fibonacci numbers have applications in mathematics, computer science, and biology.

For more information, read the Wikipedia article about Fibonacci numbers at http://en.wikipedia.org/wiki/Fibonacci_number.

This recipe uses a formula based on the golden ratio, which is an irrational number with special properties comparable to pi. The golden ratio is given by the following formula:

\[ \varphi = \frac{1 + \sqrt{5}}{2} \]

We will use the \texttt{sqrt()}, \texttt{log()}, \texttt{arange()}, \texttt{astype()}, and \texttt{sum()} functions. The Fibonacci sequence's recurrence relation has the following solution, which involves the golden ratio:

\[ F_n = \frac{\varphi^n - (-\varphi)^n}{\sqrt{5}} \]

How to do it...

The following is the complete code for this recipe from the \texttt{sum_fibonacci.py} file in this book's code bundle:

```
import numpy as np
```
Each new term in the Fibonacci sequence is generated by adding the previous two terms.

By starting with 1 and 2, the first 10 terms will be:

1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

1. Calculate phi

\[
\phi = \frac{1 + \sqrt{5}}{2}
\]

print("Phi", phi)

2. Find the index below 4 million

\[
n = \frac{\log(4 \cdot 10^6 \cdot \sqrt{5} + 0.5)}{\log(\phi)}
\]

print(n)

3. Create an array of 1-n

\[
n = \text{np.arange}(1, n)
\]

print(n)

4. Compute Fibonacci numbers

\[
fib = \frac{\phi^n - (-1/\phi)^n}{\sqrt{5}}
\]

print("First 9 Fibonacci Numbers", fib[:9])

5. Convert to integers

# optional

fib = fib.astype(int)

print("Integers", fib)

6. Select even-valued terms

eventerms = fib[fib % 2 == 0]

print(eventerms)

7. Sum the selected terms

print(eventerms.sum())

The first thing to do is calculate the golden ratio (see \url{http://en.wikipedia.org/wiki/Golden_ratio}), also called the golden section or golden mean.

1. Use the sqrt() function to calculate the square root of 5:

\[
\phi = \frac{1 + \sqrt{5}}{2}
\]

print("Phi", phi)
This prints the golden mean:

\[ \Phi \approx 1.61803398875 \]

2. Next, in the recipe, we need to find the index of the Fibonacci number below 4 million. A formula for this is given in the Wikipedia page, and we will compute it using that formula. All we need to do is convert log bases with the \( \log() \) function. We don't need to round the result down to the closest integer. This is automatically done for us in the next step of the recipe:

\[
n = \frac{\log(4 \times 10^6 \times \sqrt{5} + 0.5)}{\log(\Phi)}
\]

print(n)

The value of \( n \) is as follows:

33.2629480359

3. The \( \text{arange}() \) function is a very basic function that many people know. Still, we will mention it here for completeness:

\[
n = \text{np.arange}(1, n)
\]

4. There is a convenient formula we can use to calculate the Fibonacci numbers. We will need the golden ratio and the array from the previous step in this recipe as input parameters. Print the first nine Fibonacci numbers to check the result:

\[
fib = \frac{\Phi^n - (-\frac{1}{\Phi})^n}{\sqrt{5}}
\]

print("First 9 Fibonacci Numbers", fib[:9])

I could have made a unit test instead of a print statement. A unit test is a test that tests a small unit of code, such as a function. This variation of the recipe is left as an exercise for you.

Take a look at Chapter 8, Quality Assurance, for pointers on how to write a unit test.

We are not starting with the number 0 here, by the way. The aforementioned code gives us a series as expected:

```
First 9 Fibonacci Numbers [ 1. 1. 2. 3. 5. 8. 13. 21. 34.]
```

You can plug this right into a unit test, if you want.

5. Convert to integers.

This step is optional. I think it's nice to have an integer result at the end. Okay, I actually wanted to show you the \( \text{astype()} \) function:
fib = fib.astype(int)
print("Integers", fib)

This code gives us the following result, after snipping a bit for brevity:

```
Integers [      1       1       2       3       5       8      13
21       34
... snip ... snip ...
317811  514229  832040 1346269 2178309 3524578]
```

6. Select the even-valued terms.

This recipe demands that we select the even-valued terms now. This should be easy for you if you followed the `Indexing with Booleans` recipe in Chapter 2, Advanced Indexing and Array Concepts:

```
eventerms = fib[fib % 2 == 0]
print(eventerms)
```

There we go:

```
[      2       8      34     144     610    2584   10946   46368
196418  832040 3524578]
```

