SciPy
Programming

Succinctly®

by James McCaffrey

Syncfusion® | Technology Resource Portal

SciPy Programming Succinctly

By
James McCaffrey

Foreword by Daniel Jebaraj

Syncfusion®
Deliver innovation with ease

see more please visit: https://homeofpdf.com

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The Succinctly series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

see more please visit: https://homeofpdf.com

About the Author

James McCaffrey works for Microsoft Research in Redmond, WA. He holds a B.A. in psychology from the University of California at Irvine, a B.A. in applied mathematics from California State University at Fullerton, an M.S. in information systems from Hawaii Pacific University, and a doctorate from the University of Southern California. James enjoys exploring all forms of activity that involve human interaction and combinatorial mathematics, such as the analysis of betting behavior associated with professional sports, machine learning algorithms, and data mining.

Chapter 1 Getting Started

The SciPy library (Scientific Python, pronounced "sigh-pie") is an open source extension to the Python language. When Python was first released in 1991, the language omitted an array data structure by design. It quickly became apparent that an array type and functions that operate on arrays would be needed for numeric and scientific computing.

The SciPy stack has three components: Python, NumPy, and SciPy. The Python language has basic features, such as loop control statements and a general purpose list data structure. The NumPy library (Numerical Python) has array and matrix data structures plus some relatively simple functions such as array search. The SciPy library, which requires NumPy, has many intermediate and advanced functions that work with arrays and matrices. There is some overlap between SciPy and NumPy, meaning there are some functions that are in both libraries.

When SciPy was first released in 2001, it contained built-in array and matrix types. In 2006, the array and matrix functionality from SciPy was moved into a newly created NumPy library so that programmers who needed just an array type didn't have to import the entire SciPy library. Because of the dependency, the term SciPy also refers to NumPy.

This e-book makes no assumptions about your background or experience. Even if you have no Python programming experience at all, you should be able to follow along with a bit of effort.

Each section of this e-book presents a complete demo program. Every programmer I know, including me, learns how to program in a new language by getting an example program up and running, and then experimenting by making changes. So if you want to learn SciPy, copy and paste the source code from the demo programs, run the programs, and then fiddle with the programs. Find the code samples in Syncfusion's Bitbucket repository.

The approach I take in this e-book is not to present hundreds of one-line SciPy examples. Instead, I've tried to pick key examples that give you the knowledge you need to learn SciPy quickly. For example, section 5.4 explains how the `normal()` function generates random values. Once you understand the `normal()` function, you can easily figure out how to use the 35 other distribution functions, such as the `poisson()` and `exponential()` functions.

In my opinion, the most difficult part of learning any programming language or technology is getting a first program to run. After that, it's just details. But getting started can be frustrating. The purpose of this first chapter is to make sure you can install SciPy and run a program.

In section 1.1, you'll learn how to install the software required to access the SciPy library. In particular, you'll see how to install the Anaconda distribution, which includes Python, SciPy, NumPy, and many related and useful packages. You'll also learn how to install SciPy separately if you have an existing instance of Python installed. In section 1.2, you'll learn how to edit and execute Python programs that use the SciPy and NumPy libraries. In section 1.3, you'll learn a bit about program structure and style when using SciPy and NumPy. Section 1.4 presents a quick reference for NumPy and SciPy.

Enough chit-chat. Let's get started.

1.1 Installing SciPy and NumPy

It's no secret that the best way to learn a programming language, library, or technology is to use it. Unlike the installation process for many Python libraries, installing SciPy is not trivial. Briefly, the crux of the difficulty is that SciPy and NumPy contain hooks to C language routines.

It is possible to first install Python, and then install the SciPy and NumPy packages separately from source code using the pip (PIP Installs Packages) utility program, but this approach can be troublesome. I recommend that you either use the Anaconda distribution bundle or, if you install Python, NumPy, and SciPy separately, that you use a pre-built binary installer for NumPy and SciPy.

Note: The terms **package***, **module***, and* **library** *have different meanings but are often used more or less interchangeably.*

There are several advantages to using Anaconda. There are binary installers for Windows, OS X, and Linux. The distribution comes with Python, NumPy, and SciPy, as well as many other related packages. This means you have one relatively easy installation procedure. The distribution comes with the conda open source package and environment manager, which means you can work with multiple versions of Python. In other words, even if you already have Python installed, Anaconda will let you work with a new Python + SciPyPy installation without resource conflicts. Anaconda also comes with two nice Python editors, IDLE and Spyder.

The open source Anaconda distribution is maintained by the Continuum Analytics company at http://www.continuum.io/. Let's walk through the installation process, step by step. I'll show you screenshots for a Windows installation, but you should have little trouble installing on OS X or any flavor of Linux. First, use your web browser of choice to go to the Continuum Analytics site, and then locate the download link and click on it.
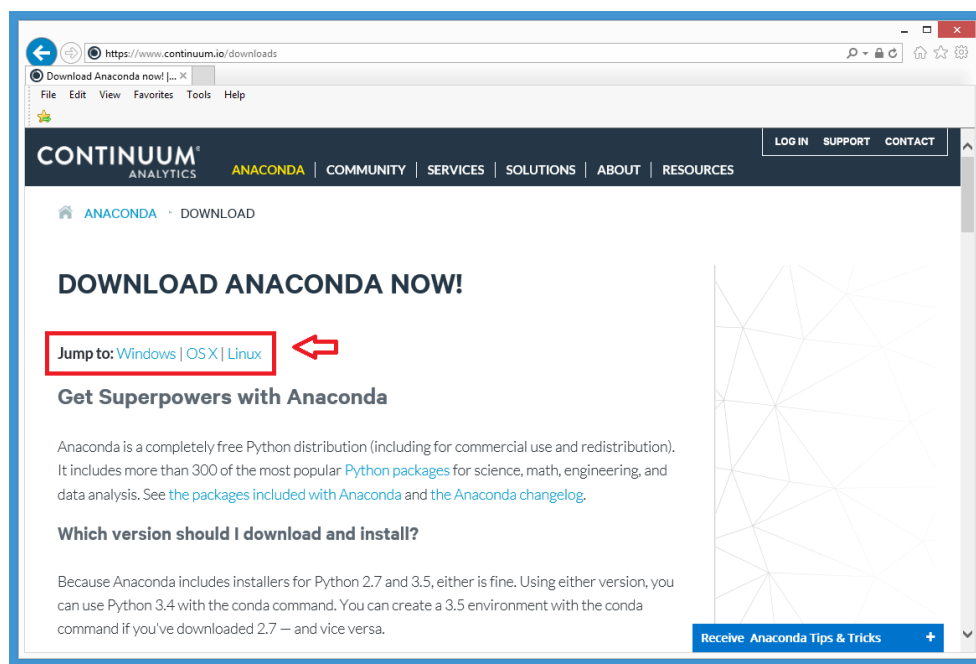
Figure 1: The Anaconda Download Site

Next, locate the link to your appropriate operating system and click on it.

At this point you must choose between Python version 2.x and Python version 3.x. If you're new to Python, the essential point is that the two versions are not fully compatible. Python users can have strong opinions about which Python version they prefer, but for use with SciPy, I recommend using Python 2.7 in order to maintain compatibility with older functions.
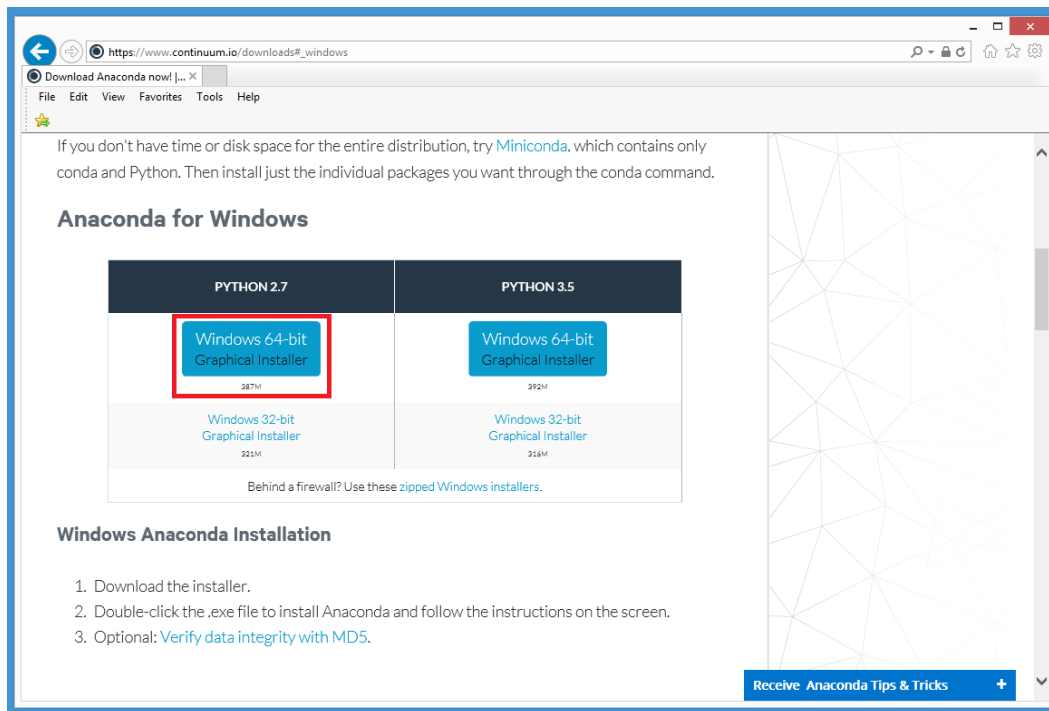
Figure 2: Python Version Selection

After selecting the Python version, you should see a message asking if you want to save the self-extracting executable installer, or if you want to run the installer immediately. You can do either. I chose the **Run** option.
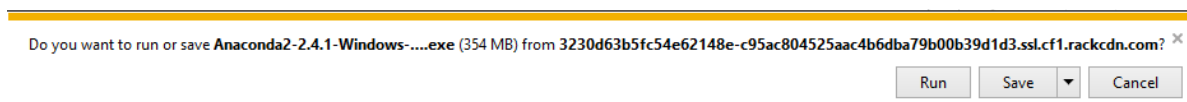
Figure 3: Run the Executable Installer

The installation process begins by displaying a welcome splash screen. Notice that the Anaconda distribution number (2.4.1 in this case) is not the same as the Python version number (2.7).
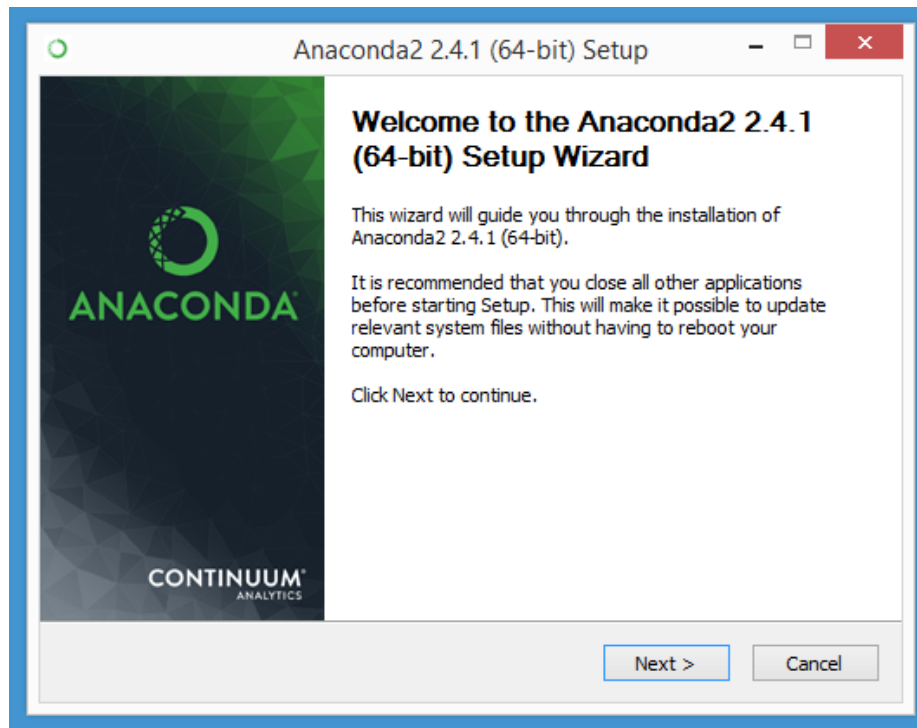
Figure 4: The Welcome Splash

After clicking **Next**, you'll be presented with a license agreement, which you can read if you're a glutton for legal jargon punishment. Click **I Agree**.
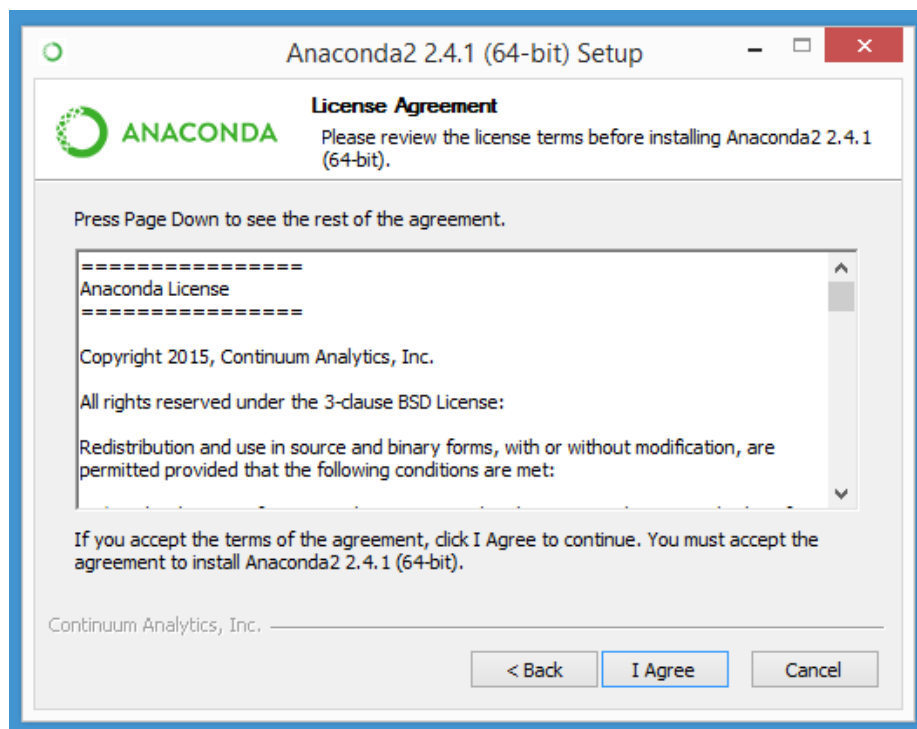
Figure 5: License Agreement

Next, you'll have the option of installing for all users or just for the current user (presumably you). I suggest using Anaconda's recommendation.
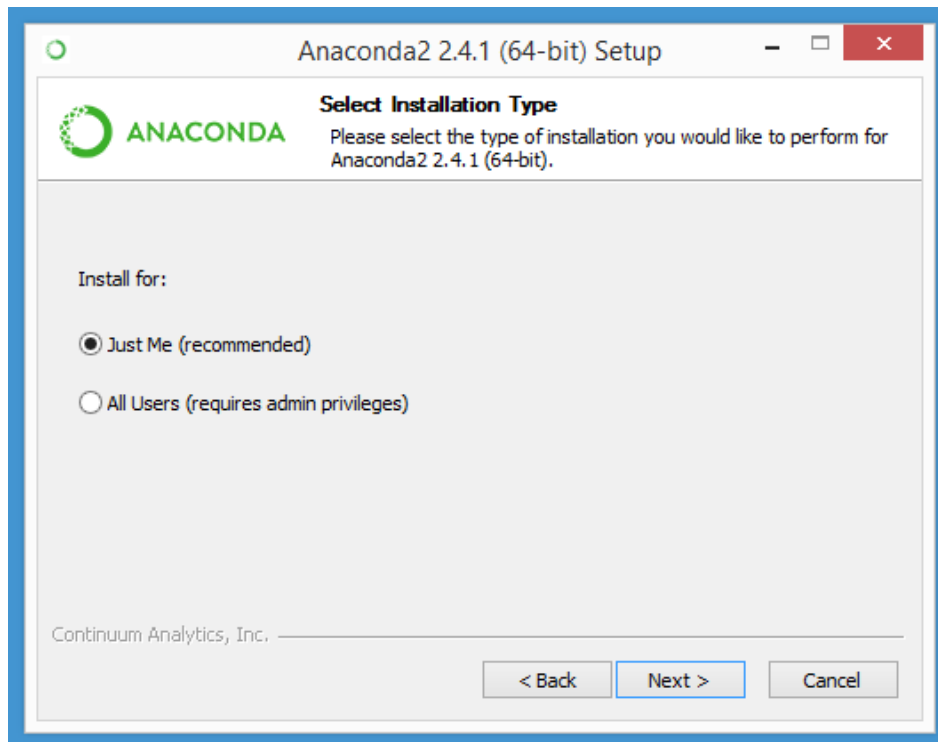
Figure 6: User Options

Then, you'll need to specify the installation root directory. With open source software such as Python, it's normal to install programs in a directory located off drive C rather than in the C:\Program Files directory. I recommend installing at C:\Anaconda2.
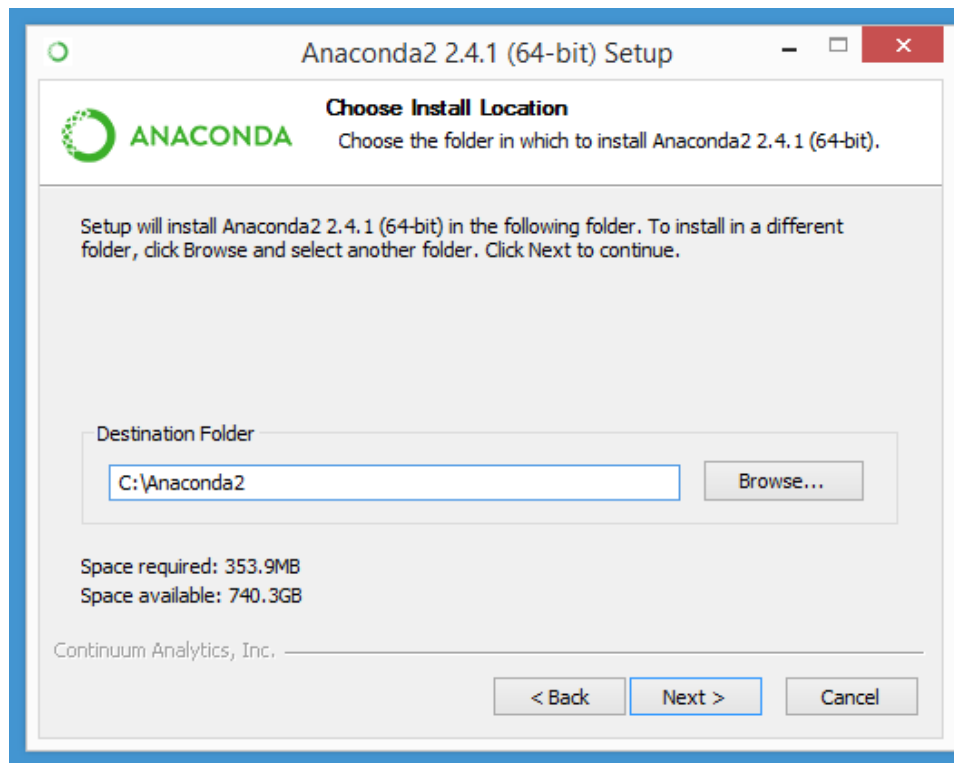
Figure 7: The Installation Directory

Next, you'll get an option to add the locations of the Anaconda executables to the System PATH variable, and an option to register the Anaconda Python as the default. Select both check boxes and click **Install**.
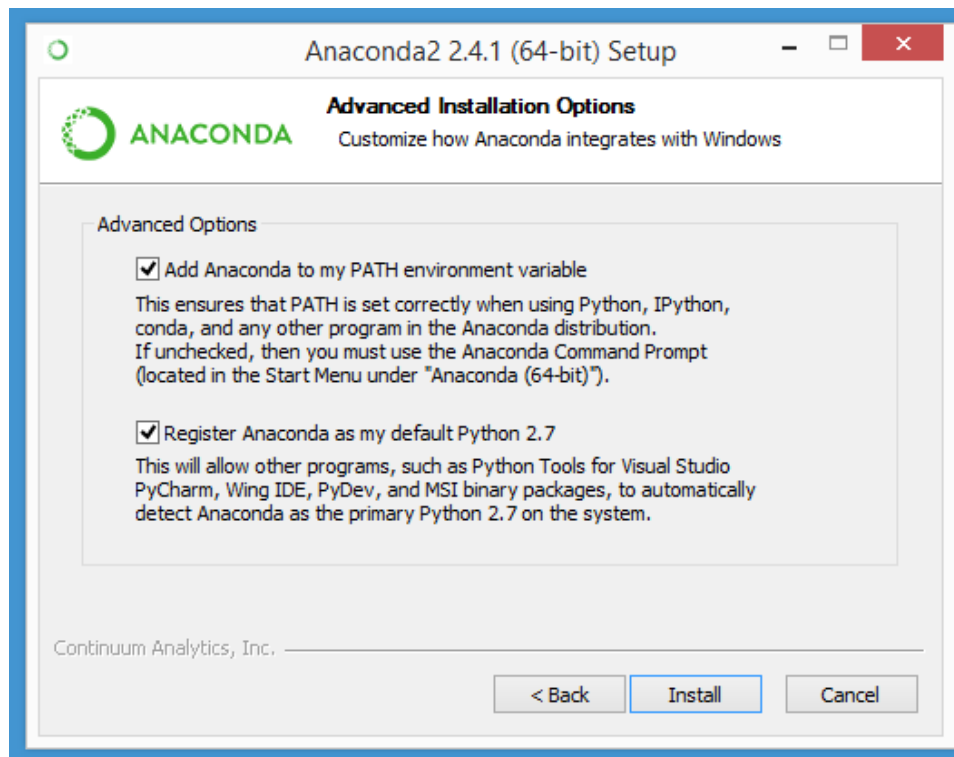
Figure 8: The PATH and Integration Options

You'll see a progress bar during the installation process. Notice that NumPy and SciPy are included in the installation components.
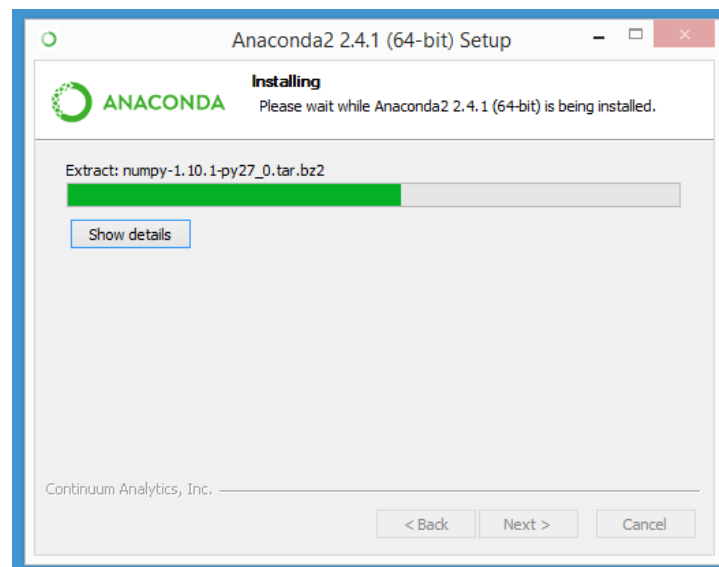
Figure 9: Anaconda Includes SciPy and NumPy

When the installation is complete, you'll see an "Installation Complete" message. If there are any errors during the installation, they'll appear here. If so, you can read the error messages, fix whatever is wrong, delete the root installation directory, and try again.
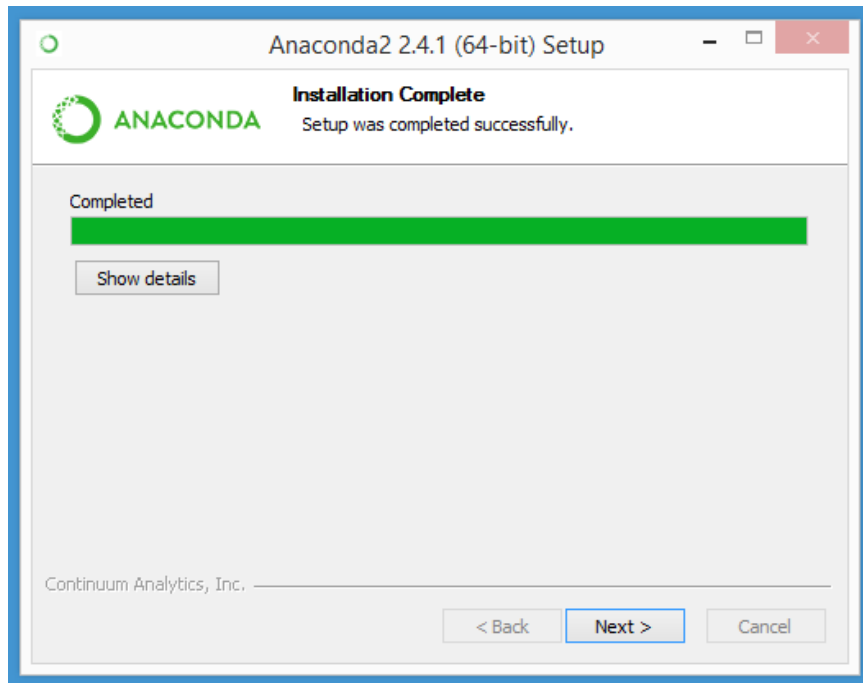
Figure 10: Installation Completed