**How it works...**

In this recipe, we used the `sqrt()`, `log()`, `arange()`, `astype()`, and `sum()` functions. Their description is as follows:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sqrt()</code></td>
<td>This function calculates the square root of array elements (see <a href="http://docs.scipy.org/doc/numpy/reference/generated/numpy.sqrt.html">http://docs.scipy.org/doc/numpy/reference/generated/numpy.sqrt.html</a>)</td>
</tr>
<tr>
<td><code>log()</code></td>
<td>This function calculates the natural logarithm of array elements (see <a href="http://docs.scipy.org/doc/numpy/reference/generated/numpy.log.html#numpy.log">http://docs.scipy.org/doc/numpy/reference/generated/numpy.log.html#numpy.log</a>)</td>
</tr>
<tr>
<td><code>arange()</code></td>
<td>This function creates an array with the specified range (see <a href="http://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html">http://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html</a>)</td>
</tr>
<tr>
<td><code>astype()</code></td>
<td>This function converts array elements to a specified data type (see <a href="http://docs.scipy.org/doc/numpy/reference/generated/numpy.chararray.astype.html">http://docs.scipy.org/doc/numpy/reference/generated/numpy.chararray.astype.html</a>)</td>
</tr>
<tr>
<td><code>sum()</code></td>
<td>This function calculates the sum of array elements (see <a href="http://docs.scipy.org/doc/numpy/reference/generated/numpy.sum.html">http://docs.scipy.org/doc/numpy/reference/generated/numpy.sum.html</a>)</td>
</tr>
</tbody>
</table>
Getting to Grips with Commonly Used Functions

See also

- The Indexing with Booleans recipe in Chapter 2, Advanced Indexing and Array Concepts

## Finding prime factors

Prime factors ([http://en.wikipedia.org/wiki/Prime_factor](http://en.wikipedia.org/wiki/Prime_factor)) are prime numbers that exactly divide an integer without leaving a remainder. Finding prime factors seems almost impossible for big numbers. Therefore, prime factors have an application in cryptography. However, using the right algorithm—Fermat's factorization method ([http://en.wikipedia.org/wiki/Fermat%27s_factorization_method](http://en.wikipedia.org/wiki/Fermat%27s_factorization_method)) and NumPy—factoring becomes relatively easy for small numbers. The idea is to factor a number \( N \) into two numbers, \( c \) and \( d \), according to the following equation:

\[
N = cd = (a + b)(a - b) = a^2 - b^2
\]

We can apply the factorization recursively until we get the required prime factors.

### How to do it...

The following is the entire code needed to solve the problem of finding the largest prime factor of 600851475143 (see the `fermatfactor.py` file in this book's code bundle):

```python
from __future__ import print_function
import numpy as np

#The prime factors of 13195 are 5, 7, 13 and 29.
#What is the largest prime factor of the number 600851475143 ?

N = 600851475143
LIM = 10 ** 6

def factor(n):
    #1. Create array of trial values
    a = np.ceil(np.sqrt(n))
    lim = min(n, LIM)
    a = np.arange(a, a + lim)
    b2 = a ** 2 - n

    #2. Check whether b is a square
```

-48-
fractions = np.modf(np.sqrt(b2))[0]

#3. Find 0 fractions
indices = np.where(fractions == 0)

#4. Find the first occurrence of a 0 fraction
a = np.ravel(np.take(a, indices))[0]
    # Or a = a[indices][0]

a = int(a)
b = int(np.sqrt(a ** 2 - n))
c = a + b
d = a - b

if c == 1 or d == 1:
    return

print(c, d)
factor(c)
factor(d)
factor(N)

The algorithm requires us to try a number of trial values for a:

1. Create an array of trial values.
   It makes sense to create a NumPy array and eliminate the need for loops.
   However, you should be careful not to create an array that is too big in terms of
   memory requirements. On my system, an array of a million elements seems to
   be just the right size:
   a = np.ceil(np.sqrt(n))
lim = min(n, LIM)
a = np.arange(a, a + lim)
b2 = a ** 2 - n

   We used the ceil() function to return the ceiling of the input, element-wise.

2. Get the fractional part of the b array.
   We are now supposed to check whether b is a square. Use the NumPy modf() function to get the fractional part of the b array:
   fractions = np.modf(np.sqrt(b2))[0]
Getting to Grips with Commonly Used Functions

3. Find 0 fractions.
   Call the `where()` NumPy function to find the indexes of zero fractions, where the
   fractional part is 0:
   
   \[
   \text{indices} = \text{np.where}(\text{fractions} == 0)
   \]

4. Find the first occurrence of a zero fraction.
   First, call the `take()` NumPy function with the `indices` array from the previous
   step to get the values of zero fractions. Now flatten this array with the `ravel()`
   NumPy function:
   
   \[
   a = \text{np.ravel}(\text{np.take}(a, \text{indices}))[0]
   \]

   This line is a bit convoluted, but it does demonstrate two useful
   functions. It would have been simpler to write \( a = a[\text{indices}][0] \).