After you click **Next**, you'll see a final completion confirmation message. You can click **Finish**.
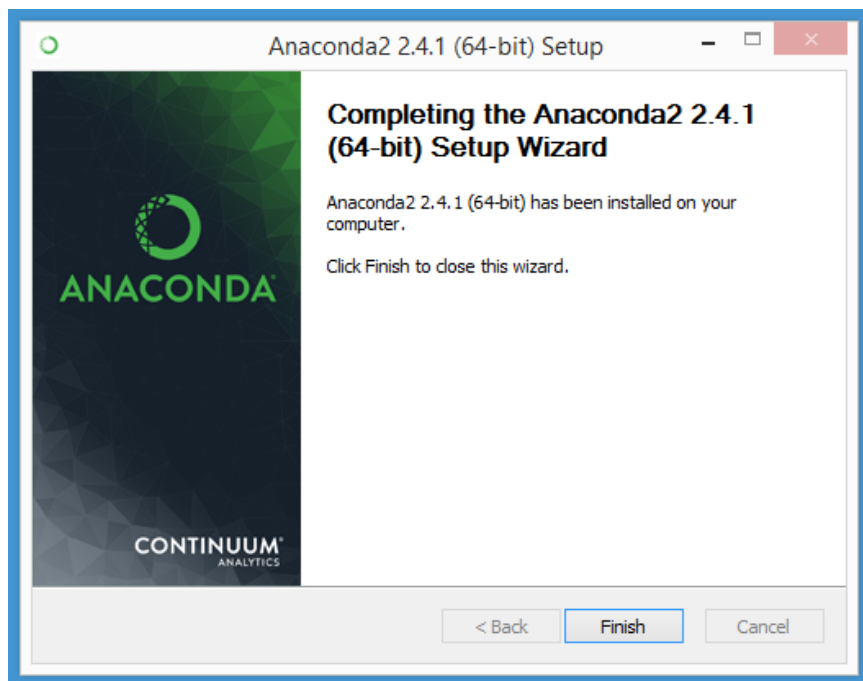
Figure 11: Installation Confirmation

The last step of the Anaconda installation process is to verify that your installation is working. First, verify that Python is up and running. Launch a command shell and navigate to the root directory by entering a **cd ** command. Then type the command:

```
C:\> python --version
```

If Python responds with information about a version, then Python is almost certainly installed correctly, but you should now verify this by executing a Python statement. At the command prompt, enter the command **python** (I've included a space after the prompt for readability):
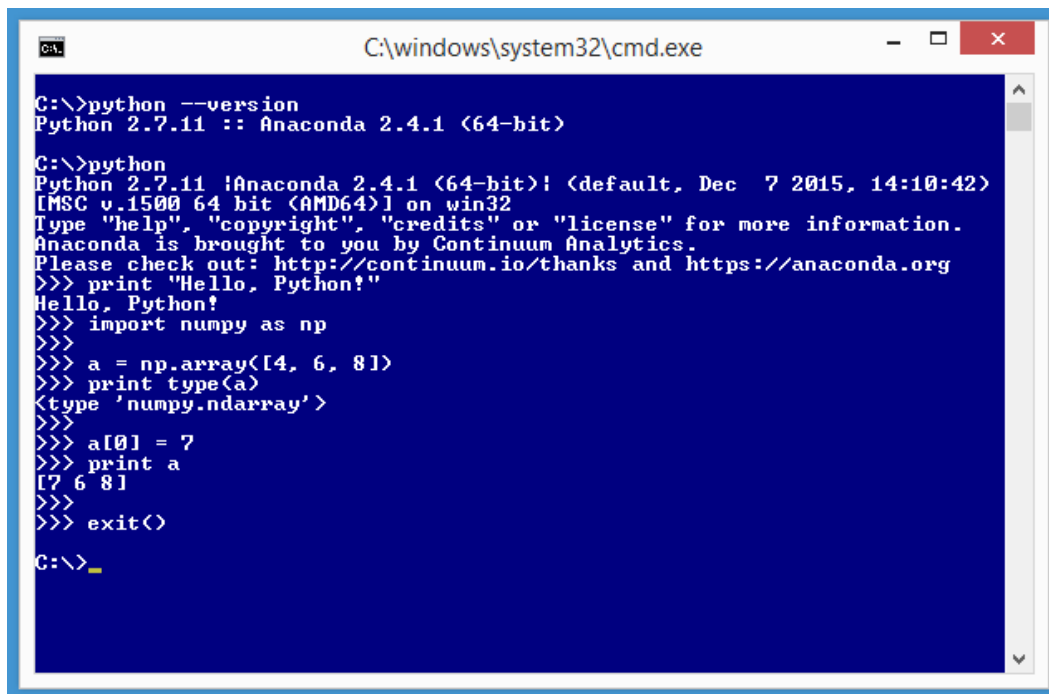
```
C:\> python
```

This will start the Python interpreter, which will be indicated by the three greater-than characters prompt. Instruct Python to print a traditional first message:

```
>>> print "Hello, Python!"
```

Finally, verify that NumPy is installed correctly by creating and manipulating an array. Enter the following commands at the Python prompt:

```
>>> import numpy as np
>>> a = np.array([4, 6, 8])
>>> print type(a)
>>> a[0] = 7
>>> print a
>>> exit()
```


Figure 12: Verify Your Installation

The **import** statement brings the NumPy library into scope so it can be used. The **np** alias could have been something else, but **np** is customary and good.

The statement `a = np.array([4, 6, 8])` creates an array named `a` with three cells with integer values 4, 6, and 8. The Python `type()` function tells you that `a` is, in fact, an array (technically an `ndarray`, which stands for an n-dimensional array).

The statement `a[0] = 7` sets the value in the first cell of the array to 7, overwriting the original value of 2. The point here is that NumPy arrays, like those in most languages, use 0-based indexing. Congratulations! You have all the software you need to explore SciPy and NumPy.

Installing Python, NumPy, and SciPy separately

Instead of using the Anaconda distribution, you can install Python, NumPy, and SciPy separately. To install Python, go to https://www.python.org/downloads/ and select the download for either a 3.x or 2.x version. I recommend the 2.7 version. After installation is complete, add C:\Python27 (or the location of python.exe if you used a non-default location) and C:\Python27\Lib\idelib to your system PATH environment variable. To install NumPy and SciPy, I strongly recommend that you use pre-built executable installers. In particular, the ones I recommend are maintained at the SourceForge repository. Install NumPy first. Go to http://sourceforge.net/projects/numpy/files/NumPy/.

That site has different versions of both NumPy and SciPy. Go into the directory of the version you wish to install. I recommend using a recent version that has the most downloads. Go into a version directory, and then look for a file named something like numpy-1.10.2-win32-superpack-python2.7.exe.
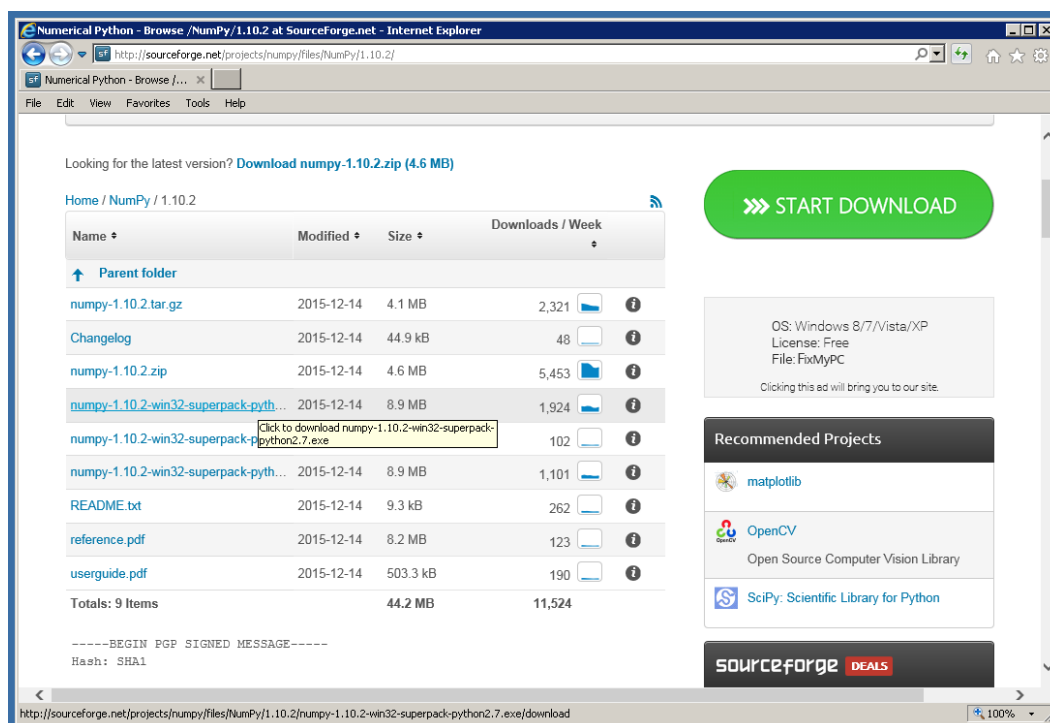
Figure 13: Binary Installer Link for NumPy

Make sure you have the version that corresponds to your Python version, then click on the link and you'll get the option to run the installer.

You'll have the option to either run the installer program immediately, or save the installer so you can run it later. I usually choose the **Run** option.
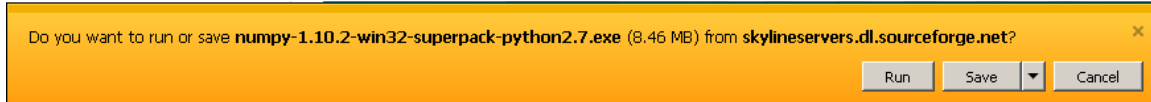
Figure 14: Run the NumPy Installer Executable

After you click **Run**, the installer will launch and present you with an installation wizard. Click **Next**.
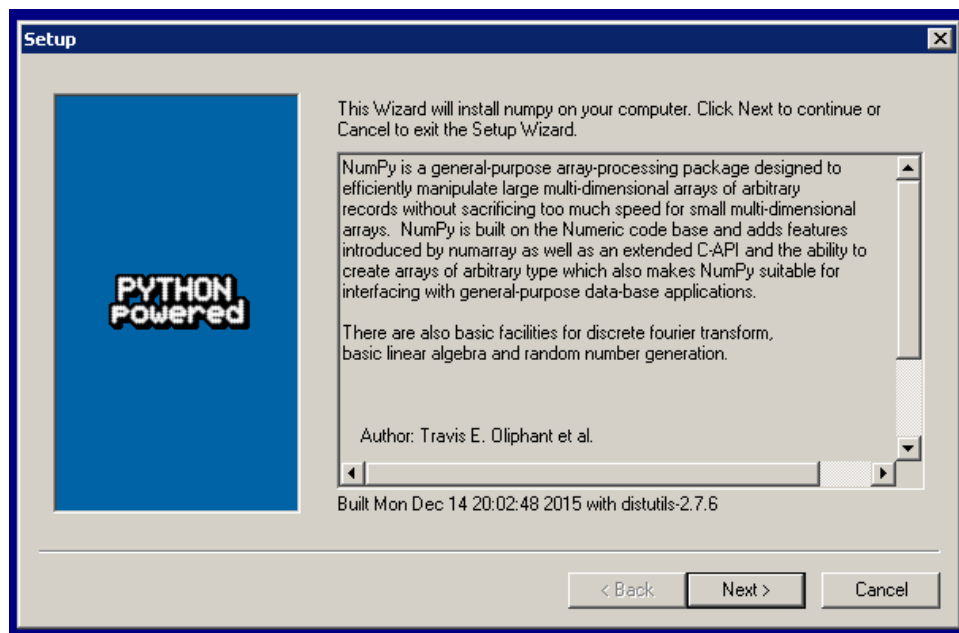
Figure 15: NumPy Installation Wizard

The installer should find your existing Python installation and recommend an installation directory for the NumPy library.
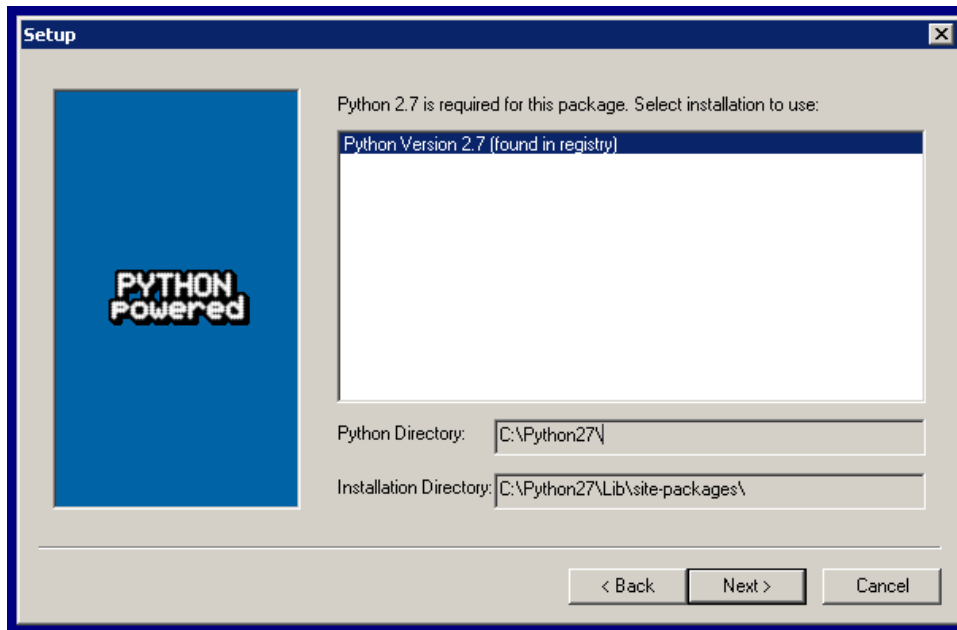
Figure 16: The NumPy Installer Finds Existing Python Installation

Click **Next** on the next few wizard pages and you'll complete the NumPy installation. You can verify NumPy was installed by launching a Python shell and entering the command **import numpy**. If no error message results, NumPy has been installed.
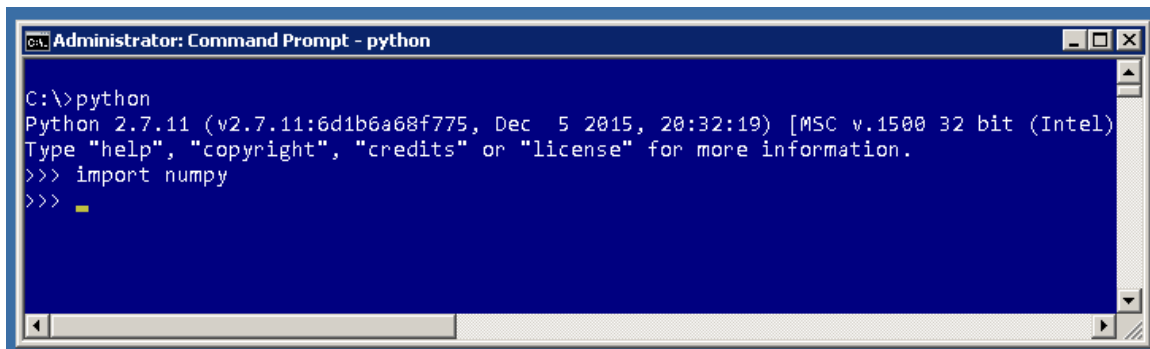
Figure 17: Verify Separate NumPy Installation

Now you can install the SciPy library from SourceForge using the exact same process.

Resources

For installation information, including alternatives to the Anaconda distribution, see http://www.scipy.org/install.html.

1.2 Editing SciPy programs

Although Python and SciPy can be used interactively, for many scenarios you'll want to write and execute a program (technically a script). If you have installed the Anaconda distribution, you have three main ways to edit and execute a Python program. First, you can use any simple text editor, such as Notepad, and execute from a command line. Second, you can edit and execute programs using the IDLE (Integrated DeveLopment Environment) program. Third, you can edit and execute using the Spyder program. I'll walk you through each approach.

Code Listing 1: A Simple SciPy/NumPy Program

```python
# test.py

import numpy as np
import scipy as sp

print "\nHello from test.py"

a = np.array([2, 4, 6, 8])
print a

length = a.size   # 4
a[length-1] = 9
print a

print "Goodbye from test.py"
```

Launch Notepad and type or copy and paste the statements shown in Code Listing 1. Save the program as test.py in any directory, such as C:\SciPy. If you use Notepad, be sure it doesn't add an extra .txt extension to the file name.
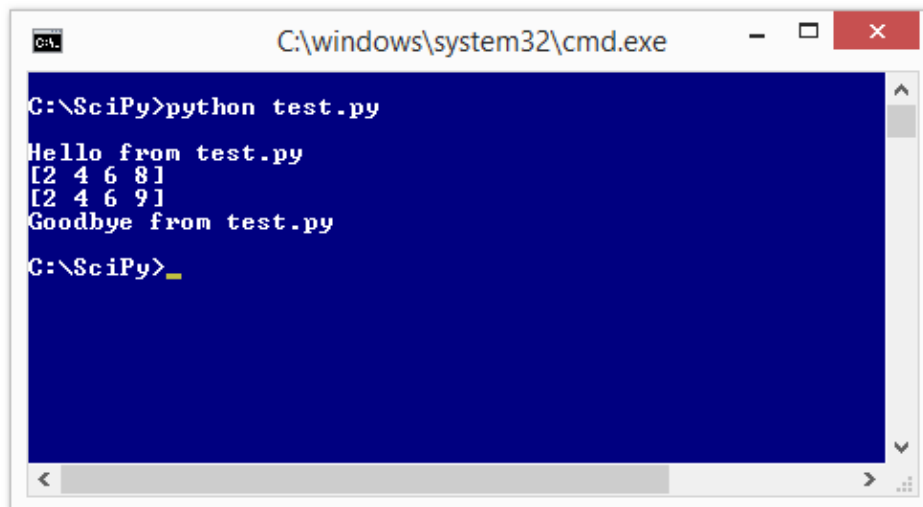
Figure 18: Executing from a Command Prompt

Launch a Command Prompt (Windows) or command shell such as bash (Linux). Navigate to the directory containing file test.py. Execute the program by entering the command:

```
> python test.py
```

Using Notepad as an editor and executing from a shell is simple and effective, but I recommend using either IDLE or Spyder. The idle.bat launcher file is typically located by default in the C:\Python27\Lib\idelib directory. To start the IDLE program, launch a command shell, navigate to the location of the .bat file if that directory is not in your PATH variable, and enter the command **idle**.

This will start a special Python Shell as shown in the top part of Figure 19.
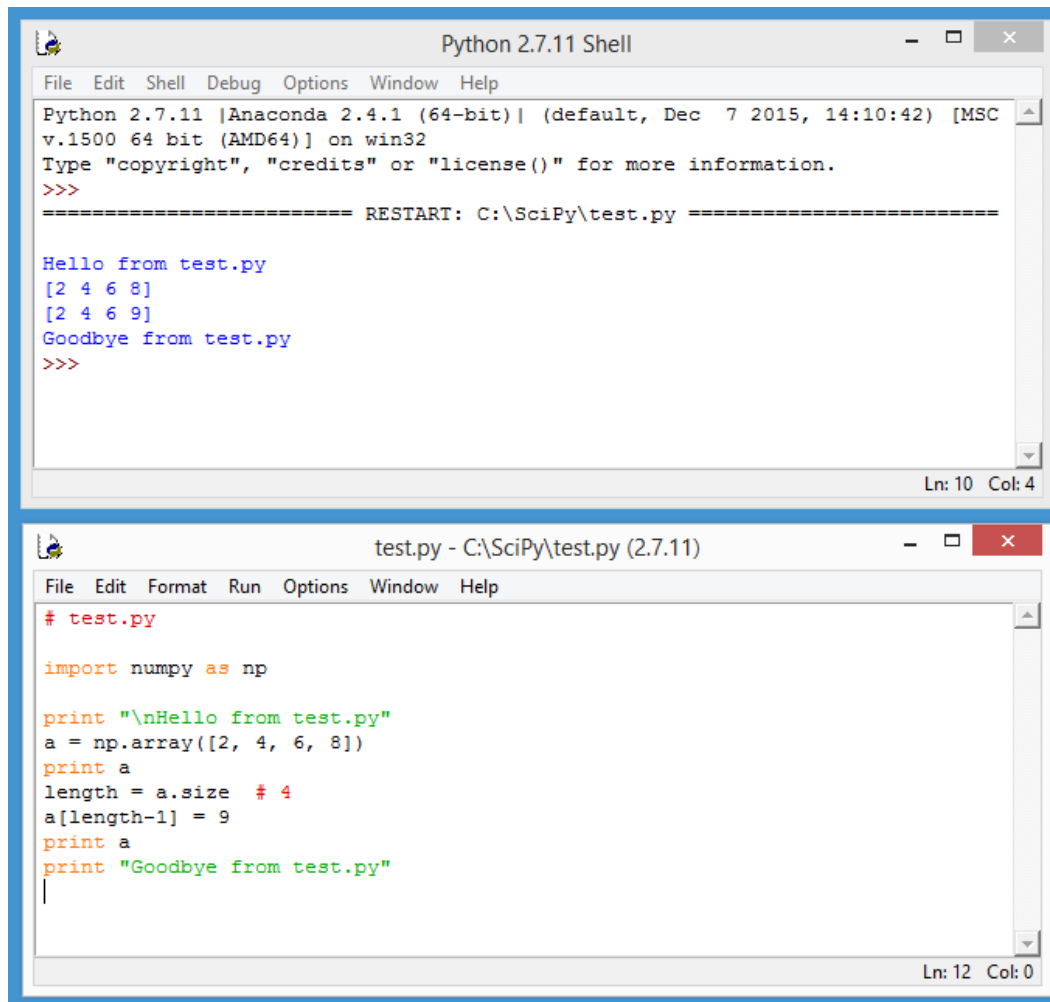
Figure 19: Using the IDLE Program

From the Python Shell menu bar, click **File** > **New File**. This will launch a similar-looking editor, as shown in the bottom part of Figure 19. Now, type or copy and paste the program in Code Listing 1 into the IDLE editor. Save the program as test.py in any convenient directory using **File** > **Save**. Execute the program by clicking **Run** > **Run Module**, or pressing the F5 shortcut key.

Program output is displayed in the Python Shell window. Some experienced Python users criticize IDLE for being too simple and lacking sophisticated editing and debugging features, but I like IDLE a lot and it's my SciPy programming environment of choice in most situations.
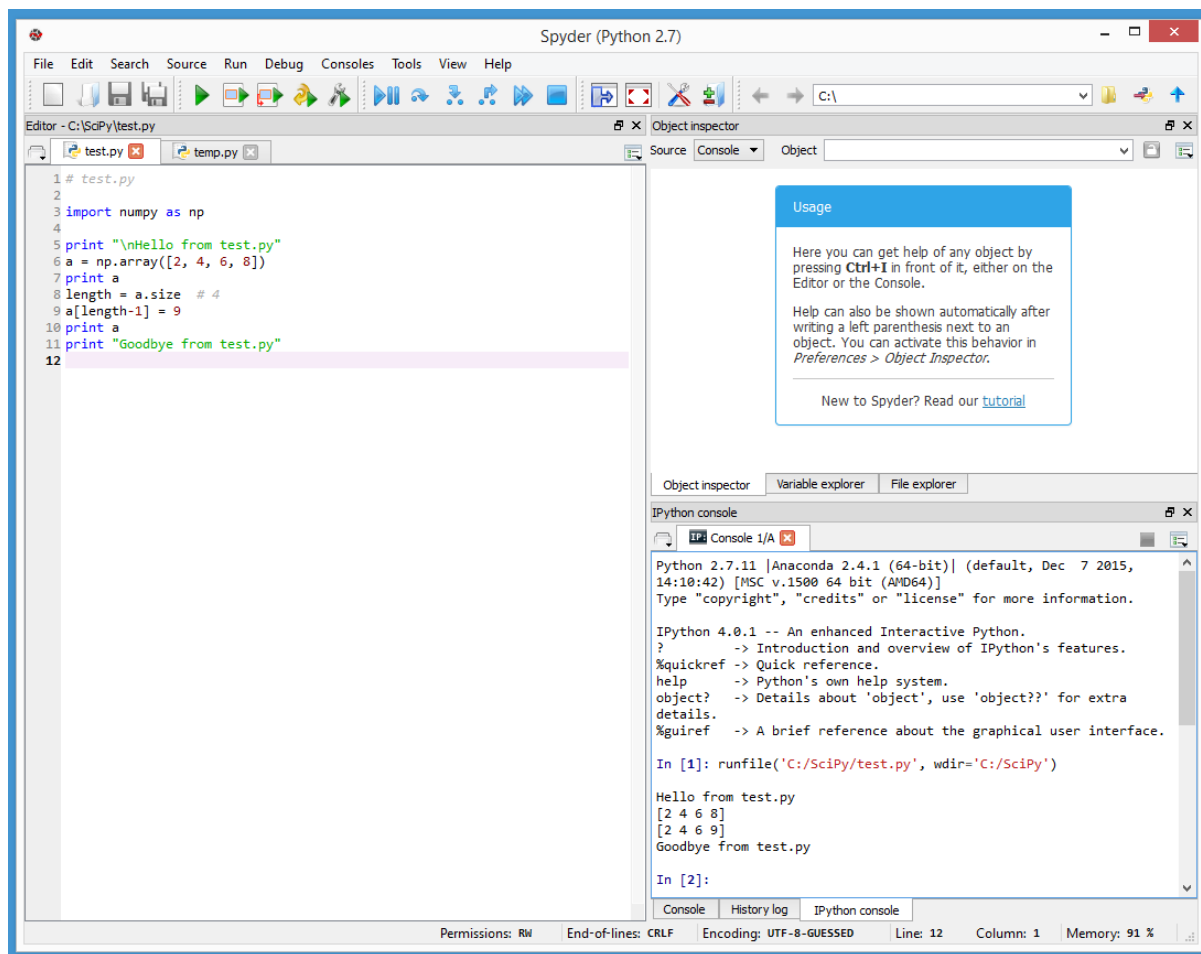
Figure 20: Using the Spyder Program

The Anaconda distribution comes with the open source Spyder (Scientific PYthon Development EnviRonment) program. To start Spyder, launch a command shell and enter:

```
> spyder
```

Type or copy and paste the program from Code Listing 1 into the Spyder editor window on the left side. You can either save first using **File** > **Save** or execute immediately by clicking **Run** > **Run**. Program output appears in the lower right window.

Resources

If you use Visual Studio, consider the Python Tools for Visual Studio (PTVS) plugin at http://microsoft.github.io/PTVS/.

If you use the Eclipse IDE, you might want to take a look at the PyDev plugin at http://www.pydev.org/.

1.3 Program structure

Because the Python language is so flexible, there are many ways to structure a program. Some experienced Python programmers have strong opinions about what constitutes good Python program structure. Other programmers, like me, believe that there's no single best program structure suitable for all situations.

Take a look at the demo program in Code Listing 2. The program begins with comments indicating the program file name and Python version. Because the Python 2.x and Python 3.x versions are not fully compatible, it's a good idea to indicate which version your program is using. If you are using Linux, you can optionally use a shebang like **#!/usr/bin/env python** as the very first statement.