   The output for this code is the following:
   
   \[1234169 \ 486847 \ 1471 \ 839 \ 6857 \ 71\]

How it works...

We applied the Fermat factorization recursively using the `ceil()`, `modf()`, `where()`,
`ravel()`, and `take()` NumPy functions. The description of these functions is as follows:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ceil()</code></td>
<td>Calculates the ceiling of array elements (see <a href="http://docs.scipy.org/doc/numpy/reference/generated/numpy.ceil.html">http://docs.scipy.org/doc/numpy/reference/generated/numpy.ceil.html</a>)</td>
</tr>
<tr>
<td><code>modf()</code></td>
<td>Returns the fractional and integral part of floating-point numbers (see <a href="http://docs.scipy.org/doc/numpy/reference/generated/numpy.modf.html">http://docs.scipy.org/doc/numpy/reference/generated/numpy.modf.html</a>)</td>
</tr>
<tr>
<td><code>where()</code></td>
<td>Returns array indices based on condition (see <a href="http://docs.scipy.org/doc/numpy/reference/generated/numpy.where.html">http://docs.scipy.org/doc/numpy/reference/generated/numpy.where.html</a>)</td>
</tr>
<tr>
<td><code>ravel()</code></td>
<td>Returns a flattened array (see <a href="http://docs.scipy.org/doc/numpy/reference/generated/numpy.ravel.html">http://docs.scipy.org/doc/numpy/reference/generated/numpy.ravel.html</a>)</td>
</tr>
<tr>
<td><code>take()</code></td>
<td>Takes an element from an array (see <a href="http://docs.scipy.org/doc/numpy/reference/generated/numpy.take.html">http://docs.scipy.org/doc/numpy/reference/generated/numpy.take.html</a>)</td>
</tr>
</tbody>
</table>
Finding palindromic numbers

A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is 9009 = 91 x 99. Let's try to find the largest palindrome made from the product of two 3-digit numbers.

How to do it...

The following is the complete program from the `palindromic.py` file in this book's code bundle:

```python
import numpy as np

# A palindromic number reads the same both ways.
# The largest palindrome made from the product of two 2-digit numbers
# is 9009 = 91 x 99.

# Find the largest palindrome made from the product of two 3-digit
# numbers.

# 1. Create 3-digits numbers array
a = np.arange(100, 1000)
np.testing.assert_equal(100, a[0])
np.testing.assert_equal(999, a[-1])

# 2. Create products array
numbers = np.outer(a, a)
numbers = np.ravel(numbers)
numbers.sort()
np.testing.assert_equal(810000, len(numbers))
np.testing.assert_equal(10000, numbers[0])
np.testing.assert_equal(998001, numbers[-1])

# 3. Find largest palindromic number
for number in numbers[::-1]:
    s = str(numbers[i])
    if s == s[::-1]:
        print(s)
        break
```
We will create an array to hold three-digit numbers from 100 to 999 using our favorite NumPy function, `arange()`.

1. Create an array of three-digit numbers.
   Check the first and the last element of the array with the `assert_equal()` function from the `numpy.testing` package:

   ```python
   a = np.arange(100, 1000)
   np.testing.assert_equal(100, a[0])
   np.testing.assert_equal(999, a[-1])
   ```

2. Create the products array.
   Now we will create an array to hold all the possible products of the elements of the three-digit array with itself. We can accomplish this with the `outer()` function. The resulting array needs to be flattened with `ravel()` to be able to easily iterate over it. Call the `sort()` method on the array to make sure that the array is properly sorted. After that, we can do some sanity checks:

   ```python
   numbers = np.outer(a, a)
   numbers = np.ravel(numbers)
   numbers.sort()
   np.testing.assert_equal(810000, len(numbers))
   np.testing.assert_equal(10000, numbers[0])
   np.testing.assert_equal(998001, numbers[-1])
   ```

The code prints 906609, which is a palindromic number.

**How it works...**

We saw the `outer()` function in action. This function returns the outer product of two arrays (http://en.wikipedia.org/wiki/Outer_product). The outer product of two vectors (one-dimensional lists of numbers) creates a matrix. This is the opposite of an inner product, which returns a scalar number for two vectors. The outer product is used in physics, signal processing, and statistics. The `sort()` function returns a sorted copy of an array.