Code Listing 2: Python Program Structure Demo

```python
# structure.py
# Python 2.7

import numpy as np

def make_x(n):
  result = np.zeros((n,n))
  for i in xrange(n):
    for j in xrange(n):
      if i == j or (i + j == n-1):
        result[i,j] = 1.0
  return result

def main():
  print "\nBegin program structure demo \n"

  try:
    n = 5
    print "X matrix with size n = " + str(n) + " is "
    mx = make_x(n)
    print mx
    print ""

    n = -1
    print "X matrix with size n = " + str(n) + " is "
    mx = make_x(n)
    print mx
    print ""
  except Exception, e:
    print "Error: " + str(e)

  print "\nEnd demo \n"

if __name__ == "__main__":
  main()
```

```
C:\SciPy\Ch1> python structure.py
Begin program structure demo

X matrix with size n = 5 is
[[ 1.  0.  0.  0.  1.]
 [ 0.  1.  0.  1.  0.]
 [ 0.  0.  1.  0.  0.]
 [ 0.  1.  0.  1.  0.]
 [ 1.  0.  0.  0.  1.]]

X matrix with size n = -1 is
Error: negative dimensions are not allowed

End demo
```

Next, the demo program imports the NumPy library and assigns a short alias:

import numpy as np

This idiom is standard for NumPy and SciPy programming and I recommend that you use it unless you have a specific reason for not doing so. Next, the demo creates a program-defined function named **make_x()**:

```
def make_x(n):
  result = np.zeros((n,n))
  for i in xrange(n):
    for j in xrange(n):
      if i == j or (i + j == n-1):
        result[i,j] = 1.0
  return result
```

The **make_x()** function accepts a matrix dimension parameter **n** (presumably an odd integer) and returns a NumPy matrix with 1.0 values on the main diagonal (upper-left cell to lower-right cell) and the minor diagonal, and 0.0 values elsewhere.

The demo uses an indentation of two spaces instead of the widely recommended four spaces. I use two-space indentation throughout this e-book mostly to save space, but to be honest, I prefer using two spaces, anyway.

The demo program defines a **main()** function that is the execution entry point:

```
def main():
  print "\nBegin program structure demo \n"
  # rest of calling statements here
  print "\nEnd demo \n"

if __name__ == "__main__":
  main()
```

The program-defined **main()** function is called using the **__main__** mechanism (note: there are two underscore characters before and after the word *main*). Defining a **main()** function has several advantages compared to simply placing the program's calling statements after import statements and function definitions.

The primary downside to using a **main()** function in your program is simply the extra time and space it takes you to write the program. Throughout the rest of this e-book, I do not use a **main()** function, just to save space.

By default, when the Python interpreter reads a source .py file, it will execute all statements in the file. However, just before beginning execution, the interpreter sets a system **__name__** variable to the value **__main__** for the source file that started execution. The value of the **__name__** variable for any other module that is called is set to the name of the module.

In other words, the interpreter knows which program or module is the main one that started execution and will execute just the statements in that program or module. Put another way, Python modules that don't have an **if __name__ == "__main__"** statement will not be automatically executed. This mechanism allows you to write Python code and then import that code into another module. In effect, this allows you to write library modules.

Additionally, by using a **main()** function, you can avoid program-defined variable and function names clashing with Python system names and keywords. Finally, using a **main()** function gives you more control over control flow if you use the try-except error handling mechanism.

The demo program uses double quote characters to delimit strings. Unlike some other languages, Python recognizes no semantic difference between single quotes and double quotes. In particular, Python does not have a character data type, so both **"c"** and **'c'** represent a string with a single character.

The demo program uses the try-except mechanism (that is, a **try** statement followed by an **except** statement). Using try-except is particularly useful when you are writing new code, but the downside is additional time and lines of code. The demo programs in the remainder of this e-book do not use try-except in order to save space.

Resources

The more or less official Python style guide is PEP 0008 (Python Enhancements Proposal #8). See https://www.python.org/dev/peps/pep-0008/.

Many Python programmers use the Google Python Style Guide. See https://google.github.io/styleguide/pyguide.html.

For additional details about the Python **try** and **except** statements and error handling, see https://docs.python.org/2/tutorial/errors.html.

For a discussion of the pros and cons of using a shebang in Linux environments, see https://en.wikipedia.org/wiki/Shebang_(Unix).

1.4 Quick reference program

The program in Code Listing 3 is a quick reference for many of the NumPy and SciPy functions and programming techniques that are presented in this e-book.

Code Listing 3: Syntax Demo

```python
# quick_ref.py                           # SciPy Programming Succinctly
# Python 2.7

import numpy as np                       # arrays, matrices, functions
import scipy.linalg as spla              # determinant, inverse, etc.
import scipy.special as ss               # advanced functions like gamma
import scipy.constants as sc             # math constants like e
import scipy.integrate as si             # functions for integration
import scipy.optimize as so              # functions for optimization
import itertools as it                   # permutations, combinations
import time                              # for timing

class Permutation:                       # custom class using an array
  def __init__(self, n):                 # constructor
    self.n = n
    self.data = np.arange(n)             # [0, 1, 2, . . (n-1)]

  def as_string(self):                   # instance method
    s = "# "
    for i in xrange(self.n):             # traverse an array
      s += str(self.data[i]) + " "
    return s + "#"

  @staticmethod
  def my_fact(n):                        # static method
    result = 1                           # iterative rather than recursive
    for i in xrange(1, n+1):             # recursion supported in Python
      result *= i                        # but usually not a good idea
    return result

# -----------------------------------
def show_matrix(m, decimals):            # standalone function
  (rows, cols) = np.shape(m)             # matrix dimensions as tuple
  for i in rows:                         # traverse a matrix
    for j in cols:
      print "%." + str(dec) % m[i,j]
    print ""
# -----------------------------------

print "\nBegin quick examples \n"

arr_a = np.array([3.0, 2.0, 0.0, 1.0])   # create array of float64
arr_b = np.zeros(4, dtype=np.int32)      # create int array [0, 0, 0, 0]
b = 1.0 in arr_a                         # search array using "in": True
result = np.where(arr_a == 1.0)          # result is (array([3]),)
arr_s = np.sort(arr_a, kind="quicksort") # sort array: [0.0, 1.0, 2.0, 3.0]
```

```python
arr_r = arr_s[::-1]                        # reverse: [3.0, 2.0, 1.0, 0.0]

np.random.seed(0)                          # set seed for reproducibility
np.random.shuffle(arr_r)                   # randomize content order

arr_ref = arr_a                            # copy array by reference
arr_d = np.copy(arr_a)                     # copy array by value
arr_v = arr_a.view()                       # copy by view reference
arr_e = arr_a + arr_b                      # add arrays

m_a = np.matrix([[1.0, 2.0], [3.0, 4.0]]) # matrix-style 2x2 matrix
m_b = np.array([[8, 7], [6, 5]])           # ndarray-style 2x2 matrix
m_c = np.zeros((2,2), dtype=np.int32)      # ndarray 2x2 matrix all 0s
try:                                       # try-except
  m = np.loadtxt("foo.txt")                # matrix from file
except Exception:
  print "Unable to open file"

m_e = m_a.transpose()                      # matrix transposition
d = spla.det(m_a)                          # matrix determinant
m_i = np.linalg.inv(m_a)                   # matrix inverse

m_idty = np.eye(2)                         # identity 2x2 matrix
m_m = np.dot(m_a, m_i)                     # matrix multiplication
b = np.allclose(m_m, m_idty, 1.0e-5)       # matrix equality by value

m_x = m_a + np.array([5.0, 8.0])           # broadcasting

p_it = it.permutations(xrange(3))          # permutations iterator
start_t = time.clock()                     # timing
for p in p_it:
  print p
end_t = time.clock()
elap = end_t - start_t
print "elapsed = " + str(elap) + "secs"    # string concatenation

pc = Permutation(3)                        # instantiate a custom class
print pc.as_string()                       # instance method call
nf = Permutation.my_fact(3)                # static method call

arr = np.array([1.0, 3.0, 5.0, 7.0])       # a sorted array
ii = np.searchsorted(arr, 2.0)             # binary search
if ii < len(arr_s) and arr_s[ii] == 2.0:   # somewhat tricky return
  print "2.0 found"

(perm, low, upp) = spla.lu(m_a)            # matrix LU decomposition
med = np.median(arr_a)                     # statistics function
rv = np.random.normal(0.0, 1.0)            # random variable generation
g = ss.gamma(3.5)                          # advanced math function

print "\nEnd quick reference \n"
```
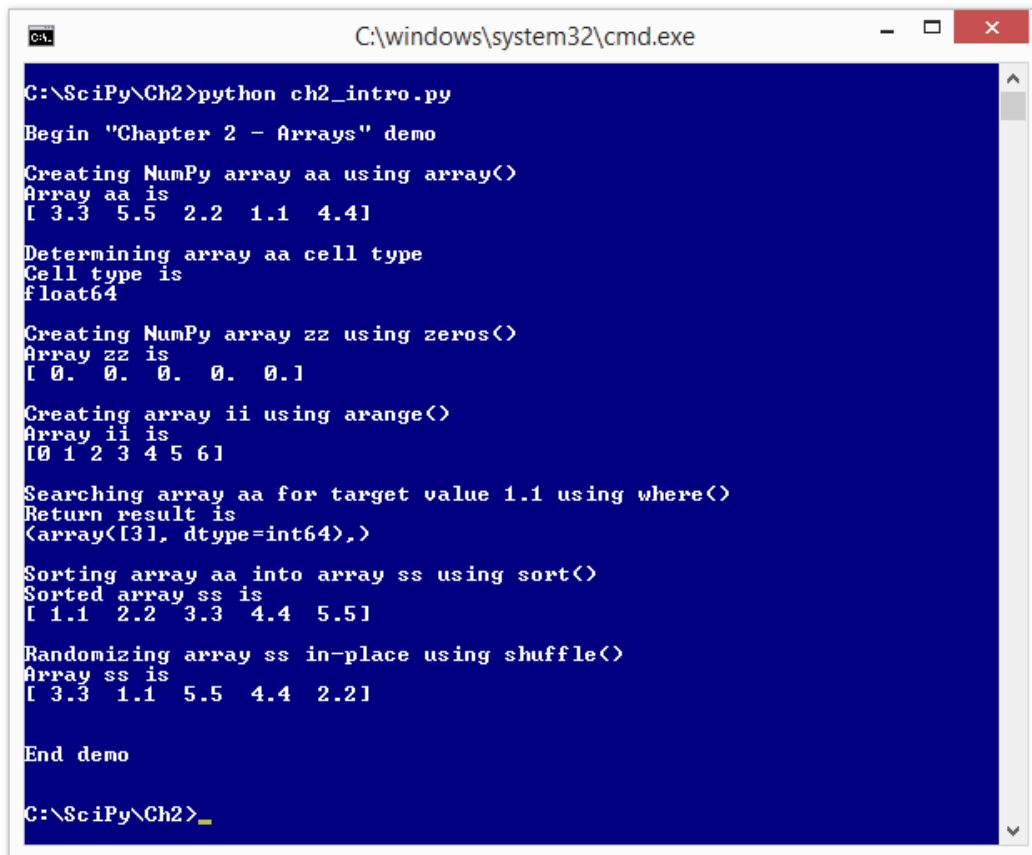
Chapter 2 Arrays

Many of my colleagues, when they first started using Python, were surprised to learn that the language does not have a built-in array data structure. The Python list data structure is versatile enough to handle many programming scenarios, but for scientific and numeric programming, arrays and matrices are needed. The most fundamental object in the SciPy and NumPy libraries is the array data structure. The following screenshot shows you where this chapter is headed.

Figure 21: NumPy Array Demo

In section 2.1, you'll learn the most common ways to create and initialize NumPy arrays, and learn about the various numeric data types supported by NumPy.

In section 2.2, you'll learn how to search an array for a target value using the `where()` function, using the `in` keyword, and by using a program-defined function.

In section 2.3, you'll learn how to sort a NumPy array using the three different built-in sorting algorithms (quicksort, merge sort, and heap sort). You'll also learn about the NumPy array reference mechanism.

In section 2.4, you'll learn how to randomize an array using the NumPy **shuffle()** function and how to randomize an array using a program-defined function and the Fisher-Yates algorithm.

2.1 Array initialization

The most fundamental object in the NumPy library is the array data structure. A NumPy array is similar to arrays in other programming languages. Arrays have a fixed size and each cell must hold the same type. The NumPy library has several functions that allow you to create an array.

Take a look at the demo program in Code Listing 4. After two preliminary **print** statements, program execution begins by creating an array using hard-coded numeric values:

```
arr = np.array([1., 3., 5., 7., 9.])
dt = np.dtype(arr[0])
print "Cell element type is " + str(dt.name) # displays 'float64'
```

The NumPy **array()** function accepts a Python list (as indicated by the square brackets) and returns an array containing the list values. Notice the decimal points. These tell the interpreter to cast the cell values as **float64**, the default floating-point data type for arrays. Without the decimals, the interpreter would cast the values to **int32**, the default integer type for arrays.

Code Listing 4: Initializing Numeric Arrays

```python
# arrays.py
# Python 2.7

import numpy as np

def my_print(arr, cols, dec):
  n = len(arr) # print array using indexing
  fmt = "%." + str(dec) + "f" # like %.4f
  for i in xrange(n):  # alt: for x in arr
    if i > 0 and i % cols == 0:
      print ""
    print fmt % arr[i],
  print "\n"

# =====

print "\nBegin array demo \n"

print "Creating array arr using np.array() and list with hard-coded values "
arr = np.array([1., 3., 5., 7., 9.]) # float64
dt = np.dtype(arr[0])
print "Cell element type is " + str(dt.name)
print ""

print "Printing array arr using built-in print() "
print arr
print ""
```

```python
print "Creating int array arr using np.arange(9) "
arr = np.arange(9) # [0, 1, . . 8] # int32
print "Printing array arr using built-in print() "
print arr
print ""

cols = 4; dec = 0
print "Printing array arr using my_print() with cols=" + str(cols),
print "and dec=" + str(dec)
my_print(arr, cols, dec)

print "Creating array arr using np.zeros(5) "
arr = np.zeros(5)
print "Printing array arr using built-in print() "
print arr
print ""

print "Creating array arr using  np.linspace(2., 5., 6)"
arr = np.linspace(2., 5., 6) # 6 values from [2.0 to 5.0] inc.
print "Printing array arr using built-in print() "
print arr
print ""

print "\nEnd demo \n"
```

```
C:\SciPy\Ch2> python arrays.py

Begin array demo

Creating array arr using np.array() and list with hard-coded values
Cell element type is float64

Printing array arr using built-in print()
[ 1.  3.  5.  7.  9.]

Creating int array arr using np.arange(9)
Printing array arr using built-in print()
[0 1 2 3 4 5 6 7 8]

Printing array arr using my_print() with cols=4 and dec=0
0 1 2 3
4 5 6 7
8

Creating array arr using np.zeros(5)
Printing array arr using built-in print()
[ 0.  0.  0.  0.  0.]

Creating array arr using  np.linspace(2., 5., 6)
```

```
Printing array arr using built-in print()
[ 2.   2.6  3.2  3.8  4.4  5. ]

End demo
```

If you are creating an array and neither **float64** nor **int32** is appropriate, you can make the data type explicit. For example:

arr = np.array([2.0, 4.0, 6.0], dtype=np.float16)

NumPy has four floating-point data types: **float_**, **float16**, **float32**, and **float64**. The default floating-point type is **float64**: a signed value with an 11-bit exponent and a 52-bit mantissa. NumPy also supports complex numbers.

NumPy has 11 integer data types, including **int32**, **int64**, and **uint64**. The default integer data type is **int32** (that is, a signed 32-bit integer with possible values between -2,147,483,648 and +2,147,483,647).

You can also create arrays with string values and with Boolean values. For example:

arr_b = np.array([True, False, True])
arr_s = np.array(["ant", "bat", "cow"])

After creating the array, the demo displays the array values using the built-in **print** statement:

print "Printing array arr using built-in print() "
print arr

The Python 2.7 **print** statement is simple and effective for displaying NumPy arrays in most situations. If you need to customize the output format, you can use the NumPy **set_printoptions()** function or write a program-defined display function.

Next, the demo program creates and initializes an array using the NumPy **arange()** function:

arr = np.arange(9)
print "Printing array arr using built-in print() "
print arr   # displays [0 1 2 3 4 5 6 7 8]

A call to **arange(n)** returns an **int32** array with sequential values 0, 1, 2,… (n-1). Note that the NumPy **arange()** function (the name stands for *array-range*, and is not a misspelling of the word *arrange*) is quite different from the Python **range()** function, which returns a list of integer values, and the Python **xrange()** function, which returns an iterator object that can be used to traverse a list or an array.

Next, the demo program displays the array generated by the **arange()** function, using a program-defined function named **my_print()**:

```
cols = 4; dec = 0
print "Printing array arr using my_print() with cols=" + str(cols),
print "and dec=" + str(dec)
my_print(arr, cols, dec)
```

The custom function displays an array in a specified number of columns (4 here), using a specified number of decimal places (0 here because the values are integers).

If you are new to Python, you might be puzzled by the trailing comma character after the first **print** statement. This syntax is used to print without a newline character and is similar to the C# **Console.Write()** method (as opposed to the **WriteLine()** method) or the Java **System.out.print()** method (as opposed to the **println()** method).

Program-defined function **my_print()** is implemented as:

```
def my_print(arr, cols, dec):
  n = len(arr)
  fmt = "%." + str(dec) + "f"  # like %.4f
  for i in xrange(n):
    if i > 0 and i % cols == 0:
      print ""
    print fmt % arr[i],
  print "\n"
```

The function first finds the number of cells in the array using the Python **len()** function. An alternative is to use the more efficient NumPy **size** property:

```
n = arr.size
```

Note that **size** has no parentheses after it because it's a property, not a function. The **my_print()** function iterates through the array using traditional array indexing:

```
for i in xrange(n):
```

Using this technique, a cell value in array **arr** is accessed as **arr[i]**. An alternative is to iterate over the array like so:

```
for x in arr
```

Here, **x** is a cell value. This technique is similar to using a "for-each" loop in other languages such as C#. In most situations, I prefer using array indexing to "for-eaching" but most of my colleagues prefer the "for x in arr" syntax.

Next, the demo program creates an array using the NumPy **zeros()** function:

```
arr = np.zeros(5)
print "Printing array arr using built-in print() "
print arr
```

Based on my experience, using the **zeros()** function is perhaps the most common way to create a NumPy array. As the name suggests, a call to **zeros(n)** creates an array with **n** cells and initializes each cell to a 0.0 value. The default element value is **float64**, so if you want an integer array initialized to 0 values, you'd have to supply the **dtype** parameter value to **zeros()** like so:

```
arr = np.zeros(5, dtype=np.int32)
```

A closely related NumPy function is **ones()**, which initializes an array to all 1.0 (or integer 1) values.

The demo concludes by creating and initializing an array using the NumPy **linspace()** function:

```
arr = np.linspace(2., 5., 6)
print "Printing array arr using built-in print() "
print arr
```

A call to **linspace(start, stop, num)** returns an array that has **num** cells with values evenly spaced between **start** and **stop**, inclusive. The demo call **np.linspace(2., 5., 6)** returns an array of six **float64** values starting with 2.0 and ending with 5.0 (2.0, 2.6, 3.2, 3.8, 4.4, and 5.0).

Note that almost all Python and NumPy functions that accept start and stop parameters return values in [start, stop), that is, between start inclusive and stop exclusive. The NumPy **linspace()** function is an exception.

There are many other NumPy functions that can create arrays, but the **array()** and **zeros()** functions can handle most programming scenarios. And you can always create specialized arrays using a program-defined function. For example, suppose you needed to create an array of the first *n* odd integers. You could define:

```
def my_odds(n):
  result = np.zeros(n, dtype=np.int32)
  v = 1
  for i in xrange(n):
    result[i] = v
    v += 2
  return result
```

And then you could create an array holding the first four odd integers with a call:

```
arr = my_odds(4)
```

A task that is closely related to creating NumPy arrays is copying an existing array. The NumPy **copy()** function can do this, and is described in detail later in this e-book.

Resources

For additional details on NumPy numeric data types, see
http://docs.scipy.org/doc/numpy-1.10.1/user/basics.types.html.

For additional information about NumPy numeric array initialization functions, see http://docs.scipy.org/doc/numpy-1.10.1/user/basics.creation.html.

For additional details on NumPy array iteration, see http://docs.scipy.org/doc/numpy-dev/reference/arrays.nditer.html#arrays-nditer

2.2 Array searching

Searching a numeric array for some target value is a common task. There are three basic ways to search a NumPy array. You can use the NumPy **where()** function, you can use the Python **in** keyword, or you can use a program-defined search function. Interestingly, there is no NumPy **index()** function like those found in several programming languages, including C# and Java.

Code Listing 5: Array Searching Demo

```python
# searching.py
# Python 2.7

import numpy as np

def my_search(a, x, eps):
  for i in xrange(len(a)):
    if (np.isclose(a[i], x, eps)):
        return i
  return -1

# =====

print "\nBegin array search demo \n"

arr = np.array([7.0, 9.0, 5.0, 1.0, 5.0, 8.0])

print "Array arr is "
print arr
print ""

target = 5.0
print "Target value is "
print target
print ""

found = target in arr
print "Search result using 'target in arr' syntax = " + str(found)
print ""

result = np.where(arr == target)
print "Search result using np.where(arr == target) is "
print result
print ""

idx = my_search(arr, target, 1.0e-6)
```

```
print "Search result using my_search() = "
print idx
print ""

print "\nEnd demo \n"
```

```
C:\SciPy\Ch2> python searching.py

Begin array search demo

Array arr is
[ 7.  9.  5.  1.  5.  8.]

Target value is
5.0

Search result using 'target in arr' syntax = True

Search result using np.where(arr == target) is
(array([2, 4], dtype=int64),)

Search result using my_search() =
2

End demo
```

The demo program begins with creating an array and a target value to search for:

```
arr = np.array([7.0, 9.0, 5.0, 1.0, 5.0, 8.0])
print "Array arr is "
print arr

target = 5.0
print "Target value is "
print target
```

Next, the demo program searches the array for the target value using the Python **in** keyword:

```
found = target in arr
print "Search result using 'target in arr' syntax = " + str(found) # 'True'
```

The return result from a call to **target in arr** is Boolean, either **True** or **False**. Nice and simple. However, using this syntax for searching an array of floating-point values is not really a good idea. The problem is that comparing two floating-point values for exact equality is very tricky. For example:

```
>>> x = 0.15 + 0.15
>>> y = 0.20 + 0.10
>>> 'yes' if x == y else 'no'
'no'
>>> # what the heck?!
```

When comparing two floating-point values for equality, you should usually not compare for exact equality; instead, you should check if the two values are very, very close to each other.

Floating-point values stored in memory are sometimes just close approximations to their true values, so comparing two floating-point values for exact equality can give unexpected results. The **target in arr** syntax doesn't give you any direct way to control how the target value is compared to the values in the array. Note that this problem with checking for exact equality doesn't exist for integer arrays (or string arrays or Boolean arrays), so the **target in arr** syntax is fine for those.

The **target in arr** syntax does work properly in the demo program, returning a correct result of **True**. Next, the demo program searches using the NumPy **where()** function:

```
result = np.where(arr == target)
print "Search result using np.where(arr == target) is "
print result
```

The somewhat tricky return result is:

```
(array([2, 4], dtype=int64,)
```

The **where()** function returns a tuple (as indicated by the parentheses) containing an array. The array holds the indices in the searched array where the target value occurs, cells 2 and 4 in this case. If you search for a target value that is not in the array, the return result is a tuple with an array with length 0:

```
(array([], dtype=int64),)
```

Therefore, if you just want to know if a target value is in an array, you can check the return value along the line of:

```
if len(result[0]) == 0:
  print "target not found in array"
else:
  print "target is in array"
```

As is the case with searching using the **in** keyword, searching an array of floating-point values using the **where()** function is not recommended because you cannot control how the cell values are compared to the target value. But using the **where()** function with integer, string, and Boolean arrays is safe and effective.

Next, the demo searches the array using a program-defined function:

```
idx = my_search(arr, target, 1.0e-6)
print "Search result using my_search() = "
print idx
```

The program-defined **my_search()** function returns -1 if the target value is not found in the array, or the cell index of the first occurrence of the target if the target is in the array. In this case the return value is 2 because the target value, 5.0, is in cells [2] and [4] of the array. The third argument, 1.0e-6, is the tolerance defining how close two floating-point values must be in order to be considered equal.

Function **my_search()** is defined:

```
def my_search(a, x, eps):
  for i in xrange(len(a)):
    if np.isclose(a[i], x, eps):
        return i
  return -1
```

The NumPy **isclose()** function compares two values and returns **True** if the values are within eps (this stands for epsilon, the Greek letter often used in mathematics to represent a small value) of each other.

Instead of using the **isclose()** function, you can compare directly using either the Python built-in **abs()** function or the NumPy **fabs()** function like so:

```
if abs(a[i] - x) < eps:
  return i
```

```
if np.fabs(a[i] - x) < eps:
  return i
```

To summarize, to search an array of floating-point values, I recommend using a program-defined function, which allows you to control the comparison of two values for equality. For integer, string, and Boolean arrays, you can use the **in** keyword, the **where()** function, or a program-defined function.

In some situations, you may want to find the location of the last occurrence of a target value in an array. Using the **where()** function with integer, string, or Boolean arrays, you could write code like:

```
result = np.where(arr == target)
```

```
if len(result[0]) == 0:
  print "-1" # not found
else:
  print result[0][len(result[0])-1] # last idx
```

To find the last occurrence of a target value in a program-defined function, you could traverse the array from back to front with a **for i in xrange(len(a)-1, -1, -1):** loop.

Resources

For additional information about the NumPy **where()** function, see
http://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.where.html.

For technical details about how NumPy stores arrays in memory, see
http://docs.scipy.org/doc/numpy-1.10.0/reference/internals.html.

For a list of Python built-in functions such as the absolute value, see
https://docs.python.org/2/library/functions.html.

2.3 Array sorting

Putting values in an array in order is a common and fundamental programming task. The NumPy library has a **sort()** function that implements three different sorting algorithms: the quicksort algorithm, the mergesort algorithm, and the heapsort algorithm.

Interestingly, unlike the sorting functions in most other languages, a call to **sort(arr)** returns a sorted array, leaving the original array **arr** unchanged. The sort functions in many programming languages sort their array argument in place, and do not return a new sorted array. However, you can sort a NumPy array **arr** in place if you wish by using the call **arr.sort()**.

Code Listing 6: Array Sorting Demo

```
# sorting.py
# Python 2.7

import numpy as np
import time

def my_qsort(a):
  quick_sorter(a, 0, len(a)-1)

def quick_sorter(a, lo, hi):
  if lo < hi:
    p = partition(a, lo, hi)
    quick_sorter(a, lo, p-1)
    quick_sorter(a, p+1, hi)

def partition(a, lo, hi):
  piv = a[hi]
  i = lo
  for j in xrange(lo, hi):
    if a[j] <= piv:
      a[i], a[j] = a[j], a[i]
      i += 1
  a[i], a[hi] = a[hi], a[i]
  return i

# =====
```

```python
print "\nBegin array sorting demo \n"

arr = np.array([4.0, 3.0, 0.0, 2.0, 1.0, 9.0, 7.0, 6.0, 5.0])
print "Original array is "
print arr
print ""

s_arr = np.sort(arr, kind='quicksort')
print "Return result of sorting using np.sort(arr, 'quicksort') is "
print s_arr
print ""
print "Original array after calling np.sort() is "
print arr
print ""

print "Calling my_qsort(arr) "
start_time = time.clock() # record starting time
my_qsort(arr)
end_time = time.clock()
elapsed_time = end_time - start_time

print "Elapsed time = "
print str(elapsed_time) + " seconds"
print ""

print "Original array after calling my_qsort(arr) is "
print arr
print ""

print "\nEnd demo \n"
```

```
C:\SciPy\Ch2> python sorting.py

Begin array sorting demo

Original array is
[ 4.  3.  0.  2.  1.  9.  7.  6.  5.]

Return result of sorting using np.sort(arr, 'quicksort') is
[ 0.  1.  2.  3.  4.  5.  6.  7.  9.]

Original array after calling np.sort() is
[ 4.  3.  0.  2.  1.  9.  7.  6.  5.]

Calling my_qsort(arr)
Elapsed time =
3.6342481559e-05 seconds

Original array after calling my_qsort(arr) is
[ 0.  1.  2.  3.  4.  5.  6.  7.  9.]
```

The demo program begins by creating an array:

```
arr = np.array([4.0, 3.0, 0.0, 2.0, 1.0, 9.0, 7.0, 6.0, 5.0])
print "Original array is "
print arr
```

Next, the demo program sorts the array using the NumPy **sort()** function:

```
s_arr = np.sort(arr, kind='quicksort')
```

The **sort()** function returns a new array with values in order, leaving the original array unchanged:

```
print "Return result of sorting using np.sort(arr, 'quicksort') is "
print s_arr    # in order
print "Original array after calling np.sort() is "
print arr      # unchanged
```

It is possible to sort an array in place using either a slightly different syntax or calling pattern:

```
arr.sort(kind='quicksort')
print arr      # arr is sorted

arr = np.sort(arr, kind='quicksort')
print arr      # arr is sorted
```

The quicksort algorithm is the default, so the call to **sort()** could have been written:

```
s_arr = np.sort(arr)
```

Supplying an explicit **kind** argument to **sort()** is useful as a form of documentation, especially in situations where other people will be using your code.

The other two sorting algorithms could have been called like so:

```
s_arr = np.sort(arr, kind='mergesort')
s_arr = np.sort(arr, kind='heapsort')

arr.sort(kind='mergesort')
arr.sort(kind='heapsort')
```

By default, the **sort()** function orders array elements from low value to high value. If you want to sort an array from high value to low, you can't do so directly, but you can use Python slicing syntax to reverse after sorting (there is no explicit **reverse()** function for arrays):

```
arr = np.array([4.0, 8.0, 6.0, 5.0])
s_arr = np.sort(arr, kind='quicksort')  # s_arr = [4.0 5.0 6.0 8.0]
r_arr = s_arr[::-1]                      # r_arr = [8.0 6.0 5.0 4.0]

arr = np.array([4.0, 8.0, 6.0, 5.0])
orig = np.copy(arr)                 # make a copy of original
arr[::-1].sort(kind='quicksort')    # reverse sort arr in-place
r_arr = np.copy(arr)                # copy reversed to r_arr if you wish
arr = np.copy(orig)                 # restore arr to original if you wish
```

Note that the **sort()** function has an optional **order** parameter. However, this parameter controls the order in which fields are compared when an array has cells holding an object with multiple fields. So **order** does not control ascending versus descending sort behavior.

The demo program concludes by sorting an array using a program-defined **my_qsort()** function:

```
start_time = time.clock()
my_qsort(arr)
end_time = time.clock()

elapsed_time = end_time - start_time
print "Elapsed time = "
print str(elapsed_time) + " seconds"

print "Original array after callng my_qsort(arr) is "
print arr     # arr is sorted
```

The program-defined **my_qsort()** function sorts its array argument in place. The demo measures the approximate amount of time used by **my_qsort()** by wrapping its call with **time.clock()** function calls. Notice the demo program has an **import time** statement at the top of the source code to bring the **clock()** function into scope.

The whole point of using a library like NumPy is that you can use built-in functions like **sort()** and so you don't have to write program-defined functions. However, there are some scenarios where writing a custom version of a NumPy function is useful. In particular, you can customize the behavior of a program-defined function, usually at the expense of extra time (to write the function) and performance.

The heart of the quicksort algorithm is the **partition()** function. A detailed explanation of how quicksort and partitioning work is outside the scope of this e-book, but the behavior of any quicksort implementation depends on how the so-called pivot value is selected. The key line of code in the custom **partition()** function is:

```
piv = a[hi]
```

The pivot value is selected as the last cell value in the current sub-array being processed. Alternatives are to select the first cell value (**piv = a[lo]**), the middle cell value, or a randomly selected cell value between **a[lo]** and **a[hi]**.

Resources

For additional information about the NumPy **sort()** function, see
http://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.sort.html.

The program-defined quicksort function in this section is based on the Wikipedia article at
https://en.wikipedia.org/wiki/Quicksort.

For additional information about working with the Python **time** module, see
https://docs.python.org/2/library/time.html.

For additional information about NumPy array slicing, see
http://docs.scipy.org/doc/numpy-1.10.0/reference/arrays.indexing.html.

2.4 Array shuffling

A surprisingly common task in data science programming is shuffling an array. Shuffling is rearranging the cell values in an array into a random order. You can think of shuffling as somewhat the opposite of sorting. You can shuffle an array using the built-in NumPy **random.shuffle()** function or by writing a program-defined shuffle function.

Code Listing 7: Array Shuffling Demo

```python
# shuffling.py
# Python 2.7

import numpy as np

def my_shuffle(a, seed):
  # shuffle array a in place using Fisher-Yates algorithm
  np.random.seed(seed)
  n = len(a)
  for i in xrange(n):
    r = np.random.randint(i, n)
    a[r], a[i] = a[i], a[r]
  return

# =====

arr = np.arange(10, dtype=np.int64) # [0, 1, 2, .. 9]
orig = np.copy(arr)
print "Array arr is "
print arr
print ""

np.random.shuffle(arr)
print "Array arr after a call to np.random.shuffle(arr) is "
print arr
print ""

print "Resetting array arr"
```

```
arr = np.copy(orig)
print "Array arr is "
print arr
print ""

my_shuffle(arr, seed=0)
print "Array arr after call to my_shuffle(arr, seed=0) is "
print arr
print ""

print "\nEnd demo \n"
```

```
C:\SciPy\Ch2> python shuffling.py

Array arr is
[0 1 2 3 4 5 6 7 8 9]

Array arr after a call to np.random.shuffle(arr) is
[1 9 8 3 0 2 7 5 6 4]

Resetting array arr
Array arr is
[0 1 2 3 4 5 6 7 8 9]

Array arr after call to my_shuffle(arr, seed=0) is
[5 1 0 6 7 3 9 8 4 2]

End demo
```

The demo program begins by creating an ordered integer array with 10 values (0 through 9) using the **arange()** function, and makes a copy of that array using the **copy()** function:

```
arr = np.arange(10, dtype=np.int64)
orig = np.copy(arr)
print "Array arr is "
print arr
```

Next, the demo shuffles the contents of the array using the NumPy **random.shuffle()** function:

```
np.random.shuffle(arr)
print "Array arr after a call to np.random.shuffle(arr) is "
print arr
```

The **random.shuffle()** function reorders the contents of its array argument in place to a random order. In this example, the seed value for the underlying random number generator was not set, so if you ran the program again, you'd almost certainly get a different ordering of the array. If you want to make your program results reproducible, which is usually the case, you can explicitly set the underlying seed value like so:

```
np.random.seed(0)
np.random.shuffle(arr)
```

Here the seed was arbitrarily set to 0. Next, the demo program resets the array to its original values using the copy:

```
print "Resetting array arr"
arr = np.copy(orig)
print "Array arr is "
print arr
```

It would have been a mistake to use the assignment operator instead of the **copy()** function in an effort to make a copy of the original array. For example, suppose you had written this code:

```
arr = np.arange(10, dtype=np.int64)
orig = arr    # assignment is probably not what you intended
print "Array arr is "
print arr
```

Because array assignment works by reference rather than by value, **orig** and **arr** are essentially pointers that both point to the same array in memory. Any change made to **arr**, such as a call to **random.shuffle(arr)**, implicitly affects **orig**, too. Therefore, an attempt to reset **arr** after a call to **random.shuffle()** would have no effect.

Another important consequence of NumPy arrays being reference objects is that a function with an array parameter can modify the array in place. You can also create a reference to an array using the **view()** function, for example **arr_v = arr.view()** creates a reference copy of **arr**.

The demo program concludes by using a program-defined function **my_shuffle()** to shuffle the array:

```
my_shuffle(arr, seed=0)
print "Array arr after call to my_shuffle(arr, seed=0) is "
print arr
```

Function **my_shuffle()** is defined as:

```
def my_shuffle(a, seed):
  np.random.seed(seed)
  n = len(a)
  for i in xrange(n):
    r = np.random.randint(i, n)
    a[r], a[i] = a[i], a[r]
  return
```

Shuffling an array into a random order is surprisingly tricky and it's very easy to write faulty code. The function **my_shuffle()** uses what is called the Fisher-Yates algorithm, which is the best approach in most situations. Notice the function uses the very handy **a,b = b,a** Python idiom to swap two values. An alternative is to use the standard **tmp=a; a=b; b=tmp** idiom that's required in other programming languages.

Resources
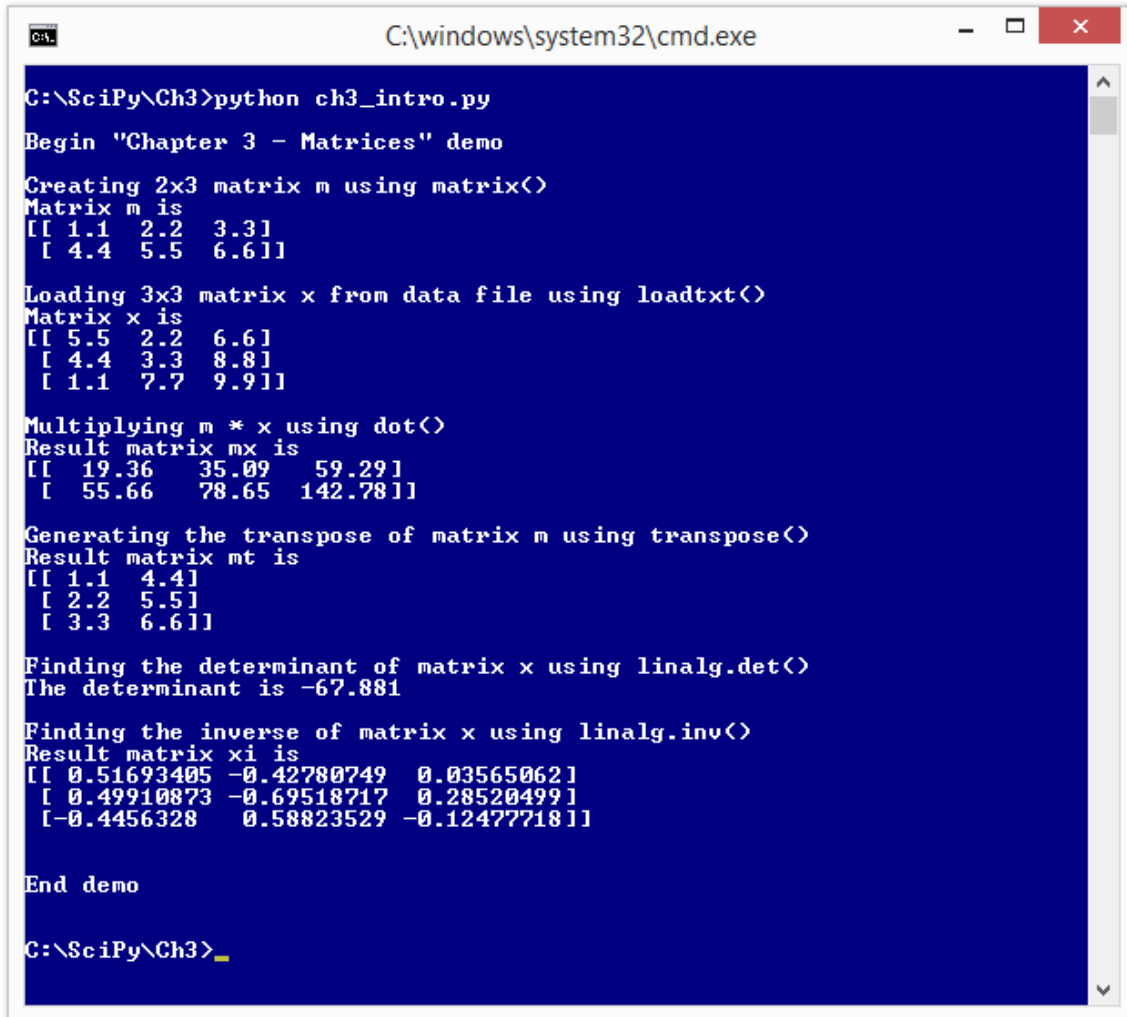
For additional details about the NumPy `random.shuffle()` function, see
http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.random.shuffle.html.

For additional information about setting the random seed, see
http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.random.seed.html.

For a really interesting explanation of the Fisher-Yates shuffle algorithm, see
https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle.

Chapter 3 Matrices

Matrices are arguably the most important data structure used in numeric and scientific programming. The following screenshot shows you where this chapter is headed.


```
C:\windows\system32\cmd.exe                                    _  □  ×

C:\SciPy\Ch3>python ch3_intro.py

Begin "Chapter 3 - Matrices" demo

Creating 2x3 matrix m using matrix()
Matrix m is
[[ 1.1  2.2  3.3]
 [ 4.4  5.5  6.6]]

Loading 3x3 matrix x from data file using loadtxt()
Matrix x is
[[ 5.5  2.2  6.6]
 [ 4.4  3.3  8.8]
 [ 1.1  7.7  9.9]]

Multiplying m * x using dot()
Result matrix mx is
[[  19.36   35.09   59.29]
 [  55.66   78.65  142.78]]

Generating the transpose of matrix m using transpose()
Result matrix mt is
[[ 1.1  4.4]
 [ 2.2  5.5]
 [ 3.3  6.6]]

Finding the determinant of matrix x using linalg.det()
The determinant is -67.881

Finding the inverse of matrix x using linalg.inv()
Result matrix xi is
[[ 0.51693405 -0.42780749  0.03565062]
 [ 0.49910873 -0.69518717  0.28520499]
 [-0.4456328   0.58823529 -0.12477718]]

End demo

C:\SciPy\Ch3>_
```

Figure 22: NumPy Matrices Demo

In section 3.1, you'll learn the most common ways to create and initialize NumPy matrices, and learn the differences between the two kinds of matrix data structures supported in NumPy.

In section 3.2, you'll learn how to perform matrix multiplication using the `dot()` function.

In section 3.3, you'll learn about the three different ways to transpose a matrix.

In section 3.4, you'll learn about the important NumPy and SciPy `linalg` modules, how to find the determinant of a matrix using the `det()` function, and what the determinant is used for.

In section 3.5, you'll learn how to create an identity matrix using the **eye()** function, find the inverse of a matrix using the **linalg.inv()** function, and correctly compare two matrices for equality using the **isclose()** function.

In section 3.6, you'll learn how to load values into a matrix from a text file using the **loadtxt()** function.

3.1 Matrix initialization

Matrices are arguably the most significant feature of the NumPy library. NumPy supports two kinds of matrices: n-dimensional array style matrices and explicit NumPy matrices. The two kinds of matrices are mostly compatible with each other. NumPy matrices can be created in many ways, but three common techniques in many data science scenarios are using the **matrix()** function, the **array()** function, and the **zeros()** function.

Code Listing 8: Matrix Initialization Demo

```python
# matrices.py
# Python 2.7

import numpy as np

def show_matrix(m, dec, wid):
  fmt = "%" + str(wid) + "." + str(dec) + "f"
  (rows, cols) = np.shape(m)
  for i in xrange(rows):
    for j in xrange(cols):
      print fmt % m[i,j],
    print ""  # end of row
  print "" # final newline

# =====

print "\nBegin matrices demo \n"

ma = np.matrix([[1.0, 2.0, 3.0], # 2x3
               [4.0, 5.0, 6.0]])

mb = np.zeros((3, 2), dtype=np.int32) # 3x2

mc = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]) # 2x3

md = np.matrix([[7.0, 8.0, 9.0]]) # 1x3

print "Matrix ma is "
print ma
print ""

print "Matrix mb is "
print mb
print ""
```

```python
print "N-dimensional array/matrix mc is "
print mc
print ""

print "Matrix ma is type " + str(type(ma))
print "Matrix mb is type " + str(type(mb))
print "Matrix mc is type " + str(type(mc))
print ""

print "Contents of matrix ma using show_matrix(ma, 3, 6) are "
show_matrix(ma, 3, 6)

msum = ma + mc
print "Result of ma + mc = "
print (msum)
print ""

md = np.matrix([[7.0, 8.0, 9.0]])
mx = ma + md
print "Matrix md is "
print md
print ""
print "Result of ma + md is "
print mx

print "\nEnd demo \n"
```

```
C:\SciPy\Ch3> python matrices.py

Begin matrices demo

Matrix ma is
[[ 1.  2.  3.]
 [ 4.  5.  6.]]

Matrix mb is
[[0 0]
 [0 0]
 [0 0]]

N-dimensional array/matrix mc is
[[ 1.  2.  3.]
 [ 4.  5.  6.]]

ma is type <class 'numpy.matrixlib.defmatrix.matrix'>
mb is type <type 'numpy.ndarray'>
mc is type <type 'numpy.ndarray'>

Contents of matrix ma using show_matrix(ma, 3, 6) are
 1.000  2.000  3.000
```

```
 4.000  5.000  6.000

Result of ma + mc =
[[  2.   4.   6.]
 [  8.  10.  12.]]

Matrix md is
[[ 7.   8.   9.]]

Result of ma + md is
[[  8.  10.  12.]
 [ 11.  13.  15.]]

End demo
```

The demo program begins by creating a matrix using the NumPy **matrix()** function:

```
ma = np.matrix([[1.0, 2.0, 3.0],
                [4.0, 5.0, 6.0]])
```

There are two rows, each with three columns, so the matrix has shape 2×3. Because no **dtype** argument was specified, each cell of the matrix holds the default **float64** data type.

Next, the demo creates a 3×2 matrix using the NumPy **zeros()** function:

```
mb = np.zeros((3, 2), dtype=np.int32)
```

Notice the double sets of parentheses used here as opposed to the single set of parentheses used to create a simple array. Each cell of matrix **mb** holds a 32-bit integer. If the **dtype** argument had been omitted, each cell would have been the default **float64** data type. As you'll see shortly, matrix **mb** is actually a NumPy n-dimensional array rather than a NumPy matrix. In the vast majority of programming situations, you can use either a NumPy 2-dimensional array or a NumPy matrix. The general terms matrix and matrices can refer to either a NumPy matrix or a NumPy n-dimensional array.

Next, the demo creates two additional matrices:

```
mc = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
md = np.matrix([[7.0, 8.0, 9.0]])
```

Matrix **mc** is a 2×3 n-dimensional array with the same values as explicit matrix **ma**. Matrix **md** is a 1×3 matrix. Matrices with one row are often called row matrices. Matrices with one column are called column matrices. For example:

```
mm = np.matrix([[7.0], [8.0], [9.0]])
```

Row and column matrices are not the same as simple one-dimensional arrays. You can create a column matrix from a row matrix (or vice versa) using the **reshape()** function, for example, **mm = np.reshape(md, (3,1))**. And you can make a regular array from an **ndarray**-style matrix using the **flatten()** or **ravel()** functions, for example:

```
aa = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])  # a 2x3 ndarray matrix
arr = np.flatten(aa)  # arr is an array [1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
```

After displaying the contents of matrices **ma**, **mb**, and **mc**, the demo displays their object types:

```
print "ma is type " + str(type(ma))  # 'numpy.matrixlib.defmatrix.matrix'
print "mb is type " + str(type(mb))  # displays 'numpy.ndarray'
print "mc is type " + str(type(mc))  # displays 'numpy.ndarray'
```

To summarize, when creating a NumPy matrix, the result can be either an explicit matrix (for example, when using the **matrix()** function) or an **ndarray** (for example, when using the **zeros()** function). In most cases, you don't have to worry about what the object type is because the two forms of matrices are usually (but not always) compatible.

Next, the demo displays the contents of matrix **ma** using the program-defined **show_matrix()** function:

```
print "Contents of matrix ma using show_matrix(ma, 3, 6) are "
show_matrix(ma, 3, 6)
```

The second and third parameters for **show_matrix()** are the number of decimals to use and the width to use when displaying each cell value. In situations like this, where there are similar parameters, it's more readable to use named-parameter syntax like:

```
show_matrix(ma, dec=3, wid=6)
```

Function **show_matrix()** illustrates how to traverse a matrix:

```
def show_matrix(m, dec, wid):
  fmt = "%" + str(wid) + "." + str(dec) + "f"
  (rows, cols) = np.shape(m)
  for i in xrange(rows):
    for j in xrange(cols):
      print fmt % m[i,j],
    print ""  # end of row
  print "" # final newline
```

The dimensions of the matrix are determined using the NumPy **shape()** function, which returns a tuple with the number of rows and columns. An alternative approach is:

```
rows = len(m)
cols = len(m[0])
```

A NumPy matrix **m** is an array of arrays. So **len(m)** is the number of rows, **m[0]** is the first row, and **len(m[0])** is the number of cells in the first row, which is the same as the number of columns (assuming all rows of **m** have the same number of columns).

The nested **for** loops iterate over the cells of the matrix from left to right, then top to bottom:

```
for i in xrange(rows):
  for j in xrange(cols):
    # curr cell is m[i,j]
```

Interestingly, NumPy allows you to access a matrix cell using either `m[i,j]` syntax or `m[i][j]` syntax. The two forms are completely equivalent. In most cases the `m[i,j]` form is preferred, only because it's easier to type.

Next, the demo program illustrates matrix addition:

```
msum = ma + mc
print "Result of ma + mc = "
print msum
```

Recall that both `ma` and `mc` are 2×3 matrices with values 1.0 through 6.0:

```
[[ 1.0  2.0  3.0]
 [ 4.0  5.0  6.0]]
```

Not surprisingly, the result (where I've added 0s after the decimal points for readability) is:

```
[[ 2.0   4.0   6.0]
 [ 8.0  10.0  12.0]]
```

However, recall that `ma` is an explicit NumPy matrix but `mc` is a NumPy **ndarray**. The point is that the two different types of matrices could be added together without any problems.

Next, the demo shows an unusual feature of NumPy called broadcasting:

```
md = np.matrix([[7.0, 8.0, 9.0]])
mx = ma + md
print "Result of ma + md is "
print mx
```

Matrix `ma` is 2×3. Matrix `md` is 1×3. In just about any other programming language that I'm aware of, an attempt to add these two matrices would generate some kind of error because these matrices have different shapes. However, NumPy allows the addition and returns:

```
[[  8.0  100.  12.0]
 [ 11.0  13.0  15.0]]
```

NumPy essentially expands the 1×3 `md` matrix to a 2×3 matrix, duplicating values, so that it has the same shape as `ma`, and then corresponding cells can be added. Some of my colleagues think NumPy broadcasting is a wonderful, useful feature. Others feel that broadcasting is a dubious feature that encourages sloppy coding and can easily lead to program bugs.

Resources

For a discussion of the differences between NumPy matrices and arrays, see
http://www.scipy.org/scipylib/faq.html#what-is-the-difference-between-matrices-and-arrays.

For details about creating matrices using the NumPy **matrix()** function, see
http://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.matrix.html.

For details about creating **ndarray** style matrices using the NumPy **array()** function, see
http://docs.scipy.org/doc/numpy-1.10.1/user/basics.creation.html.

3.2 Matrix multiplication

Performing matrix multiplication is a very common task in many numeric programming
scenarios. The NumPy **dot()** function performs matrix multiplication.

The demo program in Code Listing 9 illustrates matrix multiplication using NumPy. The program
then defines a custom function named **my_mult()**, which performs matrix multiplication using
nested loops. Program execution begins with a preliminary **print** statement and then the demo
creates a 2x3 matrix A, and a 3x2 matrix B using the NumPy **matrix()** function:

```
A = np.matrix([[1.0, 2.0, 3.0],
               [4.0, 5.0, 6.0]])

B = np.matrix([[7.0, 8.0],
               [9.0, 10.0],
               [11.0, 12.0]])
```

Code Listing 9: Matrix Multiplication Demo

```python
# multiplication.py
# Python 2.7

import numpy as np

def my_mult(a, b):
  (arows, acols) = np.shape(a)
  (brows, bcols) = np.shape(b)
  result = np.zeros((arows, bcols))
  for i in xrange(arows):
    for j in xrange(bcols):
      for k in xrange(acols):
        result[i,j] = result[i,j] + a[i,k] * b[k,j]
  return result

# =====

print "\nBegin matrix multiplication demo \n"

A = np.matrix([[1.0, 2.0, 3.0],
               [4.0, 5.0, 6.0]])

B = np.matrix([[7.0, 8.0],
               [9.0, 10.0],
               [11.0, 12.0]])
```

```python
C = np.dot(A, B) # NumPy matrix multiplication

D = my_mult(A, B) # slower Python

print "Matrix A = "
print A
print ""

print "Matrix B = "
print B
print ""

print "Result of dot(A,B) = "
print C
print ""

print "Result of my_mult(A,B) = "
print D
print ""

print "End demo \n"
```

```
C:\SciPy\Ch3> python multiplication.py

Begin matrix multiplication demo

Matrix A =
[[ 1.  2.  3.]
 [ 4.  5.  6.]]

Matrix B =
[[  7.   8.]
 [  9.  10.]
 [ 11.  12.]]

Result of dot(A,B) =
[[  58.   64.]
 [ 139.  154.]]

Result of my_mult(A,B) =
[[  58.   64.]
 [ 139.  154.]]

End demo
```

After creating the two matrices and displaying their values, we compute their product using the NumPy **dot()** function and then again using the program-defined **my_mult()** function like so:

```python
C = np.dot(A, B)
D = my_mult(A, B)
```

NumPy matrix objects can also be multiplied using the ***** operator, for example C = A * B, but **ndarray** objects must use the **dot()** function. In other words, the **dot()** function works for both types and so is preferable in most situations.