**There's more...**

It might be a good idea to check the result. Find out which two 3-digit numbers produce our palindromic number by modifying the code a bit. Try implementing the last step in the NumPy way.
The steady state vector

A Markov chain is a system that has at least two states. For detailed information on Markov chains, please refer to http://en.wikipedia.org/wiki/Markov_chain. The state at time \( t \) depends on the state at time \( t-1 \), and only the state at \( t-1 \). The system switches at random between these states. The chain doesn't have any memory about the states. Markov chains are often used to model phenomena in physics, chemistry, finance, and computer science. For instance, Google's PageRank algorithm uses Markov chains to rank web pages.

I would like to define a Markov chain for a stock. Let's say that we have the states flat, up, and down. We can determine the steady state based on the end-of-the-day close prices.

Far into the distant future or theoretically after an infinite amount of time, the state of our Markov chain system will not change anymore. This is called a steady state (see http://en.wikipedia.org/wiki/Steady_state). A dynamic equilibrium is a type of steady state. For a stock, achieving a steady state may mean that the related company has become stable. The stochastic matrix (see http://en.wikipedia.org/wiki/Stochastic_matrix) \( A \) contains the state transition probabilities, and when applied to the steady state, it yields the same state \( x \). The mathematical notation for this is as follows:

\[
Ax = x
\]

Another way to look at this is as the eigenvector (see http://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors) for eigenvalue 1. Eigenvalues and eigenvectors are fundamental concepts of linear algebra with applications in quantum mechanics, machine learning, and other sciences.

How to do it...

The following is the complete code for the steady state vector example from the steady_state_vector.py file in this book's code bundle:

```python
from __future__ import print_function
from matplotlib.finance import quotes_historical_yahoo
from datetime import date
import numpy as np

today = date.today()
```
Getting to Grips with Commonly Used Functions

```python
start = (today.year - 1, today.month, today.day)

quotes = quotes_historical_yahoo('AAPL', start, today)
close = [q[4] for q in quotes]

states = np.sign(np.diff(close))

NDIM = 3
SM = np.zeros((NDIM, NDIM))

signs = [-1, 0, 1]
k = 1

for i, signi in enumerate(signs):
    start_indices = np.where(states[:-1] == signi)[0]
    N = len(start_indices) + k * NDIM

    if N == 0:
        continue

    end_values = states[start_indices + 1]

    for j, signj in enumerate(signs):
        occurrences = len(end_values[end_values == signj])
        SM[i][j] = (occurrences + k) / float(N)

print(SM)
eig_out = np.linalg.eig(SM)
print(eig_out)

idx_vec = np.where(np.abs(eig_out[0] - 1) < 0.1)
print("Index eigenvalue 1", idx_vec)
x = eig_out[1][:, idx_vec].flatten()
print("Steady state vector", x)
print("Check", np.dot(SM, x))
```
Now we need to obtain the data:

1. Obtain 1 year of data.
   One way we can do this is with matplotlib (refer to the *Installing matplotlib* recipe in *Chapter 1, Winding Along with IPython*, if necessary). We will retrieve the data of the last year. Here is the code to do this:

   ```python
today = date.today()
start = (today.year - 1, today.month, today.day)
quotes = quotes_historical_yahoo('AAPL', start, today)
```

2. Select the close price.
   We now have historical data from Yahoo! Finance. The data is represented as a list of tuples, but we are only interested in the close price.
   The first element in the tuple represents the date. It is followed by the open, high, low, and close prices. The last element is the volume. We can select the close prices as follows:

   ```python
   close = [q[4] for q in quotes]
   ```

   The close price is the fifth number in each tuple. We should have a list of about 253 close prices now.

3. Determine the states.
   We can determine the states by subtracting the price of sequential days with the `diff()` NumPy function. The state is then given by the sign of the difference. The `sign()` NumPy function returns -1 for a negative number, 1 for a positive number, and 0 otherwise:

   ```python
   states = np.sign(np.diff(close))
   ```

4. Initialize the stochastic matrix to 0 values.
   We have three possible start states and three possible end states for each transition. For instance, if we start from an up state, we could switch to:
   - Up
   - Flat
   - Down

   Initialize the stochastic matrix with the `zeros()` NumPy function:

   ```python
   NDIM = 3
   SM = np.zeros((NDIM, NDIM))
   ```
5. For each sign, select the corresponding start state indices.