The demo concludes by displaying both results to visually verify they're the same:

```
Result of A dot B =
[[  58.   64.]
 [ 139.  154.]]


Result of my_mult(A,B) =
[[  58.   64.]
 [ 139.  154.]]
```

Matrix multiplication is perhaps best explained by example. Matrix A has shape 2×3 and matrix B has shape 3×2. The shape of their product is 2×2:

(2 x 3) * (3 x 2) = (2 x 2)

You can imagine that the two innermost dimensions, 3 and 3 here, cancel each other out, leaving the two outermost dimensions. For example, a 5×4 matrix times a 4×7 matrix will have shape 5×7. If the two innermost dimensions are not equal, NumPy will generate a "shapes not aligned" error.

The result value at cell [x,y] is the product of the values in row x of the first matrix and column y of the second matrix. So for the demo, the result at cell [0,1] uses row 0 of matrix A = [1, 2, 3] and column 1 of matrix B = [8, 10, 12], giving (1 * 8) + (2 * 10) + (3 * 12) = 64.

The implementation of program-defined function **my_mult(a, b)** begins by determining the number of rows and columns in each of the two matrix parameters by using the NumPy **shape()** function:

```
def my_mult(a, b):
  (arows, acols) = np.shape(a)
  (brows, bcols) = np.shape(b)
. . .
```

The **shape()** function returns a tuple holding the number of rows and the number of columns in a matrix. You could perform an error check here to verify that the two matrices are conformable, for example, **if acols != brows: print "Error!"**.

Once the sizes of the two input matrices are known, a result matrix with the correct shape can be initialized using the NumPy **zeros()** function:

```
result = np.zeros((arows, bcols))
```

Notice the use of double parentheses, which forces the **zeros()** function to return a matrix rather than an array. Function **my_mult()** then iterates over each row and each column, accumulating and storing the sum of products into each cell of the result matrix:

```
for i in xrange(arows):
  for j in xrange(bcols):
    for k in xrange(acols):
      result[i,j] = result[i,j] + a[i,k] * b[k,j]
return result
```

Notice that the program-defined matrix multiplication function is quite simple but does involve triple-nested **for** loops. For small matrices, the difference in performance between a program-defined method and the NumPy **dot()** function probably isn't significant in most scenarios. But for large matrices, the slower performance of a program-defined method would likely be noticeable and annoying.

The **dot()** function can be applied to NumPy one-dimensional arrays as well as matrices. For example:

```
>>> import numpy as np
>>> arr1 = np.array([1, 3, 5])
>>> arr2 = np.array([6, 4, 2])
>>> arr3 = np.dot(arr1, arr2)
>>> print arr3
28
```

In this example, the result is calculated as (1 * 6) + (3 * 4) + (5 * 2) = 6 + 12 + 10 = 28. In math terminology, this is called the dot product (hence the name of the NumPy function), the scalar product, or the inner product.

NumPy has a dedicated **inner()** function that works just with arrays. For example:

```
>>> arr4 = np.inner(arr1, arr2)
>>> print arr4
28
```

One possible program-defined implementation of an array dot product function is:

```
def my_dotprod(a1, a2):
  result = 0
  for i in xrange(len(a1)):
    result = result + a1[i] * a2[i]
  return result
```

The **dot()** function can also be applied to arrays with three or more dimensions, but this is a relatively uncommon scenario.

Resources

For additional details on the NumPy **dot()** function, see
http://docs.scipy.org/doc/numpy/reference/generated/numpy.dot.html.

For a table that lists the approximately 60 NumPy matrix functions, see
http://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.matrix.html.

For information on the NumPy **ndarray** data type, which includes matrices, see
http://docs.scipy.org/doc/numpy-1.10.1/reference/arrays.ndarray.html.

3.3 Matrix transposition

A simple but common matrix operation is transposing rows and columns. The NumPy library supports three built-in ways to transpose a matrix **m**: the **m.transpose()** function, the **np.transpose(m)** function, and the **m.T** property.

Code Listing 10: Matrix Transposition Demo

```python
# transposition.py
# Python 2.7

import numpy as np

def my_transpose(m):
  (rows, cols) = np.shape(m)
  result = np.zeros((rows, cols))
  for i in xrange(rows):
    for j in xrange(cols):
      result[j,i] = m[i,j]
  return result

# =====

print "\nBegin matrix transposition demo \n"

m = np.matrix([[1., 2., 3.],
               [4., 5., 6.],
               [7., 8., 9.]])

print "Matrix m = "
print m
print ""

mt = m.transpose()
print "Transpose from m.transpose() function = "
print mt
print ""

mt = np.transpose(m)
print "Transpose from np.transpose(m) function = "
print mt
print ""

mt = m.T
print "Transpose from m.T property  = "
```

```
print mt
print ""

mt = my_transpose(m)
print "Transpose from my_transpose() function = "
print mt
print ""

print "\nEnd demo \n"
```

```
C:\SciPy\Ch3> python transposition.py

Begin matrix transposition demo

Matrix m =
[[ 1.  2.  3.]
 [ 4.  5.  6.]
 [ 7.  8.  9.]]

Transpose from m.transpose() function =
[[ 1.  4.  7.]
 [ 2.  5.  8.]
 [ 3.  6.  9.]]

Transpose from np.transpose(m) function =
[[ 1.  4.  7.]
 [ 2.  5.  8.]
 [ 3.  6.  9.]]

Transpose from m.T property  =
[[ 1.  4.  7.]
 [ 2.  5.  8.]
 [ 3.  6.  9.]]

Transpose from my_transpose() function =
[[ 1.  4.  7.]
 [ 2.  5.  8.]
 [ 3.  6.  9.]]

End demo
```

The demo program begins by creating and displaying a simple 3×3 **float64** matrix:

```
m = np.matrix([[1., 2., 3.],
               [4., 5., 6.],
               [7., 8., 9.]])

print "Matrix m = "
print m
```

Here, matrix **m** is called a square matrix because it has the same number of rows and columns. Matrix transposition works with either square or non-square matrices.

Next, the demo program creates a transposition of the matrix **m** using three different NumPy built-in techniques:

```
mt = m.transpose()
print "Transpose from m.transpose() function = "
print mt

mt = np.transpose(m)
print "Transpose from np.transpose(m) function = "
print mt

mt = m.T
print "Transpose from m.T property  = "
print mt
```

The first function call uses the **transpose()** method of the **ndarray** class. Notice the syntax is *matrix*.**transpose()** and there are no arguments. The second function call uses the NumPy function that accepts a matrix as its argument. The third call has no parentheses, indicating it is a property. In all three function calls, the original matrix **m** is not changed. If you want to change a matrix, you can use a calling pattern along the lines of **m = np.transpose(m)**.

An immediate and obvious question is: Why are there three ways to transpose a matrix? There's no good answer. One of the strengths of open source projects like NumPy and SciPy is that they are collaborative efforts. However, this strength is offset by a certain amount of redundancy in the libraries. Basically, when you're using NumPy and SciPy you can often perform a task several ways, and frequently there's no clear best way.

The demo program concludes by calling a custom transpose function named **my_transpose()**:

```
mt = my_transpose(m)
```

Function my_transpose() is defined:

```
def my_transpose(m):
  (rows, cols) = np.shape(m)
  result = np.zeros((rows, cols))
  for i in xrange(rows):
    for j in xrange(cols):
      result[j,i] = m[i,j]
  return result
```

There's no advantage to using program-defined **my_transpose()** unless you needed to customize transposition behavior in some way.

Resources

For details about the three built-in NumPy ways to transpose a matrix, see:

http://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.transpose.html
http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.ndarray.transpose.html
http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.ndarray.T.html

3.4 Matrix determinants

The determinant of a matrix is a value that indicates whether or not the matrix has an inverse (if the determinant is 0.0, then the matrix does not have an inverse). Determinants also indicate if a set of vectors are linearly dependent and how many solutions there are to a linear system of equations. Both NumPy and SciPy have a **det()** function in their **linalg** (linear algebra) submodules.

Code Listing 11: Matrix Determinant Demo

```python
# determinants.py
# Python 2.7

import numpy as np

def extract(m, col):
  # return n-1 x n-1 submatrix w/o row 0 and col
  n = len(m)
  result = np.zeros((n-1, n-1))
  for i in xrange(1, n):
    k = 0
    for j in xrange(n):
      if j != col:
        result[i-1,k] = m[i,j]
        k += 1
  return result

def my_det(m): # inefficient!
  n = len(m)
  if n == 1:
    return m[0]
  elif n == 2:
    return (m[0,0] * m[1,1]) - (m[0,1] * m[1,0])
  else:
    sum = 0.0
    for k in xrange(n):
      sign = -1
      if k % 2 == 0:
        sign = +1
      subm = extract(m, k)
      sum = sum + sign * m[0,k] * my_det(subm)
    return sum
```

```
# =====

print "\nBegin matrix determinant demo \n"

m = np.matrix([[1., 4., 2., 3.],
               [0., 1., 5., 4.],
               [1., 0., 1., 0.],
               [2., 3., 4., 1.]])

print "Matrix m is "
print m
print ""

d = np.linalg.det(m)
print "Determinant of m using np.linalg.det() is "
print d
print ""

d = my_det(m)
print "Determinant of m using my_det() is "
print d
print ""

print "\nEnd demo \n"
```

```
C:\SciPy\Ch3> python determinants.py

Begin matrix determinant demo

Matrix m is
[[ 1.  4.  2.  3.]
 [ 0.  1.  5.  4.]
 [ 1.  0.  1.  0.]
 [ 2.  3.  4.  1.]]

Determinant of m using np.linalg.det() is
-40.0

Determinant of m using my_det() is
-40.0

End demo
```

The demo program begins by creating and displaying a 4×4 **float64** matrix:

```
m = np.matrix([[1., 4., 2., 3.],
               [0., 1., 5., 4.],
               [1., 0., 1., 0.],
               [2., 3., 4., 1.]])

print "Matrix m = "
print m
```

Determinants only apply to square matrices (those with the same number of rows and columns). The simplest case (other than a 1×1 matrix with a single value) is a 2×2 matrix. Consider the 2×2 matrix in the lower left portion of the demo matrix:

```
1.0  2.0
3.0  4.0
```

The determinant of this matrix is (1.0 * 4.0) - (2.0 * 3.0) = 4.0 - 6.0 = -2.0. In words, to calculate the determinant of a 2×2 matrix, you take upper left times lower right, and subtract upper right times lower left.

A determinant of a square matrix always exists, but it can be zero. For example, consider this matrix:

```
3.0  2.0
6.0  4.0
```

The determinant would be (3.0 * 4.0) - (2.0 * 6.0) = (12.0 - 12.0) = 0. Matrices that have a determinant of zero do not have an inverse.

For 3×3 and larger matrices, the mathematical definition of the determinant is recursive. Suppose a 3×3 matrix is:

```
a   b   c
d   e   f
g   h   i
```

In this, **a**, **b**, **c**, etc., represent arbitrary numbers. The determinant is:

$$a * det \begin{bmatrix} e & f \\ h & i \end{bmatrix} - b * det \begin{bmatrix} d & f \\ g & i \end{bmatrix} + c * det \begin{bmatrix} d & e \\ g & h \end{bmatrix}$$

Figure 23: Definition of a 3x3 Matrix Determinant

Notice you have to extract *n* submatrices of size *n*-1 × *n*-1 by removing the first row and each of the *n* columns. Writing code from scratch to calculate the determinant of a matrix with a size larger than 3×3 is very difficult, but with NumPy and SciPy, all you have to do is call the **linalg.det()** function.

The demo program finds the determinant of the matrix it created like so:

```
d = np.linalg.det(m)
print "Determinant of m using np.linalg.det() is "
print d
```

Simple and easy. The NumPy **linalg** submodule currently has 28 functions that operate on matrices, including the **det()** function. The larger SciPy **linalg** submodule has 82 functions.

Interestingly, the SciPy **linalg** submodule contains a slightly different **det()** function. The SciPy version of **det()** has a parameter **overwrite_a** that allows the matrix to be changed during the calculation of the determinant, which improves performance. Many functions appear in both the NumPy and SciPy libraries, which is both useful and a possible source of confusion.

The demo has a program-defined function **my_det()** that calculates the determinant of a matrix. Let me emphasize that the program-defined function is very inefficient and is intended only to demonstrate advanced NumPy and SciPy programming techniques. The custom **my_det()** function shouldn't be used unless you want to demonstrate a bad way to calculate a matrix determinant.

Function **my_det()** uses the same calling signature as the NumPy **det()** function:

```
d = my_det(m)
print "Determinant of m using my_det() is "
print d
```

Function **my_det()** is recursive, meaning that it calls itself. The **my_det()** function also calls a helper function **extract()** defined as:

```
def extract(m, col):
  n = len(m)
  result = np.zeros((n-1, n-1))
  for i in xrange(1, n):
    k = 0
    for j in xrange(n):
      if j != col:
        result[i-1,k] = m[i,j]
        k += 1
  return result
```

Function **extract(m, col)** accepts an *n* x *n* matrix **m** and returns an *n-1* x *n-1* matrix from which the first row and column **col** have been removed. The key code in **my_det()** is:

```
for k in xrange(n):
  sign = -1
  if k % 2 == 0: sign = +1
    subm = extract(m, k)
    sum = sum + sign * m[0,k] * my_det(subm)
```

Each of the **n** sub-matrices is extracted and **my_det()** is called recursively. There are very few situations where recursive code is a good choice, and calculating a determinant of a matrix is not one of them. The NumPy and SciPy implementations of **det()** use a technique called matrix decomposition, which is complicated, but very efficient.

Resources

For details about the NumPy **det()** function, see
http://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.linalg.det.html.

For details about the SciPy **det()** function, see
http://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.linalg.det.html.

3.5 Matrix inversion

One of the most common and important operations in numeric programming is finding the inverse of a matrix. The NumPy and SciPy **linalg.inv()** functions perform matrix inversion.

The demo program in Code Listing 12 illustrates matrix inversion using NumPy. As usual, at the top of the code, the program brings the NumPy library into scope and provides a convenience alias: **import numpy as np**.

Because the **inv()** function is part of the NumPy **linalg** (linear algebra) submodule, an alternative would be to use a **from numpy import linalg** statement. The demo program then defines a custom function named **my_close()**, which determines if two matrices are equal in the sense that all corresponding cell values are equal or nearly equal, within some small tolerance.

Program execution begins with a preliminary **print** statement and then the demo creates a 3×3 matrix **m** using the NumPy **matrix()** function, explicitly specifying the data type:

```
m = np.matrix([[3, 0, 4],
               [2, 5, 1],
               [0, 4, 5]], dtype=np.float64)
```

Code Listing 12: Matrix Inverse Demo

```
# inversion.py
# Python 2.7

import numpy as np

def my_close(m1, m2, eps):
  (rows, cols) = np.shape(m1)
  for i in xrange(rows):
    for j in xrange(cols):
      if abs(m1[i,j] - m2[i,j]) > eps:
        return False
  return True
```

```
# =====

print "\nBegin matrix inversion demo \n"

m = np.matrix([[3, 0, 4],
               [2, 5, 1],
               [0, 4, 5]], dtype=np.float64)

print "Matrix m is"
print m
print ""

mi = np.linalg.inv(m)
print "The inverse of m is"
print mi
print ""

idty = np.eye(3)
print "The 3x3 identity matrix idty is"
print idty
print ""

print "Product of mi * m is"
mim = np.dot(mi, m)
print mim
print ""

b1 = np.allclose(mim, idty)
print "Comparing mi * m with idty using np.allclose() gives"
print str(b1)
print ""

b2 = my_close(mim, idty, 1.0e-4)
print "Comparing mi * m with idty using my_close() gives"
print str(b2)

print "\nEnd demo\n"
```

```
C:\SciPy\Ch3> python inversion.py

Begin matrix inversion demo

Matrix m is
[[ 3.  0.  4.]
 [ 2.  5.  1.]
 [ 0.  4.  5.]]

The inverse of m is
[[ 0.22105263  0.16842105 -0.21052632]
 [-0.10526316  0.15789474  0.05263158]
```

```
  [ 0.08421053 -0.12631579  0.15789474]]

The 3x3 identity matrix idty is
[[ 1.   0.   0.]
 [ 0.   1.   0.]
 [ 0.   0.   1.]]

Product of mi * m is
[[  1.00000000e+00  -1.11022302e-16   0.00000000e+00]
 [  0.00000000e+00   1.00000000e+00   0.00000000e+00]
 [  0.00000000e+00   1.11022302e-16   1.00000000e+00]]

Comparing mi * m with idty using np.allclose() gives
True

Comparing mi * m with idty using my_close() gives
True

End demo
```

Matrix **m** could have been created as an *n*-dimensional array using the **array()** function:

```
m = np.array([[3., 0., 4.],[2., 5., 1.],[0., 4., 5.]])
```

After creating the matrix **m** and displaying its values, the inverse of the matrix is computed and displayed like so:

```
mi = np.linalg.inv(m)
print "The inverse of m is"
print mi
```

If the **from numpy import linalg** statement had been used at the top of the script, the **inv()** function could have been called as **linalg.inv(m)** instead. The **inv()** function applies only to square matrices (equal number of rows and columns) that have a determinant not equal to zero. The return value is a square matrix with the same shape as the original matrix.

Matrix inversion is one of the most technically challenging algorithms in numeric processing. Believe me, you do not want to try to write your own custom matrix inversion function, unless you are willing to spend a lot of time and effort, presumably because you need to implement some specialized behavior.

Not all matrices have an inverse. If you apply the **inv()** function to such a matrix, you'll get a "singular matrix" error. Therefore, you want to check first along the lines of:

```
d = np.linalg.det(m)
if d == 0.0:
  print "Matrix does not have an inverse"
else:
  mi = np.linalg.inv(m)
```

Next, the demo creates and displays a 3x3 identity matrix:

```
idty = np.eye(3)
print "The 3x3 identity matrix idty is"
print idty
```

An identity matrix is a square matrix where the cells on the diagonal from upper left to lower right contain 1.0 values, and all the other cells contain 0.0 values.

In ordinary arithmetic, the inverse of some number *x* is 1/*x*. For example, the inverse of 3 is 1/3. Notice that any number times its inverse equals 1. The identity matrix is analogous to the number 1 in ordinary arithmetic. Any matrix times its inverse equals the identity matrix.

The demo verifies the inverse is correct by multiplying the original matrix **m** by its inverse **mi** and displaying the result, which is, in fact, the identity matrix:

```
print "Product of mi * m is"
mim = np.dot(mi, m)
print mim
```

The output is somewhat difficult to read because of the **print** statement's default formatting:

```
Product of mi * m is
[[  1.00000000e+00  -1.11022302e-16   0.00000000e+00]
 [  0.00000000e+00   1.00000000e+00   0.00000000e+00]
 [  0.00000000e+00   1.11022302e-16   1.00000000e+00]]
```

If you look closely, you'll see that that main diagonal elements are 1.0 and the other cell values are very, very close to 0.0. Visual verification that two matrices (the product of the original matrix times its inverse, and the identity matrix) are equal is fine in simple scenarios, but in many situations a programmatic approach is better. The demo compares the matrix times its inverse (**mim**) and the identity matrix in two ways:

```
b1 = np.allclose(mim, idty)
print "Comparing mi * m with idty using np.allclose() gives"
print str(b1)

b2 = my_close(mim, idty, 1.0e-4)
print "Comparing mi * m with idty using my_close() gives"
print str(b2)
```

In general, it's a bad idea to compare two matrices that hold floating-point values for exact equality because floating-point values have some storage limit and therefore are sometimes only approximations to their true values. For example:

```
>>> x = 0.17 + 0.17  # 0.34
>>> y = 0.30 + 0.04  # 0.34
>>> b = (x == y)     # 0.34 == 0.34 should be True
>>> print b
False                # oops
```

The NumPy **allclose()** function accepts two matrices and returns **True** if both matrices have the same shape and all corresponding pairs of cell values are very close to each other (within 1.0e-5 (0.00001)), and **False** otherwise. If the default 1.0e-5 tolerance isn't suitable, you can pass a different tolerance argument to the **allclose()** function. For example, the statement:

```
b1 = np.allclose(mim, idty, 1.0e-8)
```

will return **True** only if all corresponding cells in matrices **mim** and **idty** are within 1.0e-8 of each other.

The demo program defines a custom method named **my_close()** that has similar functionality to the NumPy **allclose()** function. There's no advantage to writing such a custom function unless you need to implement some sort of specialized behavior, such as having a different tolerance for different rows or columns.

Program-defined function **my_close()** is implemented as:

```
def my_close(m1, m2, eps):
  (rows, cols) = np.shape(m1)
  for i in xrange(rows):
    for j in xrange(cols):
      if abs(m1[i,j] - m2[i,j]) > eps:
        return False
  return True
```

Function **my_close()** doesn't check if its two matrix parameters have the same shape. You could do so like this:

```
(rows_m1, cols_m1) = np.shape(m1)
(rows_m2, cols_m2) = np.shape(m2)
if rows_m1 != rows_m2 or cols_m1 != cols_m2:
  return None
```

The SciPy version of **inv()** has an **overwrite_a** parameter that permits the cell values in the original matrix to be overwritten during the calculation of the inverse. For example:

```
import numpy as np
import scipy.linalg as spla
m = np.random.rand(10, 10)
d = np.linalg.det(m)
if d == 0:
  print "Matrix does not have inverse"
else:
  mi = spla.inv(m, overwrite_a=True)
```

This code creates a 10×10 matrix with random values in the range [0.0 and 1.0), and then computes the matrix's inverse, allowing the matrix values to be changed in order to improve performance. However, when I've used this approach, I've never seen the original matrix changed with this form of function call.

Resources

For additional details on the NumPy matrix **inv()** function, see
http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.inv.html.

For additional details on the NumPy **allclose()** function, see
http://docs.scipy.org/doc/numpy/reference/generated/numpy.allclose.html.

For information about the NumPy **eye()** function, see
http://docs.scipy.org/doc/numpy/reference/generated/numpy.eye.html.

For information about the SciPy version of the **inv()** function, see
http://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.inv.html.

3.6 Matrix loading from file

In many practical scenarios, you'll want to read data into a matrix from a text file. The NumPy **loadtxt()** function is very versatile and can handle most situations. It's also possible to load data into a matrix from a text file using a custom function if you need specialized behavior.

Code Listing 13: Loading Matrix Data from a Text File Demo

```python
# loadingdata.py
# Python 2.7

import numpy as np

def my_load(fn, sep):
  f = open(fn, "r")

  rows = 0; cols = 0
  for line in f:
    rows += 1
    cols = len(line.strip().split(sep))

  result = np.zeros((rows,cols)) # make matrix

  f.seek(0) # back to start of file

  i = 0 # row index
  while True:
    line = f.readline()
    if not line: break
    line = line.strip()
    tokens = line.split(',') # a list
    for j in xrange(cols):
      result[i,j] = np.float64(tokens[j])
    i += 1

  f.close()
  return result
```

```
# =====

print "\nBegin matrix load demo \n"

fn = r"C:\SciPy\Ch3\datafile.txt"

m = np.loadtxt(fn, delimiter=',')
print "Matrix loaded using np.loadtxt() = "
print m
print ""

m = my_load(fn, sep=',')
print "Matrix loaded using my_load() = "
print m
print ""

print "\nEnd demo\n"
```

```
C:\SciPy\Ch3> python loadingdata.py

Begin matrix load demo

Matrix loaded using np.loadtxt() =
[[ 1.  2.]
 [ 3.  4.]
 [ 5.  6.]
 [ 7.  8.]]

Matrix loaded using my_load() =
[[ 1.  2.]
 [ 3.  4.]
 [ 5.  6.]
 [ 7.  8.]]

End demo
```

The demo program begins by specifying the location of the source data file:

fn = r"C:\SciPy\Ch3\datafile.txt"

Here, **fn** stands for file name. The **r** qualifier stands for *raw* and tells the Python interpreter to treat backslashes as literals rather than the start of an escape sequence. File **datafile.txt** is a simple comma-delimited text file with no header:

1.0, 2.0
3.0, 4.0
5.0, 6.0
7.0, 8.0

Next, the demo creates and loads a matrix like so:

```
m = np.loadtxt(fn, delimiter=',')
print "Matrix loaded using np.loadtxt() = "
print m
```

The `delimiter` argument tells `loadtxt()` how values are separated on each line. The default value is any whitespace character (spaces, tabs, newlines), so the argument is required in this case.

In addition to the required **fname** parameter and optional **delimiter** parameter, `loadtxt()` has seven additional optional parameters. Of these, based on my experience, the three most commonly used parameters are **comments**, **skiprows**, and **usecols**. For example, suppose a data file is:

```
colA : colB : colC
1.0 : 2.0 : 3.0
4.0 : 5.0 : 6.0
$ some comment
7.0 : 8.0 : 9.0
```

The following statement means: skip the first line, treat lines with '$' or '%' as comments, and load only column 0 and 2.

```
m = np.loadtxt(fn, delimiter=':', comments=['$', '%'], skiprows=1,
  usecols=[0,2])
```

Although `loadtxt()` is quite versatile, there are many scenarios it doesn't handle. In these situations, it's easy to write a custom load function. The demo program defines such a function:

```
def my_load(fn, sep):
  f = open(fn, "r")
  rows = 0; cols = 0
  for line in f:
    rows += 1
    cols = len(line.strip().split(sep))
  result = np.zeros((rows,cols)) # make matrix
  f.seek(0) # back to start of file
  i = 0      # row index
  while True:
    line = f.readline() # read a line of data
    if not line: break  # end of file?
    line = line.strip() # remove whitespace from line
    tokens = line.split(',') # split line items into a list
    for j in xrange(cols):   # store each item in the curr row
      result[i,j] = np.float64(tokens[j])
    i += 1  # next row
  f.close()
  return result
```

The function `my_load()` performs a preliminary scan of the file to determine the number of rows and columns there are, then creates a matrix with the appropriate shape, resets the file read pointer, and does a second scan to read, parse, and store each value in the data file. There are several alternative designs you can use.
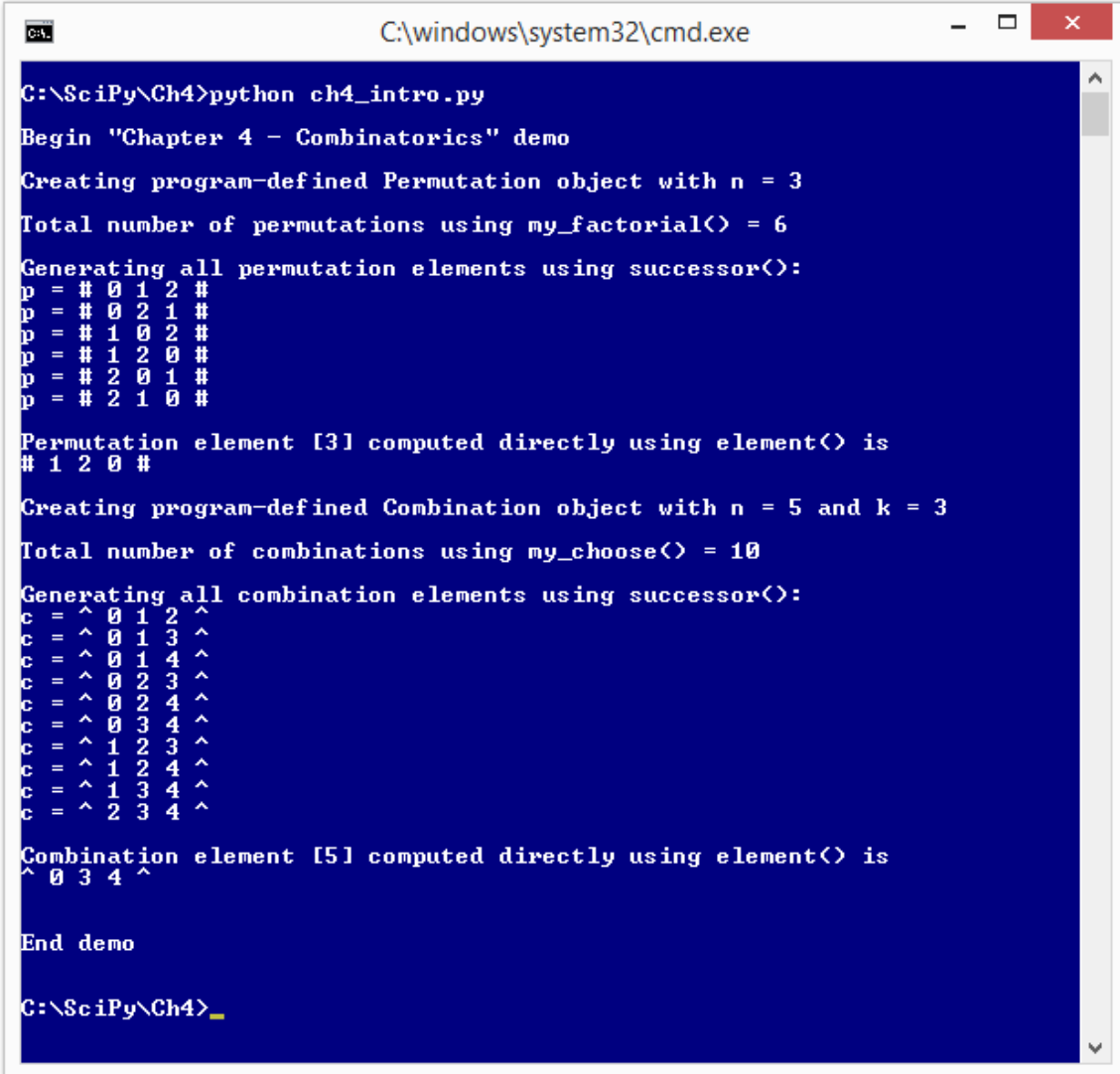
Resources

For details about the NumPy `loadtxt()` function, see
http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.loadtxt.html.

For details about NumPy function `genfromtxt()` that can handle missing values, see
http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.genfromtxt.html.

Chapter 4 Combinatorics

Combinatorics is a branch of mathematics dealing with permutations (rearrangements of items) and combinations (subsets of items). Python has limited support for combinatorics in the **itertools** and **scipy** modules, but you can create combinatorics functions using NumPy arrays and matrices. The following screenshot shows you where this chapter is headed.

```
C:\windows\system32\cmd.exe                              _  □  ×

C:\SciPy\Ch4>python ch4_intro.py

Begin "Chapter 4 - Combinatorics" demo

Creating program-defined Permutation object with n = 3

Total number of permutations using my_factorial() = 6

Generating all permutation elements using successor():
p = # 0 1 2 #
p = # 0 2 1 #
p = # 1 0 2 #
p = # 1 2 0 #
p = # 2 0 1 #
p = # 2 1 0 #

Permutation element [3] computed directly using element() is
# 1 2 0 #

Creating program-defined Combination object with n = 5 and k = 3

Total number of combinations using my_choose() = 10

Generating all combination elements using successor():
c = ^ 0 1 2 ^
c = ^ 0 1 3 ^
c = ^ 0 1 4 ^
c = ^ 0 2 3 ^
c = ^ 0 2 4 ^
c = ^ 0 3 4 ^
c = ^ 1 2 3 ^
c = ^ 1 2 4 ^
c = ^ 1 3 4 ^
c = ^ 2 3 4 ^

Combination element [5] computed directly using element() is
^ 0 3 4 ^

End demo

C:\SciPy\Ch4>_
```

Figure 24: Combinatorics Demo

In section 4.1, you'll learn how to create a program-defined **Permutation** class using NumPy, and how to write an effective factorial function.

In section 4.2, you'll learn how to write a **successor()** function that returns the next permutation element in lexicographical order.

In section 4.3, you'll learn how to create a useful **element()** function that directly generates a specified permutation element.

In section 4.4, you'll learn how to create a **Combination** class.

In section 4.5, you'll learn how to write a **Combination successor()** function.

And in section 4.6, you'll learn how to write an **element()** function for combinations.

4.1 Permutations

A mathematical permutation set is all possible orderings of *n* items. For example, if *n* = 3 and the items are the integers (0, 1, 2) then there are six possible permutation elements:

(0, 1, 2)
(0, 2, 1)
(1, 0, 2)
(1, 2, 0)
(2, 0, 1)
(2, 1, 0)

Python supports permutations in the SciPy **special** module and in the Python **itertools** module. Interestingly, NumPy has no direct support for permutations, but it is possible to implement custom permutation functions using NumPy arrays.

Code Listing 14: Permutations Demo

```python
# permutations.py
# Python 2.7

import numpy as np
import itertools as it
import scipy.special as ss

class Permutation:
  def __init__(self, n):
    self.n = n
    self.data = np.arange(n)

  @staticmethod
  def my_fact(n):
    ans = 1
    for i in xrange(1, n+1):
      ans *= i
    return ans

  def as_string(self):
    s = "# "
    for i in xrange(self.n):
      s = s + str(self.data[i]) + " "
```

```python
        s = s + "#"
        return s

# =====

print "\nBegin permutation demo \n"

n = 3
print "Setting n = " + str(n)
print ""

num_perms = ss.factorial(n)
print "Using scipy.special.factorial(n) there are ",
print str(num_perms),
print "possible permutation elements"
print ""

print "Making all permutations using itertools.permutations()"
all_perms = it.permutations(xrange(n))
p = all_perms.next()

print "The first itertools permutation is "
print p
print ""

num_perms = Permutation.my_fact(n)
print "Using my_fact(n) there are " + str(num_perms),
print "possible permutation elements"
print ""

print "Making a custom Permutation object "
p = Permutation(n)
print "The first custom permutation element is "
print p.as_string()

print "\nEnd demo \n"
```

```
C:\SciPy\Ch4> python permutations.py

Begin permutation demo

Setting n = 3

Using scipy.special.factorial(n) there are  6.0 possible permutation elements

Making all permutations using itertools.permutations()
The first itertools permutation is
(0, 1, 2)

Using my_fact(n) there are 6 possible permutation elements
```

see more please visit: https://homeofpdf.com

```
Making a custom Permutation object
The first custom permutation element is
# 0 1 2 #

End demo
```

The demo program begins by importing three modules:

```
import numpy as np
import itertools as it
import scipy.special as ss
```

The **itertools** module has the primary permutations class, but the closely associated **factorial()** function is defined in the **special** submodule of the **scipy** module. If this feels a bit awkward to you, you're not alone.

The demo program defines a custom **Permutation** class. In most cases, you will only want to define a custom implementation of a function when you need to implement some specialized behavior, or you want to avoid using a module that contains the function.

Program execution begins by setting up the number of permutation elements:

```
n = 3
print "Setting n = " + str(n)
```

Using lowercase *n* for the number of permutations is traditional, so you should use it unless you have a reason not to. Next, the demo program determines the number of possible permutations using the SciPy **factorial()** function:

```
num_perms = ss.factorial(n)
print "Using scipy.special.factorial(n) there are ",
print str(num_perms),
print "possible permutation elements"
```

The **factorial(n)** function is often written as **n!** as a shortcut. The factorial of a number is best explained by example:

```
factorial(3) = 3 * 2 * 1 = 6
factorial(5) = 5 * 4 * 3 * 2 * 1 = 120
```

The value of **factorial(0)** is usually considered a special case and defined to be 1. Next, the demo creates a Python **permutations** iterator:

```
all_perms = it.permutations(xrange(n))
```

I like to think of a Python iterator object as a little factory that can emit data when a request is made of it using an explicit or implicit call to a **next()** function. Notice the call to the **permutations()** function accepts **xrange(n)** rather than just **n**, as you might have thought.

The demo program requests and displays the first **itertools** permutation element like so:

```
p = all_perms.next()
print "The first itertools permutation is "
print p
```

Next, the demo program uses the custom functions. First, the **my_fact()** function is called:

```
num_perms = Permutation.my_fact(n)
print "Using my_fact(n) there are " + str(num_perms),
print "possible permutation elements"
```

Notice that the call to **my_fact()** is appended to **Permutation**, which is the name of its defining class. This is because the **my_fact()** function is decorated with the **@staticmethod** attribute.

Next, the demo creates an instance of the custom **Permutation** class. The **Permutation** class **__init__()** constructor method initializes an object to the first permutation element so there's no need to call a **next()** function:

```
p = Permutation(n)
print "The first custom permutation element is "
print p.as_string()
```

The custom **as_string()** function displays a **Permutation** element delimited by the % character so that the element can be easily distinguished from a tuple, a list, or another Python collection. I used % because both *permutation* and *percent* start with the letter *p*.

The custom **my_fact()** function is short and simple:

```
def my_fact(n):
  ans = 1
  for i in xrange(1, n+1):
    ans *= i
  return ans
```

The mathematical factorial function is often used in computer science classes as an example of a function that can be implemented using recursion:

```
@staticmethod
def my_fact_rec(n):
  if n == 0 or n == 1:
    return 1
  else:
    return n * Permutation.my_fact_rec(n-1)
```

Although recursion has a certain mysterious aura, in most situations (such as this one), recursion is highly inefficient and so should be avoided.

An option for any implementation of a factorial function, especially where the function will be called many times, is to create a pre-calculated lookup table with values for the first handful (say 1,000) results. The extra storage is usually a small price to pay for much-improved performance.

Resources

For details about the Python **itertools** module that contains the **permutations** iterator, see
https://docs.python.org/2/library/itertools.html.

For details about the SciPy **factorial()** function, see
http://docs.scipy.org/doc/scipy-0.16.1/reference/generated/scipy.misc.factorial.html.

For information about mathematical permutations, see
https://en.wikipedia.org/wiki/Permutation.

4.2 Permutation successor

When working with mathematical permutations, a key operation is generating the successor to a given permutation element. For example, if $n = 3$ and the items are the integers (0, 1, 2) then there are six possible permutation elements. When listed in what is called lexicographical order, the elements are:

(0, 1, 2)
(0, 2, 1)
(1, 0, 2)
(1, 2, 0)
(2, 0, 1)
(2, 1, 0)

Notice that if we removed the separating commas and interpreted each element as an ordinary integer (like 120), the elements would be in ascending order (12 < 21 < 102 < 120 < 201 < 210).

Code Listing 15: Permutation Successor Demo

```python
# perm_succ.py
# Python 2.7

import numpy as np
import itertools as it

class Permutation:
  def __init__(self, n):
    self.n = n
    self.data = np.arange(n)

  def as_string(self):
    s = "# "
    for i in xrange(self.n):
      s = s + str(self.data[i]) + " "
    s = s + "#"
    return s

  def successor(self):
    res = Permutation(self.n) # result
```

```python
      res.data = np.copy(self.data)

      left = self.n - 2
      while res.data[left] > res.data[left+1] and left >= 1:
        left -= 1

      if left == 0 and res.data[left] > res.data[left+1]:
        return None

      right = self.n - 1
      while res.data[left] > res.data[right]:
        right -= 1

      res.data[left], res.data[right] = \
                       res.data[right], res.data[left]

      i = left + 1
      j = self.n - 1
      while i < j:
        tmp = res.data[i]
        res.data[i] = res.data[j]
        res.data[j] = tmp
        i += 1; j -= 1
      return res

# =====

print "\nBegin permutation successor demo \n"

n = 3
print "Setting n = " + str(n)
print ""

perm_it = it.permutations(xrange(n))
print "Iterating all permutations using itertools permutations(): "

for p in perm_it:
  print "p = " + str(p)
print ""

p = Permutation(n)
print "Iterating all permutations using custom Permutation class: "
while p is not None:
  print "p = " + p.as_string()
  p = p.successor()

print "\nEnd demo \n"
```

```
C:\SciPy\Ch4> python perm_succ.py

Begin permutation successor demo
```

```
Setting n = 3

Iterating all permutations using itertools permutations():
p = (0, 1, 2)
p = (0, 2, 1)
p = (1, 0, 2)
p = (1, 2, 0)
p = (2, 0, 1)
p = (2, 1, 0)

Iterating all permutations using custom Permutation class:
p = # 0 1 2 #
p = # 0 2 1 #
p = # 1 0 2 #
p = # 1 2 0 #
p = # 2 0 1 #
p = # 2 1 0 #

End demo
```

The demo program begins by importing two modules:

```
import numpy as np
import itertools as it
```

Since the **itertools** module has many kinds of iterable objects, an alternative is to bring just the permutations iterator into scope:

```
from itertools import permutations
```

The demo program defines a custom **Permutation** class. In most cases, you will only want to define a custom implementation of a function when you need to implement some specialized behavior, or you want to avoid using a module that contains the function.

Program execution begins by setting up the number of permutation elements:

```
n = 3
print "Setting n = " + str(n)
```

Using lowercase *n* for the number of permutations is traditional, so you should use it unless you have a reason not to.

Next, the demo program iterates through all possible permutation elements using an implicit mechanism:

```
perm_it = permutations(xrange(n))
print "Iterating all permutations using itertools permutations(): "
for p in perm_it:
  print "p = " + str(p)
print ""
```

The **perm_it** iterator can emit all possible permutation elements. In most situations, Python iterators are designed to be called using a **for item in iterator** pattern, as shown. In other programming languages, this pattern is sometimes distinguished from a regular **for** loop by using a **foreach** keyword.

Note that the **itertools.permutations()** iterator emits tuples, indicated by the parentheses in the output, rather than a list or a NumPy array.

It is possible, but somewhat awkward, to explicitly call the permutations iterator using the **next()** function like so:

```
perm_it = it.permutations(xrange(n))
while True:
  try:
    p = perm_it.next()
    print "p = " + str(p)
  except StopIteration:
    break
print ""
```

By design, iterator objects don't have an explicit way to signal the end of iteration, such as an **end()** function or returning a special value like **None**. Instead, when an iterator object has no more items to emit and a call to **next()** is made, a **StopIteration** exception is thrown. To terminate a loop, you must catch the exception.

Next, the demo program iterates through all permutation elements for $n = 3$ using the program-defined **Permutation** class:

```
p = Permutation(n)
print "Iterating all permutations using custom Permutation class: "
while p is not None:
  print "p = " + p.as_string()
  p = p.successor()
```

The **successor()** function of the **Permutation** class uses a traditional stopping technique by returning **None** when there are no more permutation elements. The function **successor()** uses an unobvious approach to determine when the current permutation element is the last one. A straightforward approach isn't efficient. For example, if n = 5, the last element is (4 3 2 1 0) and it'd be very time-consuming to check if data[0] > data[1] > data[2] > . . > data[n-1] on each call.

The logic in the program-defined **successor()** function is rather clever. Suppose n = 5 and the current permutation element is:

0 1 4 3 2

The next element in lexicographical order after 01432, using the digits 0 through 4, is 02134. The **successor()** function first finds the indices of two items to swap, called **left** and **right**. In this case, **left** = 1 and **right** = 4. The items at those indices are swapped, giving a preliminary result of 02431. Then the items from index **right** through the end of the element are placed in order (431 in this example) giving the final result of 02134.

Resources

For details about the Python **itertools** module and the **permutations** iterator, see https://docs.python.org/2/library/itertools.html.

The **itertools.permutations** iterator uses the Python **yield** mechanism. See https://docs.python.org/2/reference/simple_stmts.html#yield.

4.3 Permutation element

When working with mathematical permutations, it's often useful to be able to generate a specific element. For example, if n = 3 and the items are the integers (0, 1, 2), then there are six permutation elements. When listed in lexicographical order, the elements are:

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

In many situations, you want to iterate through all possible permutations, but in some cases you may want to generate just a specific permutation element. For example, a function call like **pe = perm_element(4)** would store (2, 0, 1) into **pe**.

Code Listing 16: Generating a Permutation Element Directly

```python
# perm_elem.py
# Python 2.7

import numpy as np
import itertools as it
import time

class Permutation:
  def __init__(self, n):
    self.n = n
    self.data = np.arange(n)

  def as_string(self):
    s = "# "
    for i in xrange(self.n):
      s = s + str(self.data[i]) + " "
    s = s + "#"
    return s

  def element(self, idx):
    result = Permutation(self.n)

    factoradic = np.zeros(self.n)
```

```python
    for j in xrange(1, self.n + 1):
      factoradic[self.n-j] = idx % j
      idx = idx / j

    for i in xrange(self.n):
      factoradic[i] += 1

    result.data[self.n - 1] = 1

    for i in xrange(self.n - 2, -1, -1):
      result.data[i] = factoradic[i]
      for j in xrange(i + 1, self.n):
        if result.data[j] >= result.data[i]:
          result.data[j] += 1

    for i in xrange(self.n):
      result.data[i] -= 1

    return result;

# =====

def perm_element(n, idx):
  p_it = it.permutations(xrange(n))
  i = 0
  for p in p_it:
    if i == idx:
      return p
      break
    i += 1

# =====

print "\nBegin permutation element demo \n"

n = 20
print "Setting n = " + str(n) + "\n"

idx = 1000000000
print "Element " + str(idx) + " using itertools.permutations() is "
start_time = time.clock()
pe = perm_element(n, idx)
end_time = time.clock()
elapsed_time = end_time - start_time
print pe
print "Elapsed time = " + str(elapsed_time) + " seconds "
print ""

p = Permutation(n)
start_time = time.clock()
pe = p.element(idx)
end_time = time.clock()
elapsed_time = end_time - start_time
print "Element " + str(idx) + " using custom Permutation class is "
```

```
print pe.as_string()
print "Elapsed time = " + str(elapsed_time) + " seconds "
print ""

print "\nEnd demo \n"
```

```
C:\SciPy\Ch4> python perm_elem.py

Begin permutation element demo

Setting n = 20

Element 1000000000 using itertools.permutations() is
(0, 1, 2, 3, 4, 5, 6, 9, 8, 7, 15, 17, 14, 16, 19, 11, 13, 18, 10, 12)
Elapsed time = 162.92199766 seconds

Element 1000000000 using custom Permutation class is
# 0 1 2 3 4 5 6 9 8 7 15 17 14 16 19 11 13 18 10 12 #
Elapsed time = 0.000253287676799 seconds

End demo
```

The demo program begins by importing three modules:

```
import numpy as np
import itertools as it
import time
```

The demo program defines a custom **Permutation** class that has an **element()** member function and a stand-alone function **perm_element()** that is not part of a class. Both functions return a specific permutation element. Function **perm_element()** uses the built-in **permutations()** iterator from the **itertools** module. Function **element()** uses a NumPy array plus a clever algorithm that involves something called the factoradic. Program execution begins by setting up the order of a permutation, n:

```
n = 20
print "Setting n = " + str(n) + "\n"
```

The order of a permutation is the number of items in each permutation. For n = 20 there are 20! = 2,432,902,008,176,640,000 different permutation elements. Next, the demo finds the permutation element 1,000,000,000 using the program-defined **perm_element()** function:

```
print "Element " + str(idx) + " using itertools.permutations() is "
start_time = time.clock()
pe = perm_element(n, idx)
end_time = time.clock()
```

After the permutation element has been computed, the element and the elapsed time required are displayed:

```
elapsed_time = end_time - start_time
print pe
print "Elapsed time = " + str(elapsed_time) + " seconds "
```

In this example, the **perm_element()** function took over 2 and a half minutes to execute. Not very good performance.

Next, the demo computes the same permutation element using the program-defined **Permutation** class:

```
p = Permutation(n)
start_time = time.clock()
pe = p.element(idx)
end_time = time.clock()
```

Then the element and the elapsed time required are displayed using the custom class approach:

```
elapsed_time = end_time - start_time
print "Element " + str(idx) + " using custom Permutation class is "
print pe.as_string()
print "Elapsed time = " + str(elapsed_time) + " seconds "
```

The elapsed time using the custom **Permutation element()** function class was approximately 0.0003 seconds—much better performance than the 160+ seconds for the **itertools**-based function.

It really wasn't a fair fight. The **perm_element()** function works by creating an **itertools. permutations** iterator and then generating each successive permutation one at a time until the desired permutation element is reached. The function definition is:

```
p_it = it.permutations(xrange(n)) # make a permutation iterator
  i = 0              # index counter
  for p in p_it:  # request next permutation
    if i == idx:  # are we there yet?
      return p     # if so, return curr permutation tuple
      break        # and break out of loop
    i += 1          # next index
```

On the other hand, the custom **element()** function uses some very clever mathematics and an entity called the factoradic of a number to construct the requested permutation element directly.

The regular decimal representation of numbers is based on powers of 10. For example, 1047 is $(1 * 10^3) + (0 * 10^2) + (4 * 10^1) + (7 * 10^0)$. The factoradic of a number is an alternate representation based on factorials. For example, 1047 is 1232110 because it's $(1 * 6!) + (2 * 5!) + (3 * 4!) + (2 * 3!) + (1 * 2!) + (1 * 1!) + (0 * 0!)$. Using some rather remarkable mathematics, it's possible to use the factoradic of a permutation element index to compute the element directly.

Resources

For details about the Python **itertools** module, which contains the **permutations** iterator, see https://docs.python.org/2/library/itertools.html.

For information about mathematical factoradics, see https://en.wikipedia.org/wiki/Factorial_number_system.

4.4 Combinations

A mathematical combination set is a collection of all possible subsets of k items selected from n items. For example, if $n = 5$ and $k = 3$ and the items are the integers (0, 1, 2, 3, 4), then there are 10 possible combination elements:

(0, 1, 2)
(0, 1, 3)
(0, 1, 4)
(0, 2, 3)
(0, 2, 4)
(0, 3, 4)
(1, 2, 3)
(1, 2, 4)
(1, 3, 4)
(2, 3, 4)

For combinations, the order of the items does not matter. Therefore, there is no element (0, 2, 1) because it is considered the same as (0, 1, 2). Python supports combinations in the SciPy **special** module and in the Python **itertools** module. There is no direct support for combinations in SciPy, but it's possible to implement combination functions using NumPy arrays.

Code Listing 17: Combinations Demo

```python
# combinations.py
# Python 2.7

import numpy as np
import itertools as it
import scipy.special as ss

class Combination:
  # n == order, k == subset size
  def __init__(self, n, k):
    self.n = n
    self.k = k
    self.data = np.arange(self.k)

  def as_string(self):
    s = "^ "
    for i in xrange(self.k):
```

```python
        s = s + str(self.data[i]) + " "
      s = s + "^"
      return s

  @staticmethod
  def my_choose(n,k):
    if n < k: return 0
    if n == k: return 1;

    delta = k
    imax = n - k
    if k < n-k:
      delta = n-k
      imax = k

    ans = delta + 1
    for i in xrange(2, imax+1):
      ans = (ans * (delta + i)) / i
    return ans

# =====

print "\nBegin combinations demo \n"

n = 5
k = 3
print "Setting n = " + str(n) + " k = " + str(k)
print ""

num_combs = ss.comb(n, k)
print "n choose k using scipy.comb() is ",
print num_combs
print ""

print "Making all combinations using itertools.combinations() "
all_combs = it.combinations(xrange(n), k)

c = all_combs.next()
print "First itertools combination element is "
print c
print ""

num_combs = Combination.my_choose(n, k)
print "n choose k using my_choose(n, k) is ",
print num_combs
print ""

print "Making a custom Combination object "
c = Combination(n, k)
print "The first custom combination element is "
print c.as_string()

print "\nEnd demo \n"
```

```
C:\SciPy\Ch4> python combinations.py

Begin combinations demo

Setting n = 5 k = 3

n choose k using scipy.comb() is  10.0

Making all combinations using itertools.combinations()
First itertools combination element is
(0, 1, 2)

n choose k using my_choose(n, k) is  10

Making a custom Combination object
The first custom combination element is
^ 0 1 2 ^

End demo
```

The demo program begins by importing three modules:

```
import numpy as np
import itertools as it
import scipy.special as ss
```

The **itertools** module has the primary combinations class, but the closely associated **comb()** function is defined in the **special** submodule of the **scipy** module (and also in **scipy.misc**).

The demo program defines a custom **Combination** class. In most cases, you will only want to define a custom implementation of a function when you need to implement some specialized behavior, or you want to avoid using a module that contains the function.