Now the code becomes a bit messy. We will have to use actual loops! We will loop over the possible signs and select the start state indices corresponding to each sign. Select the indices with the `where()` NumPy function. Here, k is a smoothing constant, which we will discuss later on:

```python
signs = [-1, 0, 1]
k = 1

for i, signi in enumerate(signs):
    # we start the transition from the state with the specified sign
    start_indices = np.where(states[:-1] == signi)[0]
```


We can now count the number of occurrences of each transition. Dividing it by the total number of transitions for a given start state gives us the transition probabilities for our stochastic matrix. This is not the best method, by the way, since it could be overfitting.

In real life, we could have a day when the close price does not change, although this is unlikely for liquid stock markets. One way to deal with zero occurrences is to apply additive smoothing (http://en.wikipedia.org/wiki/Additive_smoothing). The idea is to add a certain constant to the number of occurrences we find, getting rid of zeroes. The following code calculates the values of the stochastic matrix:

```python
N = len(start_indices) + k * NDIM

# skip since there are no transitions possible
if N == 0:
    continue

# find the values of states at the end positions
end_values = states[start_indices + 1]

for j, signj in enumerate(signs):
    # number of occurrences of this transition
    occurrences = len(end_values[end_values == signj])
    SM[i][j] = (occurrences + k)/float(N)
```

```python
print(SM)
```
What the aforementioned code does is compute the transition probabilities for each possible transition based on the number of occurrences and additive smoothing. On one of the test runs, I got the following matrix:

\[
\begin{bmatrix}
0.5047619 & 0.00952381 & 0.48571429 \\
0.33333333 & 0.33333333 & 0.33333333 \\
0.33774834 & 0.00662252 & 0.65562914
\end{bmatrix}
\]

7. Eigenvalues and eigenvectors.

To get the eigenvalues and eigenvectors we will need the `linalg` NumPy module and the `eig()` function:

```python
eig_out = numpy.linalg.eig(SM)
print(eig_out)
```

The `eig()` function returns an array containing the eigenvalues and another array containing the eigenvectors:

```plaintext
(array([ 1.        ,  0.16709381,  0.32663057]), array([[
  5.77350269e-01,   7.31108409e-01,   7.90138877e-04,
  5.77350269e-01,  -4.65117036e-01,  -9.99813147e-01,
  5.77350269e-01,  -4.99145907e-01,   1.93144030e-02]
]))
```

8. Select the eigenvector for eigenvalue 1.

Currently, we are only interested in the eigenvector for eigenvalue 1. In reality, the eigenvalue might not be exactly 1, so we should build a margin for error. We can find the index for eigenvalue between 0.9 and 1.1, as follows:

```python
idx_vec = np.where((np.abs(eig_out[0] - 1) < 0.1)
print("Index eigenvalue 1", idx_vec)
x = eig_out[1][:,idx_vec].flatten()
```

The rest of the output for this code is as follows:

```
Index eigenvalue 1 (array([0]),)
Steady state vector [ 0.57735027  0.57735027  0.57735027]
Check [ 0.57735027  0.57735027  0.57735027]
```
How it works...

The values for the eigenvector we get are not normalized. Since we are dealing with probabilities, they should sum up to one. The **diff()**, **sign()**, and **eig()** functions were introduced in this example. Their descriptions are as follows:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>diff()</td>
<td>Calculates the discrete difference. By default, the first order (see <a href="http://docs.scipy.org/doc/numpy/reference/generated/numpy.diff.html">http://docs.scipy.org/doc/numpy/reference/generated/numpy.diff.html</a>).</td>
</tr>
<tr>
<td>sign()</td>
<td>Returns the sign of array elements (see <a href="http://docs.scipy.org/doc/numpy/reference/generated/numpy.sign.html">http://docs.scipy.org/doc/numpy/reference/generated/numpy.sign.html</a>).</td>
</tr>
<tr>
<td>eig()</td>
<td>Returns the eigenvalues and eigenvectors of an array (see <a href="http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.eig.html">http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.eig.html</a>).</td>
</tr>
</tbody>
</table>

See also

- The *Installing matplotlib* recipe in Chapter 1, *Winding Along with IPython*

Discovering a power law

For the purpose of this recipe, imagine that we are operating a hedge fund. Let it sink in; you are part of the one percent now!

Power laws occur in a lot of places; see [http://en.wikipedia.org/wiki/Power_law](http://en.wikipedia.org/wiki/Power_law) for more information. In such a law, one variable is equal to the power of another:

\[ y = cx^k \]

The Pareto principle (see [http://en.wikipedia.org/wiki/Pareto_principle](http://en.wikipedia.org/wiki/Pareto_principle)) for instance, is a power law. It states that wealth is unevenly distributed. This principle tells us that if we group people by their wealth, the size of the groups will vary exponentially. To put it simply, there are not a lot of rich people, and there are even less billionaires; hence the one percent.