Program execution begins by setting up the number of items **n**, and the subset size **k**:

```
n = 5
k = 3
print "Setting n = " + str(n) + " k = " + str(k)
```

Lowercase *n* and *k* are most often used with combinations, so if you use different variable names it would be a good idea to comment on which is the number of items and which is the subset size. Next, the demo program determines the number of possible combination elements using the SciPy **comb()** function:

```
num_combs = ss.comb(n, k)
print "n choose k using scipy.comb() is ",
print num_combs
```

The function that returns the number of ways to select k items from n items is almost universally called choose(*n*, *k*) so it's not clear why the SciPy code implementation is named **comb(n, k)**. The mathematical definition of choose(*n*, *k*) is *n*! / *k*! * (*n-k*)! where ! is the factorial function. For example:

choose(5, 3) = 5! / (3! * 2!) = 120 / (6 * 2) = 10

As it turns out, a useful fact is that choose(*n*, *k*) = choose(*n*, *n-k*). For example, choose(10, 7) = choose(10, 3). The choose function is easier to calculate using smaller values of the subset size.

Next, the demo creates a Python **combinations** iterator:

```
all_combs = it.combinations(xrange(n), k)
```

I like to think of a Python iterator object as a little factory that can emit data when a request is made of it using an explicit or implicit call to a **next()** function. Notice the call to the **it.combinations()** function accepts **xrange(n)** rather than just **n**. The choice of the name **all_combs** could be somewhat misleading if you're not familiar with Python iterators. The **all_combs** iterator doesn't generate all possible combination elements when it is created. It does, however, have the ability to emit all combination elements.

In addition to **xrange()**, the **it.combinations()** iterator can accept any iterable object. For example:

```
all_combs = it.combinations(np.array(["a", "b", "c"]), k)
```

Next, the demo program requests and displays the first **itertools** combination element like so:

```
c = all_combs.next()
print "The first itertools combination element is "
print c
```

Next, the demo program demonstrates the custom functions. First, the program-defined **my_choose()** function is called:

```
num_combs = Combination.my_choose(n, k)
print "n choose k using my_choose(n, k) is ",
print num_combs
```

Notice that the call to **my_choose()** is appended to **Combination**, which is the name of its defining class. This is because the **my_choose()** function is decorated with the **@staticmethod** attribute.

Next, the demo creates an instance of the custom **Combination** class. The **Combination** class **__init__()** constructor method initializes an object to the first combination element, so there's no need to call a **next()** function to get the first element:

```
print "Making a custom Combination object "
c = Combination(n, k)
print "The first custom combination element is "
print c.as_string()
```

The custom **as_string()** function displays a **Combination** element delimited by the ^ (carat) character so that the element can be easily distinguished from a tuple, a list, or another Python collection. I used ^ because both *combination* and *carat* start with the letter c.

The custom **my_choose()** function is rather subtle. It would be a weak approach to implement a choose function directly using the math definition because that would involve the calculation of three factorial functions. The factorial of a number can be very large. For example, 20! is 2,432,902,008,176,640,000 and 1000! is an almost unimaginably large number.

The **my_choose()** function uses a clever alternate definition that is best explained by example:

choose(10, 7) = choose(10, 3) = (10 * 9 * 8) / (3 * 2 * 1) = 120

Expressed in words, to calculate a choose(n, k) value, first simplify k to an equivalent smaller k if possible. Then the result is a division with k! on the bottom and $n * n\text{-}1 * n\text{-}2 * \ldots * (n\text{-}k\text{+}1)$ on the top.

Furthermore, the top and bottom parts of the division don't have to be computed fully. Instead, the product of each pair of terms in the top can be iteratively divided by a term in the bottom. For example:

choose(10, 3) = 10 * 9 / 3 = 30 * 8 / 2 = 120

The implementation of **my_choose()** is presented in Code Listing 18.

Code Listing 18: Program-Defined Choose() Function

```python
def my_choose(n, k):
  if n < k:
    return 0
  if n == k:
    return 1;

  delta = k
  imax = n - k
  if k < n-k:
    delta = n-k
    imax = k

  ans = delta + 1
  for i in xrange(2, imax+1):
    ans = (ans * (delta + i)) / i

  return ans
```

The first two statements look for early exit conditions. The statements with **delta** and **imax** simplify **k** if possible. The **for** loop performs the iterated pair-multiplication and division.

Resources

For details about the Python **itertools** module that contains the **combinations** iterator, see https://docs.python.org/2/library/itertools.html.

For details about the SciPy **factorial()** function, see http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.misc.comb.html.

4.5 Combination successor

When working with mathematical combinations, a key operation is generating the successor to a given combination element. For example, if n = 5, k = 3, and the n items are the integers (0, 1, 2, 3, 4), then there are 10 possible combination elements. When listed in lexicographical order, the elements are:

(0, 1, 2)
(0, 1, 3)
(0, 1, 4)
(0, 2, 3)
(0, 2, 4)
(0, 3, 4)
(1, 2, 3)
(1, 2, 4)
(1, 3, 4)
(2, 3, 4)

Notice that if we removed the separating commas and interpreted each element as an ordinary integer (like 124), the elements would be in ascending order (12 < 13 < 14 < 23 < . . < 234).

Code Listing 19: Combinations Successor Demo

```python
# comb_succ.py
# Python 2.7

import numpy as np
import itertools as it

class Combination:
  # n == order, k == subset size

  def __init__(self, n, k):
    self.n = n
    self.k = k
    self.data = np.arange(self.k)

  def as_string(self):
    s = "^ "
    for i in xrange(self.k):
      s = s + str(self.data[i]) + " "
    s = s + "^"
```

```python
    return s

  def successor(self):
    if self.data[0] == self.n - self.k:
      return None

    res = Combination(self.n, self.k)
    for i in xrange(self.k):
      res.data[i] = self.data[i]

    i = self.k - 1
    while i > 0 and res.data[i] == self.n - self.k + i:
      i -= 1

    res.data[i] += 1

    for j in xrange(i, self.k - 1):
      res.data[j+1] = res.data[j] + 1

    return res

# =====

print "\nBegin combination successor demo \n"

n = 5
k = 3
print "Setting n = " + str(n) + " k = " + str(k)
print ""

print "Iterating through all elements using itertools.combinations()"
comb_iter = it.combinations(xrange(n), k)
for c in comb_iter:
  print "c = " + str(c)
print ""

print "Iterating through all elements using custom Combination class"
c = Combination(n, k)
while c is not None:
  print "c = " + c.as_string()
  c = c.successor()
print ""

print "\nEnd demo \n"
```

```
C:\SciPy\Ch4> python comb_succ.py

Begin combination successor demo

Setting n = 5 k = 3

Iterating through all elements using itertools.combinations()
```

```
c = (0, 1, 2)
c = (0, 1, 3)
c = (0, 1, 4)
c = (0, 2, 3)
c = (0, 2, 4)
c = (0, 3, 4)
c = (1, 2, 3)
c = (1, 2, 4)
c = (1, 3, 4)
c = (2, 3, 4)

Iterating through all elements using custom Combination class
c = ^ 0 1 2 ^
c = ^ 0 1 3 ^
c = ^ 0 1 4 ^
c = ^ 0 2 3 ^
c = ^ 0 2 4 ^
c = ^ 0 3 4 ^
c = ^ 1 2 3 ^
c = ^ 1 2 4 ^
c = ^ 1 3 4 ^
c = ^ 2 3 4 ^

End demo
```

The demo program begins by importing two modules:

```
import numpy as np
import itertools as it
```

Since the **itertools** module has many kinds of iterable objects, an alternative is to bring just the permutations iterator into scope:

```
from itertools import combinations
```

The demo program defines a custom **Combination** class. In most cases, you will only want to define a custom implementation of a function when you need to implement some specialized behavior, or you want to avoid using a module that contains the function (such as **itertools**).

Program execution begins by setting up the number of items and the subset size:

```
n = 5
k = 3
print "Setting n = " + str(n) + " k = " + str(k)
```

It is customary to use n and k when working with mathematical combinations, so you should do so unless you have a reason to use different variable names.

Next, the demo program iterates through all possible combination elements using an implicit mechanism:

```
print "Iterating through all elements using itertools.combinations()"
comb_iter = it.combinations(xrange(n), k)
for c in comb_iter:
  print "c = " + str(c)
print ""
```

The **comb_iter** iterator can emit all possible combination elements. In most situations, Python iterators are designed to be called using a **for item in iterator** pattern, as shown. In other programming languages, this pattern is sometimes distinguished from a regular **for** loop by using a **foreach** keyword (C#) or special syntax like **for x : somearr** (Java).

Note that the **itertools.combinations()** iterator emits tuples, indicated by the parentheses in the output, rather than a list or a NumPy array.

It is possible but awkward to explicitly call the **combinations** iterator using the **next()** function like so:

```
comb_iter = it.combinations(xrange(n), k)
while True:
  try:
    c = comb_iter.next()
    print "c = " + str(c)
  except StopIteration:
    break
print ""
```

By design, iterator objects don't have an explicit way to signal the end of iteration, such as a **last()** function or returning a special value like **None**. Instead, when an iterator object has no more items to emit and a call to **next()** is made, a **StopIteration** exception is thrown. To terminate a loop, you must catch the exception. Note that you could catch a general **Exception** rather than the more specific **StopIteration**.

Next, the demo program iterates through all combination elements for **n** = 5 and **k** = 3 using the **successor()** function of the program-defined **Combination** class:

```
print "Iterating through all elements using custom Combination class"
c = Combination(n, k)
while c is not None:
  print "c = " + c.as_string()
  c = c.successor()
print ""
```

The **successor()** function of the **Combination** class uses a traditional stopping technique by returning **None** when there are no more permutation elements. The logic in the program-defined **successor()** function is rather clever. Suppose **n** = 7, **k** = 4, and the current combination element is:

^ 0 1 5 6 ^

The next element in lexicographical order after 0256, using the digits 0 through 6, is 0345. The successor algorithm first finds the index **i** of the left-most item that must change. In this case, **i** = 1, which corresponds to item 2. The item at **i** is incremented, giving a preliminary result of 0356. Then the items to the right of the new value at **i** (56 in this case) are updated so that they are all consecutive relative to the value at **i** (45 in this case), giving the final result of 0345.

Notice that it's quite easy for **successor()** to determine the last combination element because it's the only one that has a value of n-k at index 0. For example, with **n** = 5 and **k** = 3, **n-k** = 2 and the last combination element is (2 3 4). Or, if n = 20 and **k** = 8, the last combination element would be (12 13 14 . . . 19).

One potential advantage of using a program-defined **Combination** class rather than the **itertools.combinations()** iterator is that you can easily define a **predecessor()** function. For example, consider the functions in Code Listing 20:

Code Listing 20: A Combination Predecessor Function

```python
def predecessor(self):
  if self.data[self.n - self.k] == self.n - self.k:
    return None
  res = Combination(self.n, self.k)
  res.data = np.copy(self.data)
  i = self.k - 1
  while i > 0 and res.data[i] == res.data[i-1] + 1:
    i -= 1
  res.data[i] -= 1; i += 1
  while i < k:
    res.data[i] = self.n - self.k + i
    i += 1
  return res

def last(self):
  res = Combination(self.n, self.k)
  nk = self.n - self.k
  for i in xrange(self.k):
    res.data[i] = nk + i
  return res
```

Then the following statements would iterate through all combination elements in reverse order:

```python
c = Combination(n, k) # 0 1 2
c = c.last()          # 2 3 4
while c is not None:
  print "c = " + c.as_string()
  c = c.predecessor()
```

Resources

For details about the Python **itertools** module, which contains the **combinations** iterator, see https://docs.python.org/2/library/itertools.html.

The **itertools.combinations** iterator uses the Python **yield** mechanism. See https://docs.python.org/2/reference/simple_stmts.html#yield.

4.6 Combination element

When working with mathematical combinations, it's often useful to be able to generate a specific element. For example, if $n = 5$, $k = 3$, and the items are the integers (0, 1, 2, 3, 4), then there are 10 combination elements. When listed in lexicographical order, the elements are:

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

In many situations, you want to iterate through all possible combination elements, but in some cases you may want to generate just a specific combination element. For example, a function call like **ce = comb_element(5)** would store (0, 3, 4) into **ce**.

Using the built-in **itertools.combinations** iterator, the only way you can get a specific combination element is to iterate from the first element until you reach the desired element. This approach is impractical in all but the simplest scenarios. An efficient alternative is to define a custom **Combination** class and **element()** function that use NumPy arrays for data.

Code Listing 21: Generating a Combination Element Directly

```python
# comb_elem.py
# Python 2.7

import numpy as np          # to make custom Combination class
import itertools as it      # has combinations iterator
import scipy.special as ss  # has comb() aka choose() function
import time                 # to time performance

class Combination:
  def __init__(self, n, k):
    self.n = n
    self.k = k
    self.data = np.arange(k)

  def as_string(self):
    s = "^ "
    for i in xrange(self.k):
      s = s + str(self.data[i]) + " "
    s = s + "^"
    return s

  @staticmethod
  def my_choose(n,k):
    if n < k: return 0
```

```python
        if n == k: return 1;

        delta = k
        imax = n - k
        if k < n-k:
          delta = n-k
          imax = k

        ans = delta + 1
        for i in xrange(2, imax+1):
          ans = (ans * (delta + i)) / i
        return ans

    def element(self, idx):
        maxM = Combination.my_choose(self.n, self.k) - 1

        ans = np.zeros(self.k, dtype=np.int64)
        a = self.n
        b = self.k
        x = maxM - idx
        for i in xrange(self.k):
          ans[i] = self.my_largestV(a, b, x)
          x = x - Combination.my_choose(ans[i], b)
          a = ans[i]
          b -= 1

        for i in xrange(self.k):
          ans[i] = (self.n - 1) - ans[i]

        result = Combination(self.n, self.k)
        for i in xrange(self.k):
          result.data[i] = ans[i]
        return result

    def my_largestV(self, a, b, x):
        v = a - 1
        while Combination.my_choose(v, b) > x:
          v -= 1
        return v

# =====

def comb_element(n, k, idx):
    comb_it = it.combinations(xrange(n), k)
    i = 0
    for c in comb_it:
      if i == idx:
        return c
        break
      i += 1
    return None

# =====
```

```
print "\nBegin combination element demo \n"

n = 100
k = 8
print "Setting n = " + str(n) + " k = " + str(k)
ces = ss.comb(n, k)
print "There are " + str(ces) + " different combinations \n"

idx = 100000000

print "Element " + str(idx) + " using itertools.combinations() is "
start_time = time.clock()
ce = comb_element(n, k, idx)
end_time = time.clock()
elapsed_time = end_time - start_time
print ce
print "Elapsed time = " + str(elapsed_time) + " seconds "
print ""

c = Combination(n, k)
start_time = time.clock()
ce = c.element(idx)
end_time = time.clock()
elapsed_time = end_time - start_time
print "Element " + str(idx) + " using custom Combination class is "
print ce.as_string()
print "Elapsed time = " + str(elapsed_time) + " seconds "
print ""

print "\nEnd demo \n"
```

```
C:\SciPy\Ch4> python comb_elem.py

Begin combination element demo

Setting n = 100 k = 8
There are 186087894300.0 different combinations

Element 100000000 using itertools.combinations() is
(0, 1, 3, 19, 20, 44, 47, 90)
Elapsed time = 10.664860732 seconds

Element 100000000 using custom Combination class is
^ 0 1 3 19 20 44 47 90 ^
Elapsed time = 0.001009821625 seconds

End demo
```

The demo program sets up a combinatorial problem with **n** = 100 items taken **k** = 8 at a time. So the first combination element is (0, 1, 2, 3, 4, 5, 6, 7). The number of different combination elements is calculated using the **comb()** function from the **scipy.special** module and is 186,087,894,300. Note that in virtually all other programming language libraries, the function to calculate the number of different combination elements is called **choose()**.

The demo calculates combination element 100,000,000 using a stand-alone, program-defined function **comb_element()** that uses the built-in **itertools.combinations** iterator. This approach took just over 10 seconds on a more or less standard desktop PC machine.

The demo calculates the same combination element using a program-defined **Combination** class and **element()** function. This approach took just over 0.001 seconds. The point is that Python iterators are designed to iterate well, but are not well suited for other scenarios.

The program-defined function **comb_element()** is:

```
def comb_element(n, k, idx):
  comb_it = it.combinations(xrange(n), k) # make an iterator
  i = 0                     # index counter
  for c in comb_it:         # request next combination element
    if i == idx:            # are we there yet?
      return c; break       # if so, return current element and exit loop
    i += 1                  # otherwise bump counter
  return None               # should never get here
```

The function doesn't check if parameter **idx** is valid. You could do so using a statement like:

```
if idx >= ss.comb(n, k):  # error
```

The obvious problem with using an iterator is that there's no way to avoid walking through every combination element until you reach the desired element. On the other hand, the program-defined **element()** function in the **Combination** class uses a clever mathematical idea called the combinadic to generate a combination element directly.

The regular decimal representation of numbers is based on powers of 10. For example, 7203 is $(7 * 10^3) + (2 * 10^2) + (0 * 10^1) + (3 * 10^0)$. The combinadic of a number is an alternate representation based on the mathematical choose(n,k) function. For example, if $n = 7$ and $k = 4$, the number 27 is 6521 in combinadic form because 27 = choose(6,4) + choose(5,3) + choose(2,2) + choose(1,1). Using some rather remarkable mathematics, it's possible to use the combinadic of a combination element index to compute the element directly.
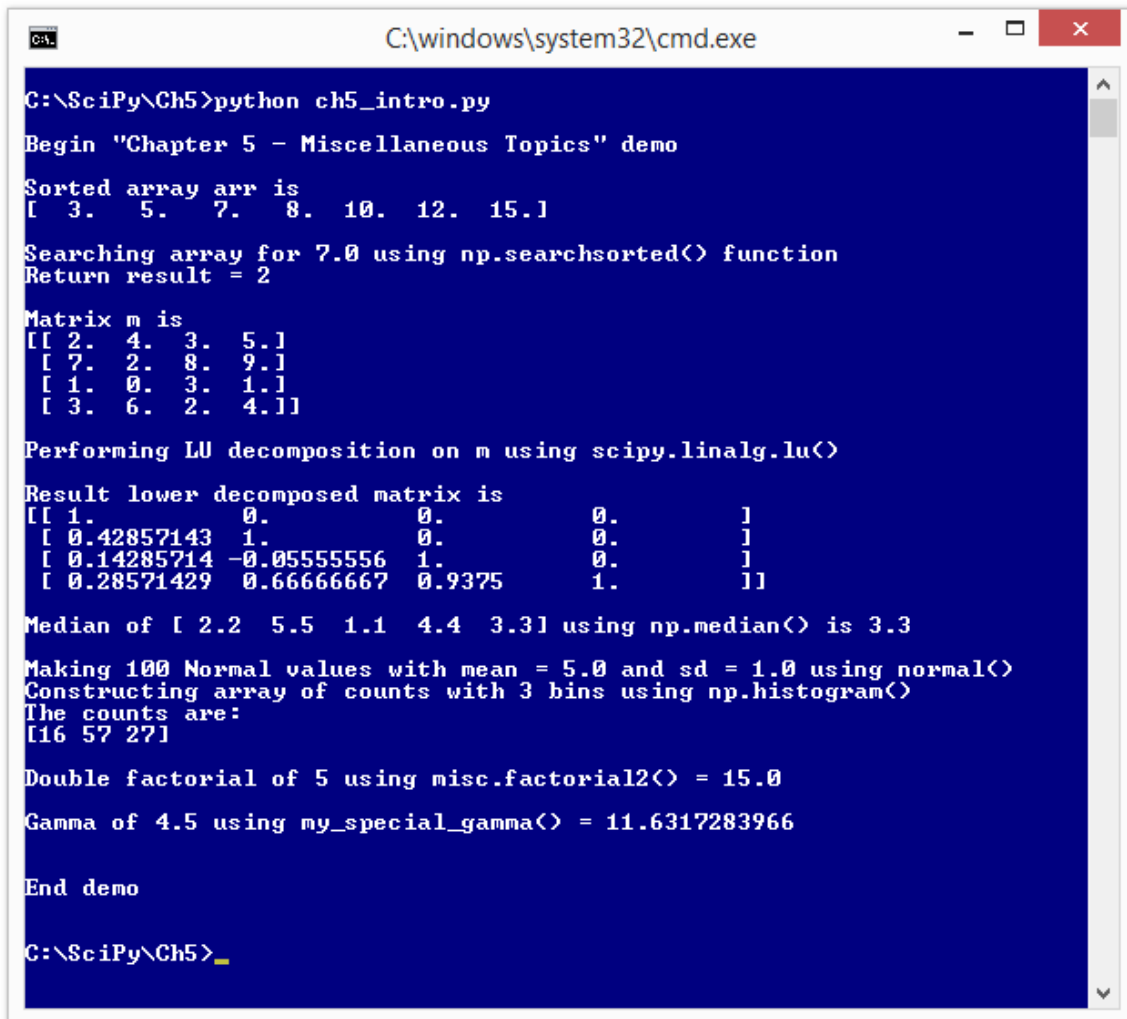
Resources

For details about the Python **itertools** module that contains the **combinations** iterator, see https://docs.python.org/2/library/itertools.html.

For information about mathematial combinadics, see https://en.wikipedia.org/wiki/Combinatorial_number_system.

Chapter 5 Miscellaneous Topics

This chapter deals with miscellaneous NumPy and SciPy functions and techniques. The goal is to present representative examples so you'll be able to search the SciPy documentation more efficiently. The following screenshot shows you where this chapter is headed.

Figure 25: Miscellaneous NumPy Functions Demo

In section 5.1, you'll learn how to use the NumPy `searchsorted()` binary search function and how to interpret its unusual return value.

In section 5.2, you'll learn how to use SciPy to perform LU decomposition on a square matrix and why decomposition is important.

In section 5.3, you'll learn about NumPy and SciPy statistics functions such as `mean()` and `std()`.

In section 5.4, you'll learn how to generate random values from a specified distribution such as the Normal or Poisson, and how to bin data using the **histogram()** function.

In section 5.5, you'll learn about SciPy miscellaneous functions such as the double factorial.

In section 5.6, you'll learn how to use special SciPy functions such as **bernoulli()** and **gamma()**.

5.1 Array binary search

To search a sorted array, you can use the NumPy **searchsorted()** function. The **searchsorted()** function is quite different from the binary search functions in other languages. Also, you must be careful when dealing with arrays that have floating-point values.

Code Listing 22: Array Binary Search Demo

```python
# binsearch.py
# Python 2.7

import numpy as np

def my_bin_search(a, t, eps):
  lo = 0
  hi = len(a)-1
  while lo <= hi:
    mid = (lo + hi) / 2
    if np.isclose(a[mid], t, eps):
      return mid
    elif a[mid] < t:
      lo = mid + 1
    else:
      hi = mid - 1
  return -1

print "\nBegin array binary search demo \n"

arr = np.array([1.0, 3.0, 4.0, 6.0, 8.0, 11.0, 13.0])
print "Array arr is "
print arr
print ""

target = 11.0
print "Target value to find is " + str(target)
print ""

print "Searching array using np.searchsorted() function "
idx = np.searchsorted(arr, target)
if idx < len(arr) and arr[idx] == target:
  print "Target found at cell " + str(idx)
else:
  print "Target not found "
```

```
print ""

print "Searching array using my_bin_search() function "
idx = my_bin_search(arr, target, 1.0e-5)
if idx == -1:
  print "Target not found "
else:
  print "Target found at cell = " + str(idx)
print ""

print "\nEnd demo \n"
```

```
C:\SciPy\Ch5> python binsearch.py

Begin array binary search demo

Array arr is
[  1.   3.   4.   6.   8.  11.  13.]

Target value to find is 11.0

Searching array using np.searchsorted() function
Target found at cell 5

Searching array using my_bin_search() function
Target found at cell = 5

End demo
```

The demo program execution begins by setting up an array to search and a target value to search for:

```
arr = np.array([1.0, 3.0, 4.0, 6.0, 8.0, 11.0, 13.0])
print "Array arr is "
print arr

target = 11.0
print "Target value to find is " + str(target)
```

If you need to search a very large array and the array is already sorted, a binary search is often the best approach because it's much faster than a simple sequential search. For small arrays (typically those with less than 100 cells), the marginally faster performance of a binary search is often unimportant, and if your array is not already sorted, the time required to sort the array usually wipes out any time saved by a binary search.

Next, the demo calls the NumPy **searchsorted()** function like so:

```
print "Searching array using np.searchsorted() function "
idx = np.searchsorted(arr, target)
if idx < len(arr) and arr[idx] == target:
  print "Target found at cell " + str(idx)
else:
  print "Target not found "
```

The binary search functions in most programming languages return a -1 if the target is not found, or return the cell index that holds the target value if the target is found. The **searchsorted()** function works a bit differently.

A call to **searchsorted(arr, x)** returns the cell index in sorted array **arr** where **x** would be inserted so that the array would remain sorted. For example, if **arr = [2.0, 5.0, 6.0, 9.0]** and **x = 3.0**, then **searchsorted(arr, x)** returns 1 because the 3.0 would be inserted at cell 1 in order to keep the array sorted. If **x = 11.0**, then **searchsorted(arr, x)** would return 4 because the 11.0 would have to be inserted beyond the end of the array.

If **x** is a value that is already in the array, then **searchsorted(arr, x)** will return the cell where the value is. Therefore, to determine if a value is in an array **arr** using the return value **idx** from **searchsorted(arr, x)**, you must first check that **idx** is less than the length of **arr** and then check to see if the value at **arr[idx]** equals the target value.

If the search array holds floating-point values, using **searchsorted()** is somewhat risky. For example, if the target value is 11.0000000000000001 (there are 15 zeros), it would not be found by the demo program, but a slightly less precise target of 11.000000000000001 (there are 14 zeros) would be found.

The lesson is that, when searching a sorted array of floating-point values using the NumPy **searchsorted()**, you don't have control over how the function determines floating-point value equality, so you may want to write a program-defined binary search function like **my_bin_search()** in the demo program:

```
def my_bin_search(a, t, eps):
  lo = 0
  hi = len(a)-1
  while lo <= hi:
    mid = (lo + hi) / 2
    if np.isclose(a[mid], t, eps):
      return mid
    elif a[mid] < t:
      lo = mid + 1
    else:
      hi = mid - 1
  return -1   # not found
```

The program-defined function **my_bin_search()** uses a standard iterative (as opposed to recursive) binary search algorithm with early check for equality, combined with an epsilon parameter to control how close two floating-point values must be for them to be evaluated as equal.

Resources

For details about the NumPy `searchsorted()` function, see
http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.searchsorted.html.

For information about the array binary search algorithm used by the demo, see
https://en.wikipedia.org/wiki/Binary_search_algorithm.

5.2 Matrix decomposition

Matrix decomposition is the process of breaking a matrix down into two smaller matrices that, when multiplied together, give the original matrix (or a slightly rearranged version of the original). An analogy in regular arithmetic is breaking down the number 21 into 3 and 7 because 3 * 7 = 21.