Assume that there is a power law in the closing stock prices log returns. This is a big assumption, of course, but power law assumptions seem to pop up all over the place.

We don't want to trade too often, because of the involved transaction costs per trade. Let's say that we would prefer to buy and sell once a month based on a significant correction (with other words a big drop). The issue is to determine an appropriate signal given that we want to initiate a transaction for every 1 out of about 20 days.
How to do it...

The following is the complete code from the `powerlaw.py` file in this book's code bundle:

```python
from matplotlib.finance import quotes_historical_yahoo
from datetime import date
import numpy as np
import matplotlib.pyplot as plt

#1. Get close prices.
today = date.today()
start = (today.year - 1, today.month, today.day)
quotes = quotes_historical_yahoo('IBM', start, today)
close = np.array([q[4] for q in quotes])

#2. Get positive log returns.
logreturns = np.diff(np.log(close))
pos = logreturns[logreturns > 0]

#3. Get frequencies of returns.
counts, rets = np.histogram(pos)
# 0 counts indices
indices0 = np.where(counts != 0)
rets = rets[:-1] + (rets[1] - rets[0])/2
# Could generate divide by 0 warning
freqs = 1.0/counts
freqs = np.take(freqs, indices0)[0]
rets = np.take(rets, indices0)[0]
freqs = np.log(freqs)

#4. Fit the frequencies and returns to a line.
p = np.polyfit(rets, freqs, 1)

#5. Plot the results.
plt.title('Power Law')
plt.plot(rets, freqs, 'o', label='Data')
plt.plot(rets, p[0] * rets + p[1], label='Fit')
plt.xlabel('Log Returns')
plt.ylabel('Log Frequencies')
plt.legend()
plt.grid()
plt.show()
```
First let's get the historical end-of-day data for the past year from Yahoo! Finance. After that, we extract the close prices for this period. These steps are described in the previous recipe:

1. Get the positive log returns.

   Now calculate the log returns for the close prices. For more information on log returns, refer to http://en.wikipedia.org/wiki/Rate_of_return.

   First we will take the log of the close prices, and then compute the first difference of these values with the `diff()` NumPy function. Let's select the positive values from the log returns:

   ```
   logreturns = np.diff(np.log(close))
   pos = logreturns[logreturns > 0]
   ```

2. Get the frequencies of the returns.

   We need to get the frequencies of the returns with the `histogram()` function. Counts and an array of the bins are returned. At the end, we need to take the log of the frequencies in order to get a nice linear relation:

   ```
   counts, rets = np.histogram(pos)
   # 0 counts indices
   indices0 = np.where(counts != 0)
   rets = rets[:-1] + (rets[1] - rets[0])/2
   # Could generate divide by 0 warning
   freqs = 1.0/counts
   freqs = np.take(freqs, indices0)[0]
   rets = np.take(rets, indices0)[0]
   freqs = np.log(freqs)
   ```

3. Fit the frequencies and returns into a line.

   Use the `polyfit()` function to do a linear fit:

   ```
   p = np.polyfit(rets, freqs, 1)
   ```

4. Plot the results.

   Finally, we will plot the data and linearly fit it with matplotlib:

   ```
   plt.title('Power Law')
   plt.plot(rets, freqs, 'o', label='Data')
   plt.plot(rets, p[0] * rets + p[1], label='Fit')
   plt.xlabel('Log Returns')
   plt.ylabel('Log Frequencies')
   plt.legend()
   plt.grid()
   plt.show()
   ```
We get a nice plot of the linear fit, returns, and frequencies, like this:

![Power Law Chart](image)

**How it works...**

The `histogram()` function calculates the histogram of a dataset. It returns the histogram values and bin edges. The `polyfit()` function fits data to a polynomial of a given order. In this case, we chose a linear fit. We discovered a power law—you have to be careful making such claims, but the evidence looks promising.

**See also**

- The *Installing matplotlib* recipe in Chapter 1, *Winding Along with IPython*
Trading periodically on dips

Stock prices periodically dip and go up. We will take a look at the probability distribution of the stock price log returns and try a very simple strategy. This strategy is based on regression towards the mean. This is a concept originally discovered in genetics by Sir Francis Galton. It was discovered that children of tall parents tend to be shorter than their parents. Children of short parents tend to be taller than their parents. Of course, this is a statistical phenomenon and doesn’t take into account fundamental factors and trends such as improvement in nutrition. Regression towards the mean is also relevant to the stock market. However, it gives no guarantees. If a company starts making bad products or makes bad investments, regression towards the mean will not save the stock.

Let’s start by downloading the historical data for a stock, for instance, AAPL. Next, we calculate the daily log returns (http://en.wikipedia.org/wiki/Rate_of_return) of the close prices. We will skip these steps since they were already done in the previous recipe.

Getting ready

If necessary, install matplotlib and SciPy. Refer to the See also section for the corresponding recipes.

How to do it...

The following is the complete code from the periodic.py file in this book’s code bundle:

```python
from __future__ import print_function
from matplotlib.finance import quotes_historical_yahoo
from datetime import date
import numpy as np
import scipy.stats
import matplotlib.pyplot as plt

#1. Get close prices.
today = date.today()
start = (today.year - 1, today.month, today.day)
quotes = quotes_historical_yahoo('AAPL', start, today)
close = np.array([q[4] for q in quotes])

#2. Get log returns.
logreturns = np.diff(np.log(close))

#3. Calculate breakout and pullback
```
freq = 0.02
breakout = scipy.stats.scoreatpercentile(logreturns, 100 * (1 - freq))
pullback = scipy.stats.scoreatpercentile(logreturns, 100 * freq)

# Generate buys and sells
buys = np.compress(logreturns < pullback, close)
sells = np.compress(logreturns > breakout, close)
print(len(buys), len(sells))
print(sells.sum() - buys.sum())

# Plot a histogram of the log returns
plt.title('Periodic Trading')
plt.hist(logreturns)
plt.grid()
plt.xlabel('Log Returns')
plt.ylabel('Counts')
plt.show()

Now comes the interesting part:

1. Calculate the breakout and pullback.

   Let’s say we want to trade five times a year, or roughly, every 50 days. One strategy would be to buy when the price drops by a certain percentage (a pullback), and sell when the price increases by another percentage (a breakout).

   By setting the percentile appropriate for our trading frequency, we can match the corresponding log returns. SciPy offers the scoreatpercentile() function, which we will use:

   freq = 0.02
   breakout = scipy.stats.scoreatpercentile(logreturns, 100 * (1 - freq))
pullback = scipy.stats.scoreatpercentile(logreturns, 100 * freq)

2. Generate buys and sells.

   Use the compress() NumPy function to generate buys and sells for our close price data. This function returns elements based on a condition:

   buys = np.compress(logreturns < pullback, close)
sells = np.compress(logreturns > breakout, close)
print(buys)
print(sells)
print(len(buys), len(sells))
print(sells.sum() - buys.sum())
The output for AAPL and a 50-day period is as follows:

\[
\begin{array}{cccccc}
77.76375466 & 76.69249773 & 102.72 & 101.2 & 98.57 \\
74.95502967 & 76.55980292 & 74.13759123 & 80.93512599 & 98.22 \\
\end{array}
\]

Thus, we have a loss of 52 dollars if we buy and sell an AAPL share five times. When I ran the script, the entire market was in recovery mode after a correction. You may want to look at not just the AAPL stock price but maybe the ratio of AAPL and SPY. SPY can be used as a proxy for the U.S. stock market.

3. Plot a histogram of the log returns.

Just for fun, let's plot the histogram of the log returns with matplotlib:

```python
plt.title('Periodic Trading')
plt.hist(logreturns)
plt.grid()
plt.xlabel('Log Returns')
plt.ylabel('Counts')
plt.show()
```

This is what the histogram looks like:
How it works...

We encountered the `compress()` function, which returns an array containing the array elements of the input that satisfy a given condition. The input array remains unchanged.

See also

- The `Installing matplotlib` recipe in Chapter 1, Winding Along with IPython
- The `Installing SciPy` recipe in Chapter 2, Advanced Indexing and Array Concepts
- The `Discovering a power law` recipe in this chapter
- The documentation page for the `compress()` function at http://docs.scipy.org/doc/numpy/reference/generated/numpy.compress.html

Simulating trading at random

In the previous recipe, we tried out a trading idea. However, we have no benchmark that can tell us whether the result we got was any good. It is common in such cases to trade at random under the assumption that we should be able to beat a random process. We will simulate trading by taking some random days from a trading year. This should illustrate working with random numbers using NumPy.

Getting ready

If necessary, install matplotlib. Refer to the See also section of the corresponding recipe.

How to do it...