Matrix decomposition, also called matrix factorization, is rarely used by itself, but decomposition is the basis for efficient algorithms that find the inverse and the determinant of a matrix.

There are several kinds of matrix decomposition. The most common form is called lower-upper decomposition for reasons that will become clear shortly. The `scipy.linalg.lu()` function performs lower-upper matrix decomposition. It's sometimes useful to write a program-defined matrix decomposition function.

Code Listing 23: Matrix Decomposition Demo

```python
# decomposition.py
# Python 2.7

import numpy as np
import scipy.linalg as spla

def my_decomp(m):
  # LU decompose matrix m using Crout's algorithm
  n = len(m)
  toggle = 1 # row swapping parity
  lum = np.copy(m) # result matrix
  perm = np.arange(n) # row permutation info

  for j in xrange(n-1):
    max = abs(lum[j,j])
    piv = j

    for i in xrange(j+1, n): # find pivot row
      xij = abs(lum[i,j])
      if (xij > max):
        max = xij
        piv = i

    if (piv != j):
      for k in xrange(n): # swap rows j, piv
```

```python
          t = lum[piv,k]
          lum[piv,k] = lum[j,k]
          lum[j,k] = t

      perm[j], perm[piv] = perm[piv], perm[j]
      toggle = -toggle

    xjj = lum[j,j]
    if xjj != 0.0:
      for i in xrange(j+1, n):
        xij = lum[i,j] / xjj
        lum[i,j] = xij
        for k in xrange(j+1, n):
          lum[i,k] = lum[i,k] - (xij * lum[j,k])

  return (lum, perm, toggle)

# =====

print "\nBegin matrix decomposition demo \n"

m = np.matrix([[3., 2., 1., 3.],
               [5., 6., 4., 2.],
               [7., 9., 8., 1.],
               [4., 2., 3., 0.]])

print "Original matrix m = "
print m

print "\nDecomposing m using scipy.linalg.lu() "
(perm, low, upp) = spla.lu(m)

print "\nResult permutation matrix is "
print perm

print "\nResult lower matrix is "
print low

print "\nResult upper matrix is "
print upp

prod = np.dot(low, upp)
print "\nProduct of lower * upper is "
print prod

print "\n----------"

print "\nDecomposing m using my_decomp() "
(lum, perm, t) = my_decomp(m)

print "\nResult row swap parity (+1 / -1) = " + str(t)

print "\nResult permutation array is "
print perm
```

```
print "\nResult combined LU matrix = "
print lum

print "\nEnd demo\n"
```

```
C:\SciPy\Ch5> python decomposition.py

Begin matrix decomposition demo

Original matrix m =
[[ 3.   2.   1.   3.]
 [ 5.   6.   4.   2.]
 [ 7.   9.   8.   1.]
 [ 4.   2.   3.   0.]]


Decomposing m using scipy.linalg.lu()

Result permutation matrix is
[[ 0.   0.   1.   0.]
 [ 0.   0.   0.   1.]
 [ 1.   0.   0.   0.]
 [ 0.   1.   0.   0.]]


Result lower matrix is
[[ 1.          0.          0.          0.        ]
 [ 0.57142857  1.          0.          0.        ]
 [ 0.42857143  0.59090909  1.          0.        ]
 [ 0.71428571  0.13636364  1.          1.        ]]


Result upper matrix is
[[ 7.          9.          8.          1.        ]
 [ 0.         -3.14285714 -1.57142857 -0.57142857]
 [ 0.          0.         -1.5         2.90909091]
 [ 0.          0.          0.         -1.54545455]]


Product of lower * upper is
[[ 7.   9.   8.   1.]
 [ 4.   2.   3.   0.]
 [ 3.   2.   1.   3.]
 [ 5.   6.   4.   2.]]


----------

Decomposing m using my_decomp()

Result row swap parity (+1 / -1) = 1

Result permutation array is
```

```
[2 3 0 1]

Result combined LU matrix =
[[ 7.          9.          8.          1.        ]
 [ 0.57142857 -3.14285714 -1.57142857 -0.57142857]
 [ 0.42857143  0.59090909 -1.5         2.90909091]
 [ 0.71428571  0.13636364  1.         -1.54545455]]

End demo
```

The demo program begins by bringing the **scipy.linalg** submodule into scope:

import numpy as np
import scipy.linalg as spla

After creating the source matrix **m** and displaying its values, the matrix is decomposed using the **linalg.lu()** function like so:

print "\nDecomposing m using scipy.linalg.lu() "
(perm, low, upp) = spla.lu(m)

The return result is a tuple with three items. The first item, **perm**, will be explained shortly. The second and third items are the decomposed matrices. For the demo, return matrix **low** is:

```
[[ 1.          0.          0.          0.        ]
 [ 0.57142857  1.          0.          0.        ]
 [ 0.42857143  0.59090909  1.          0.        ]
 [ 0.71428571  0.13636364  1.          1.        ]]
```

Notice that the relevant values are in the lower part of the matrix, and there are dummy 1.0 values on the main diagonal. The return matrix **upp** is:

```
[[ 7.          9.          8.          1.        ]
 [ 0.         -3.14285714 -1.57142857 -0.57142857]
 [ 0.          0.         -1.5         2.90909091]
 [ 0.          0.          0.         -1.54545455]]
```

Here, all the relevant values are on the main diagonal and above. Next, the demo multiplies **low** and **upp** using the NumPy **dot()** function and displays the resulting matrix:

```
[[ 7.  9.  8.  1.]
 [ 4.  2.  3.  0.]
 [ 3.  2.  1.  3.]
 [ 5.  6.  4.  2.]]
```

The original matrix is:

```
[[ 3.   2.   1.   3.]
 [ 5.   6.   4.   2.]
 [ 7.   9.   8.   1.]
 [ 4.   2.   3.   0.]]
```

Notice that the product of matrices **low** and **upp** is almost the original matrix. Rows 0 and 1 have been swapped and rows 2 and 4 have been swapped. The swap information is contained in the **perm** matrix result:

```
[[ 0.   0.   1.   0.]
 [ 0.   0.   0.   1.]
 [ 1.   0.   0.   0.]
 [ 0.   1.   0.   0.]]
```

This may be interesting, but what's the point? As it turns out, the lower and upper matrices of a decomposition can be used to easily calculate the determinant of the original matrix, and can also be used to compute the inverse of the original matrix.

The determinant of a matrix is the product of the parity of row swaps times the product of the diagonal elements of the upper matrix. The inverse of a matrix can be computed using a short helper function that performs what is called elimination on the lower and upper matrices.

This is exactly how SciPy calculates the determinant and inverse of a matrix. It may seem odd to use such an indirect approach, but decomposing a matrix and then finding the determinant or the inverse is much easier and faster than finding the determinant or inverse directly.

The LU decomposition functions in many other libraries return different values than the **scipy.linalg.lu()** function. The demo program implements a custom **my_decomp()** decomposition function that returns values in a different format. The call to **my_decomp()** is:

```
print "\nDecomposing m using my_decomp() "
(lum, perm, t) = my_decomp(m)
```

The program-defined function returns a tuple of three items. The first is a combined lower-upper matrix (instead of separate lower and upper matrices). The second item is a permutation array (instead of a matrix). And the third item is a toggle parity where +1 indicates an even number of row swaps and -1 indicates an odd number of row swaps. For the demo, the combined lower-upper matrix result from **my_decomp()** is:

```
[[ 7.          9.          8.          1.         ]
 [ 0.57142857 -3.14285714 -1.57142857 -0.57142857]
 [ 0.42857143  0.59090909 -1.5          2.90909091]
 [ 0.71428571  0.13636364  1.          -1.54545455]]
```

These are the same values from **linalg.lu()** except combined into a single matrix to save space. The result **perm** array from **my_decomp()** is:

```
[2 3 0 1]
```

This contains essentially the same information as the **perm** matrix return from **linalg.lu()**, indicating that if the lower and upper matrices were extracted from the combined LU matrix, and then multiplied together, the result would be the original matrix with rows 0 and 2 swapped and rows 1 and 3 swapped.

Resources

For general information about matrix LU decomposition, see https://en.wikipedia.org/wiki/LU_decomposition.

For details about the SciPy **linalg.lu()** decomposition function, see http://docs.scipy.org/doc/scipy-0.16.0/reference/generated/scipy.linalg.lu.html.

5.3 Statistics

The NumPy and SciPy libraries have a wide range of statistics functions that work with arrays and matrices. Representative examples include the **mean()**, **std()**, **median()**, and **corrcoef()** functions.

Code Listing 24: Statistics Functions Demo

```python
# statistics.py
# Python 2.7

import numpy as np
import math

def my_corr(x, y):
  n = len(x)
  mx = np.mean(x)
  my = np.mean(y)

  num = 0.0
  for i in xrange(n):
    num += (x[i] - mx) * (y[i] - my)
  ssx = 0.0
  ssy = 0.0
  for i in xrange(n):
    ssx += math.pow(x[i] - mx, 2)
    ssy += math.pow(y[i] - my, 2)

  denom = math.sqrt(ssx) * math.sqrt(ssy)
  return num / denom

# =====

print "\nBegin statistics demo \n"

ability = np.array([0., 1., 3., 4., 4., 6.])
payrate = np.array([15., 15., 25., 20., 30., 33. ])

print "ability array = "
```

```python
print ability
print ""

print "payrate array = "
print payrate
print ""

ma = np.median(ability)
print "The median ability score is "
print ma
print ""

s_sd = np.std(payrate, ddof=1)
print "The sample standard deviation of payrates is "
print s_sd
print ""

pr = np.corrcoef(ability, payrate)
print "Pearson r calculated using np.corrcoef() = "
print pr
print ""

pr = my_corr(ability, payrate)
print "Pearson r calculated using my_corr() = "
print "%1.8f" % pr

print "\nEnd demo \n"
```

```
C:\SciPy\Ch5> python statistics.py

Begin statistics demo

ability array =
[ 0.  1.  3.  4.  4.  6.]

payrate array =
[ 15.  15.  25.  20.  30.  33.]

The median ability score is
3.5

The sample standard deviation of payrates is
7.61577310586

Pearson r calculated using np.corrcoef() =
[[ 1.          0.88700711]
 [ 0.88700711  1.        ]]

Pearson r calculated using my_corr() =
0.88700711
End demo
```

The demo program execution begins by setting up two parallel arrays. The first array represents the ability scores of six people. The second array represents the pay rates of the six people:

```
ability = np.array([0., 1., 3., 4., 4., 6.])
payrate = np.array([15., 15., 25., 20., 30., 33. ])
```

Next, after displaying the values in the two arrays, the demo illustrates the use of the NumPy **median()** and **std()** functions:

```
ma = np.median(ability)
print "The median ability score is "
print ma

s_sd = np.std(payrate, ddof=1)
print "The sample standard deviation of payrates is "
print s_sd
```

The median is the middle value in an array or, as in this example when there isn't a single middle value, the average of the two values closest to the middle.

By default, the NumPy **std()** function returns the population standard deviation of its array argument. If you want the sample standard deviation, you can use the **ddof** (delta degrees of freedom) parameter with value = 1.

Next, the demo computes and displays the Pearson r coefficient of correlation using the **corrcoef()** function:

```
pr = np.corrcoef(ability, payrate)
print "Pearson r calculated using np.corrcoef() = "
print pr
```

The correlation coefficient is a value between -1.0 and +1.0, the magnitude indicating the strength of the linear relation and the sign indicating the direction of the relationship. Notice the output is in the form of a matrix with the coefficient value (0.88700711) duplicated on the minor diagonal.

The demo concludes by calling a program-defined function **my_corr()** that also calculates the Pearson r coefficient of correlation:

```
pr = my_corr(ability, payrate)
print "Pearson r calculated using my_corr() = "
print "%1.8f" % pr
```

There's no advantage to using the program-defined correlation function. The point is that NumPy and SciPy have many built-in statistics functions, but in the rare situations when you need to implement a custom statistics function, NumPy and SciPy have all the tools you need.

Resources

For a list of the NumPy statistics functions, see
http://docs.scipy.org/doc/numpy-1.10.0/reference/routines.statistics.html.

For an explanation of the Pearson correlation coefficient that was used for **my_corr()**, see
https://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient.

For information about the NumPy **corrcoef()** function, see
http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.corrcoef.html.

5.4 Random numbers

The NumPy library has a wide range of functions that can generate pseudo-random values from a specified distribution type. Representative examples include the **random.normal()**, **random.poisson()**, **random.exponential()**, and **random.logistic()** functions.

Code Listing 25: Random Sampling Demo

```python
# distributions.py
# Python 2.7

import numpy as np
import math          # for custom Gaussian class
import random        # for custom Gaussian class

class Gaussian:
  # generate using Box-Muller algorithm
  def __init__(self, mean, sd, seed):
    self.mean = mean
    self.sd = sd
    self.rnd = random.Random(seed)

  def next(self):
    two_pi = 2.0*3.14159265358979323846
    u1 = self.rnd.random()  # [0.0 to 1.0)
    while u1 < 1.0e-10:
      u1 = self.rnd.random()
    u2 = self.rnd.random()
    z = math.sqrt(-2.0 * math.log(u1)) * math.cos(two_pi * u2)
    return z * self.sd + self.mean

# =====

print "\nBegin distributions demo \n"

np.random.seed(0)
mean = 0.0
std = 1.0
n = 100

print "Setting mean = " + str(mean)
print "Setting std  = " + str(std)
print ""

print "Generating " + str(n) + " Normal values "
```

```
values = np.zeros(n)
for i in xrange(n):
  x = np.random.normal(mean, std)
  values[i] = x

print "Normally distributed random values are: "
print values
print ""

bins = 5
print "Constructing histogram data using " + str(bins) + " bins "
(histo, edges) = np.histogram(values, bins=5)

print "Count of values in each bin: "
print histo
print ""

print "The beginning and end values of each bin: "
print edges
print ""

print "Generating 5 values using custom Gaussian class: "
g = Gaussian(0.0, 1.0, 0)
for i in xrange(5):
  x = g.next()
  print "%1.5f" % x,
print ""

print "\nEnd demo \n"
```

```
C:\SciPy\Ch5> python distributions.py

Begin distributions demo

Setting mean = 0.0
Setting std  = 1.0

Generating 100 Normal values
Normally distributed random values are:
[ 1.76405235  0.40015721  0.97873798  2.2408932   1.86755799 -0.97727788
  0.95008842 -0.15135721 -0.10321885  0.4105985   0.14404357  1.45427351
  0.76103773  0.12167502  0.44386323  0.33367433  1.49407907 -0.20515826
  0.3130677  -0.85409574 -2.55298982  0.6536186   0.8644362  -0.74216502
  2.26975462 -1.45436567  0.04575852 -0.18718385  1.53277921  1.46935877
  0.15494743  0.37816252 -0.88778575 -1.98079647 -0.34791215  0.15634897
  1.23029068  1.20237985 -0.38732682 -0.30230275 -1.04855297 -1.42001794
 -1.70627019  1.9507754  -0.50965218 -0.4380743  -1.25279536  0.77749036
 -1.61389785 -0.21274028 -0.89546656  0.3869025  -0.51080514 -1.18063218
 -0.02818223  0.42833187  0.06651722  0.3024719  -0.63432209 -0.36274117
 -0.67246045 -0.35955316 -0.81314628 -1.7262826   0.17742614 -0.40178094
 -1.63019835  0.46278226 -0.90729836  0.0519454   0.72909056  0.12898291
```

```
   1.13940068 -1.23482582  0.40234164 -0.68481009 -0.87079715 -0.57884966
  -0.31155253  0.05616534 -1.16514984  0.90082649  0.46566244 -1.53624369
   1.48825219  1.89588918  1.17877957 -0.17992484 -1.07075262  1.05445173
  -0.40317695  1.22244507  0.20827498  0.97663904  0.3563664   0.70657317
   0.01050002  1.78587049  0.12691209  0.40198936]

Constructing histogram data using 5 bins
Count of values in each bin:
[ 6 20 35 27 12]

The beginning and end values of each bin
[-2.55298982 -1.58844093 -0.62389204  0.34065685  1.30520574  2.26975462]

Generating 5 values using custom Gaussian class:
0.02905 -0.07370 -0.95775 -0.22946 -1.05415

End demo
```

The demo program begins by preparing to generate 100 random values that come from a Normal (also called Gaussian or bell-shaped) distribution with mean = 0.0 and standard deviation = 1.0.

```
np.random.seed(0)
mean = 0.0
std = 1.0
n = 100
```

Setting the global random seed, in this case to an arbitrary value of 0, means that the program results will be the same every time the program is run. For a Normal distribution with mean = 0.0, the vast majority of values will be between (-3 * std) and (+3 * std), so we expect all generated values to be in the range [-3.0, +3.0].

Next, the demo program creates an array with 100 cells and fills each cell with a Normal distributed random value:

```
print "Generating " + str(n) + " Normal values "
values = np.zeros(n)
for i in xrange(n):
  x = np.random.normal(mean, std)
  values[i] = x
```

An alternative approach is to create the array directly by supplying a value for the optional **size** parameter: **values = np.normal(mean, std, 100)**. After displaying the 100 values, the demo program constructs histogram information from the values:

```
bins = 5
print "Constructing histogram data using " + str(bins) + " bins "
(histo, edges) = np.histogram(values, bins=5)
```

The NumPy **histogram()** function returns a tuple that has two arrays. The first array stores the count of values in each bin. The second array stores the boundary values for each bin. This is clearer when you examine the output. The statements:

```
print histo
print edges
```

produce the following output:

```
Count of values in each bin:
[ 6 20 35 27 12]

The beginning and end values of each bin:
[-2.55298982 -1.58844093 -0.62389204  0.34065685  1.30520574  2.26975462]
```

This means there were 6 values in the interval [-2.55, -1.58), 20 values in [-1.58, -0.62), 35 values in [-0.62, 0.34), 27 values in [0.34, 1.30), and 12 values in [1.30, 2.26]. If you visually scan the 100 values, you can see the smallest value generated is -2.55298982 and the largest is 2.26975462.

The demo program concludes by showing you how to implement a Normal distribution value generator without using NumPy via a program-defined class named **Gaussian**. The class constructor accepts a mean, a standard deviation, and a seed:

```
class Gaussian:
  def __init__(self, mean, sd, seed):
    self.mean = mean
    self.sd = sd
    self.rnd = random.Random(seed)
```

The class uses a **Random** object from the Python **random** module. The **next()** function uses the clever Box-Muller algorithm to transform two uniform random values into one that is Normal.

```
def next(self):
  two_pi = 2.0*3.14159265358979323846
  u1 = self.rnd.random()  # [0.0 to 1.0]
  while u1 < 1.0e-10:
    u1 = self.rnd.random()
  u2 = self.rnd.random()
  z = math.sqrt(-2.0 * math.log(u1)) * math.cos(two_pi * u2)
  return z * self.sd + self.mean
```

The **while** loop in function **next()** guarantees that variable **u1** is not a very small value so that **log(u1)** won't fail. This example illustrates that it's relatively easy to implement a custom generator in rare situations where NumPy doesn't have the generator you need.

Resources

For a list of NumPy random sampling functions, see
http://docs.scipy.org/doc/numpy-1.10.0/reference/routines.random.html.

For details about the NumPy **histogram()** function, see
http://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.histogram.html.

For information about the Box-Muller algorithm, see
https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform.

5.5 Double factorial

The SciPy library has a collection of useful mathematical functions in the **scipy.misc** submodule. Examples include the **misc.derivative()**, **misc.logsumexp()**, and **misc.factorial2()** functions.

Code Listing 26: Double Factorial Demo

```python
# doublefact.py
# Python 2.7

import scipy.misc as sm

def my_double_fact(n):
  result = 1
  stop = 2 # for even n
  if n % 2 == 0:
    stop = 1 # odd n
  for i in xrange(n, stop-1, -2):
    result *= i
  return result

# =====

print "\nBegin double factorial function demo \n"

n = 3
dfact = sm.factorial2(n)
print "Double factorial of " + str(n) + " using misc.factorial2() = "
print str(dfact)
print ""

n = 4
dfact = sm.factorial2(n)
print "Double factorial of " + str(n) + " using misc.factorial2() = "
print str(dfact)
print ""

n = 4
dfact = my_double_fact(n)
print "Double factorial of " + str(n) + " using my_double_fact() = "
print str(dfact)
print ""

print "\nEnd demo \n"
```

```
C:\SciPy\Ch5> python doublefact.py

Begin double factorial function demo

Double factorial of 3 using misc.factorial2() =
3.0

Double factorial of 4 using misc.factorial2() =
8.0

Double factorial of 4 using my_double_fact() =
8

End demo
```

The demo program illustrates the double factorial function, which is best explained by example. The double factorial of *n* is often abbreviated as *n*!!, much like *n*! is an abbreviation for the regular factorial function.

7!! = 7 * 5 * 3 * 1 = 105
6!! = 6 * 4 * 2 = 48

In words, the double factorial is like the regular factorial function except every other term in the product is skipped in the product. The double factorial function is used as a helper in many important mathematical functions such as the specialized gamma function. The demo program begins by importing the **scipy.misc** submodule:

import scipy.misc as sm

Note that the **factorial2()** function is also in the **scipy.special** submodule. After import, the **factorial2()** function can be called like so:

```
n = 3
dfact = sm.factorial2(n)
print "Double factorial of " + str(n) + " using misc.factorial2() = "
print str(dfact)
```

The **factorial2()** function has an optional parameter **exact** that, if set to **False**, allows the function to do a fast approximation rather than a slower exact calculation.

The demo implements a program-defined version of the double factorial function named **my_double_fact()**. There's no advantage to a program-defined version unless you need some sort of specialized behavior, or wish to avoid importing a module for some reason.

Resources

For details about the `misc.factorial2()` function, see
http://docs.scipy.org/doc/scipy-0.16.1/reference/generated/scipy.misc.factorial2.html.

For information about the double factorial function, including alternate definitions, see
https://en.wikipedia.org/wiki/Double_factorial.

5.6 The gamma function

The SciPy library has a large collection of mathematical functions in the `scipy.special`
submodule. Examples include elliptic functions, Bessel functions, advanced statistical functions,
and gamma functions.

Code Listing 27: Gamma and Special Gamma Demo

```python
# gamma.py
# Python 2.7

import scipy.special as ss
import math

def my_special_gamma(n):
  # return gamma(n/2)
  if n % 2 == 0: # n/2 is an integer
    return math.factorial(n / 2 - 1)
  else:
    root_pi = math.sqrt(math.pi)
    return root_pi * ss.factorial2(n-2) / math.pow(2.0, (n-1) / 2.0)

# =====

print "\nBegin gamma function demo \n"

n = 3
n_fact = math.factorial(n)
print "Factorial of " + str(n) + " = " + str(n_fact)

n = 4
n_fact = math.factorial(n)
print "Factorial of " + str(n) + " = " + str(n_fact)
print ""

n = 5
n_gamma = ss.gamma(n)
print "Gamma of " + str(n) + " using special.gamma() = "
print str(n_gamma)
print ""

n = 4.5
n_gamma = ss.gamma(n)
print "Gamma of " + str(n) + " using special.gamma() = "
```

```
print str(n_gamma)
print ""

n = 9
s_gamma = my_special_gamma(n)
print "Gamma of " + str(n) + "/2 using my_special_gamma() = "
print str(s_gamma)
print ""

print "\nEnd demo \n"
```

```
C:\SciPy\Ch5> python gamma.py

Begin gamma function demo

Factorial of 3 = 6
Factorial of 4 = 24

Gamma of 5 using special.gamma() =
24.0

Gamma of 4.5 using special.gamma() =
11.6317283966

Gamma of 9/2 using my_special_gamma() =
11.6317283966

End demo
```

The factorial function applies only to integers. The gamma function extends the factorial function to real numbers. For example, factorial(3) = 3 * 2 * 1 = 6 and factorial(4) = 4 * 3 * 2 * 1 = 24. However factorial(3.5) is not defined.

For integer arguments, gamma(n) = factorial(n-1). For example, gamma(5) = factorial(4) = 24. For non-integer arguments, such as n = 4.5, the gamma() function returns a value between factorial(3) and factorial(4).

Without a routine like the SciPy **special.gamma()** function, calculating the gamma value for an arbitrary argument like n = 4.68 is difficult. However, there is a relatively easy way to calculate gamma for arguments that are integers divided by two. If n is even, then $n/2$ is an integer and gamma can be calculated using factorial. For example, gamma(10/2) = gamma(5.0) = factorial(4). If n is odd, there is a special equation that can be used. For example, if n = 9 then gamma(9/2) = gamma(4.5) has a shortcut solution. These types of arguments are called positive half-integers. But for all other arguments, calculating gamma is difficult.

The demo program begins by importing the **scipy.special** submodule and the Python math module:

import scipy.special as ss
import math

Next, the demo program calculates and displays the factorial for **n**=3 and **n**=4 in order to illustrate the relationship between `special.gamma(n)` and `math.factorial(n)`:

```
n = 3
n_fact = math.factorial(n)
print "Factorial of " + str(n) + " = " + str(n_fact)
n = 4
n_fact = math.factorial(n)
print "Factorial of " + str(n) + " = " + str(n_fact)
```

Next, the demo calculates and displays the value of gamma(5):

```
n = 5
n_gamma = ss.gamma(n)
print "Gamma of " + str(n) + " using special.gamma() = "
print str(n_gamma)
```

The output is 24.0, verifying that if *n* is an integer, then gamma(*n*) = factorial(*n*-1). Next, the demo calculates and displays the value of gamma(4.5):

```
n = 4.5
n_gamma = ss.gamma(n)
print "Gamma of " + str(n) + " using special.gamma() = "
print str(n_gamma)
```

The point here is that gamma(4.5) = 11.63 is a value between factorial(3) = 6 and factorial(4) = 24.

The demo program implements a program-defined function `my_special_gamma()` that works with positive half-integers:

```
def my_special_gamma(n):
  # return gamma(n/2)
  if n % 2 == 0: # n/2 is an integer
    return math.factorial(n / 2 - 1)
  else:
    root_pi = math.sqrt(math.pi)
    return root_pi * ss.factorial2(n-2) / math.pow(2.0, (n-1) / 2.0)
```

For odd values of **n**, the function's return value is not at all obvious and comes from math theory. Interestingly, even though the `scipy.special` submodule has 17 functions that are related to `gamma()`, there is no dedicated gamma function for positive half-integer arguments.

Resources

For a complete list of the 300+ SciPy special functions, see http://docs.scipy.org/doc/scipy/reference/special.html.

For information about the specialized gamma function for positive half-integers, see https://en.wikipedia.org/wiki/Particular_values_of_the_Gamma_function.