The following is the complete code from the `random_periodic.py` file in this book's code bundle:

```python
from __future__ import print_function
from matplotlib.finance import quotes_historical_yahoo
from datetime import date
import numpy as np
import matplotlib.pyplot as plt

def get_indices(high, size):
    #2. Generate random indices
    return np.random.randint(0, high, size)

    #1. Get close prices.
```

```python
```
today = date.today()
start = (today.year - 1, today.month, today.day)

quotes = quotes_historical_yahoo('AAPL', start, today)
close = np.array([q[4] for q in quotes])

nbuys = 5
N = 2000
profits = np.zeros(N)

for i in xrange(N):
    #3. Simulate trades
    buys = np.take(close, get_indices(len(close), nbuys))
    sells = np.take(close, get_indices(len(close), nbuys))
    profits[i] = sells.sum() - buys.sum()

print("Mean", profits.mean())
print("Std", profits.std())

#4. Plot a histogram of the profits
plt.title('Simulation')
plt.hist(profits)
plt.xlabel('Profits')
plt.ylabel('Counts')
plt.grid()
plt.show()

First we need an array filled with random integers:

1. Generate random indices.
   You can generate random integers with the randint() NumPy function. This will be linked to random days of a trading year:
   
   return np.random.randint(0, high, size)

2. Simulate trades.
   You can simulate trades with the random indices from the previous step. Use the take() NumPy function to extract random close prices from the array of step 1:
   
   buys = np.take(close, get_indices(len(close), nbuys))
   sells = np.take(close, get_indices(len(close), nbuys))
   profits[i] = sells.sum() - buys.sum()

3. Plot a histogram of the profits for a large number of simulations:
   
   plt.title('Simulation')
plt.hist(profits)
plt.xlabel('Profits')
plt.ylabel('Counts')
plt.grid()
plt.show()

Here is a screenshot of the resulting histogram of 2,000 simulations for AAPL, with five buys and sells in a year:

![Histogram of 2,000 simulations for AAPL](image)

**How it works...**

We used the `randint()` function, which can be found in the `numpy.random` module. This module contains more convenient random generators, as described in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rand()</code></td>
<td>Creates an array from a uniform distribution over [0,1] with a shape based on dimension parameters. If no dimensions are specified, a single float is returned (see <a href="http://docs.scipy.org/doc/numpy/reference/generated/numpy.random.rand.html">http://docs.scipy.org/doc/numpy/reference/generated/numpy.random.rand.html</a>).</td>
</tr>
<tr>
<td><code>randn()</code></td>
<td>Sample values from the normal distribution with mean 0 and variance 1. The dimension parameters function the same as for <code>rand()</code> (see <a href="http://docs.scipy.org/doc/numpy/reference/generated/numpy.random.randn.html">http://docs.scipy.org/doc/numpy/reference/generated/numpy.random.randn.html</a>).</td>
</tr>
<tr>
<td><code>randint()</code></td>
<td>Returns an integer array given a low boundary, an optional high bound, and an optional output shape (see <a href="http://docs.scipy.org/doc/numpy/reference/generated/numpy.random.randint.html">http://docs.scipy.org/doc/numpy/reference/generated/numpy.random.randint.html</a>).</td>
</tr>
</tbody>
</table>
See also

- The Installing matplotlib recipe in Chapter 1, Winding Along with IPython

**Sieving integers with the Sieve of Eratosthenes**

The **Sieve of Eratosthenes** (see [http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes](http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes)) is an algorithm that filters prime numbers. It iteratively identifies multiples of found primes. The multiples are, by definition, not primes and can be eliminated. This sieve is efficient for primes less than 10 million. Let's now try to find the 10001st prime number.

**How to do it...**

The first mandatory step is to create a list of natural numbers:

1. Create a list of consecutive integers. NumPy has the `arange()` function for that:
   
   ```
   a = np.arange(i, i + LIM, 2)
   ```

2. Sieve out the multiples of $p$.

   We are not sure if this is what Eratosthenes wanted us to do, but it works. In the following code, we are passing a NumPy array and getting rid of all the elements that have a zero remainder when divided by $p$:
   
   ```
   a = a[a % p != 0]
   ```

   The following is the entire code for this problem:

   ```
   from __future__ import print_function
   import numpy as np

   LIM = 10 ** 6
   N = 10 ** 9
   P = 10001
   primes = []
   p = 2

   # By listing the first six prime numbers: 2, 3, 5, 7, 11, and 13, we can see that the 6th prime is 13.

   # What is the 10 001st prime number?

   def sieve_primes(a, p):
   ```
#2. Sieve out multiples of p
a = a[a % p != 0]

return a

for i in xrange(3, N, LIM):
    #1. Create a list of consecutive integers
    a = np.arange(i, i + LIM, 2)

    while len(primes) < P:
        a = sieve_primes(a, p)
        primes.append(p)

        p = a[0]

    print(len(primes), primes[P-1])
Where to buy this book
Click here for ordering and shipping details.