Contents

An Introduction to Pandas

Book Review

Python for Data Analysis is concerned with the nuts and bolts of manipulating, processing, cleaning, and crunching data in Python. It is also a practical, modern introduction to scientific computing in Python, tailored for data-intensive applications. This is a book about the parts of the Python language and libraries you'll need to effectively solve a broad set of data analysis problems. This book is not an exposition on analytical methods using Python as the implementation language.

Written by Wes McKinney, the main author of the pandas library, this hands-on book is packed with practical cases studies. It's ideal for analysts new to Python and for Python programmers new to scientific computing.

- Use the IPython interactive shell as your primary development environment

- Learn basic and advanced NumPy (Numerical Python) features

- Get started with data analysis tools in the pandas library

- Use high-performance tools to load, clean, transform, merge, and reshape data

- Create scatter plots and static or interactive visualizations with matplotlib

- Apply the pandas groupby facility to slice, dice, and summarize datasets

- Measure data by points in time, whether it's specific instances, fixed periods, or intervals

- Learn how to solve problems in web analytics, social sciences, finance, and economics, through detailed examples

Introduction

- Python has been one of the premier general scripting languages, and a major web development language.

- Numerical and data analysis and scientific programming developed through the packages Numpy and Scipy, which, along with the visualization package Matplotlib formed the basis for an open-source alternative to Matlab.

- Numpy provided array objects, cross-language integration, linear algebra and other functionalities.

- Scipy adds to this and provides optimization, linear algebra, optimization, statistics and basic image analysis capabilities.

- Matplotlib provides sophisticated 2-D and basic 3-D graphics capabilities with Matlab-like syntax.

- Further recent development has resulted in a rather complete stack for data manipulation and analysis, that includes Sympy for symbolic mathematics, pandas for data structures and analysis, and IPython as an enhanced console and HTML notebook that also facilitates parallel computation.

Environment Setup

First thing, we'll need a Python environment suitable for scientific and statistical computing.

You should install each in the order they appear here:

- numpy - (pronounced num-pie) Powerful numerical arrays. A foundational package for the two packages below.

- scipy - (sigh-pie) Scientific, mathematical, and engineering package

- scikit-learn - Easy to use machine learning library

Click through the links above for the home pages of each project and get the installation for your operating system or, if you're running Linux, you can install from a package manager (`pip`). If you're on a Windows machine, it's easiest to install using the setup executables for scipy and scikit-learn rather than installing from a package manager.

It is recommend to setting up a decent Python development environment. You can certainly execute Python scripts from the command line, but it's much easier to use a proper environment with debugging support.

1 Python 2.7 vs. 3

Python 2.7 is the final version of the Python 2.x line – all future development work will focus on Python 3. It may seem strange to learn an "old" language. The reasons for using 2.7 are:

- There are more modules available for Python 2.7. While all of the core python modules are available for both Python 2.7 and 3, some of the more esoteric modules are either only available for 2.7 or have not been extensively tested in Python 3. Over time, many of these modules will be available for Python 3, but they aren't ready yet.

- The language changes relevant for numerical computing are very small – and these notes explicitly minimize these so that there should few changes needed to run against Python 3+ in the future (ideally none).

- Configuring and installing 2.7 is easier.

- Anaconda defaults to 2.7 and the selection of packages available for Python 3 is limited.

Learning Python 3 has some advantages:

- No need to update in the future.

- Some improved out-of-box behavior for numerical applications.

1.1 2.A Relevant Differences between Python 2.7 and 3

Most differences between Python 2.7 and 3 are not important for using Python for analysis.

print

print is a function used to display test in the console when running programs. In Python 2.7, `print` is a keyword which behaves differently from other functions. In Python 3, `print` behaves like most functions. The standard use in Python 2.7 is

```
print 'String to Print'
```

while in Python 3 the standard use is

```
print('String to Print')
```

which resembles calling a standard function. Python 2.7 contains a version of the Python 3 `print`, which can be used in any program by including

```
from __future__ import print_function
```

at the top of the file.

Division

- Python 3 changes the way integers are divided. In Python 2.7, the ratio of two integers was always an integer, and so results are truncated towards 0 if the result was fractional. For example, in Python 2.7, 9/5 is 1.

- Python 3 gracefully converts the result to a floating point number, and so in Python 3, 9/5 is 1.8. When working with numerical data, automatically converting ratios avoids some rare errors.

- Python 2.7 can use the Python 3 behavior by including

```
from __future__ import division
```

at the top of the program. We will assume that all relevant programs will include this statement.

range and xrange

It is often useful to generate a sequence of number for use when iterating over the some data. In Python 2.7, the best practice is to use the keyword `xrange` to do this, while in Python 3, this keyword has been renamed `range`. I will always use `xrange` and so it is necessary to replace `xrange` with `range` if using Python 3.

2 Data Types

- Before diving into Python for analyzing data, it is necessary to understand some basic concepts about the core Python data types.

- Unlike MATLAB or R, where the default data type has been chosen for numerical work, Python is a general purpose programming language which is very suited to data analysis.

- For example, the basic numeric type in MATLAB is an array (using double precision, which is useful for floating point mathematics), while the basic numeric data type in Python is a 1-dimensional scalar which may be either an integer or a double-precision floating point, depending on the formatting of the number when input.

2.1 Variable Names

Variable names can take many forms, although they can only contain numbers, letters (both upper and lower), and underscores (_).

They must begin with a letter or an underscore and are CaSe SeNsItIve. Additionally, some words are reserved in Python and so cannot be used for variable names (e.g. `import` or `for`). For example,

```
x = 1.0
X = 1.0
X1 = 1.0
X1 = 1.0
x1 = 1.0
dell = 1.0
dellreturns = 1.0
dellReturns = 1.0
_x = 1.0
x_ = 1.0
```

are all legal and distinct variable names. Note that names which begin or end with an underscore, while legal, are not normally used since by convention these convey special meaning.(What?) Illegal names do not follow these rules.

```
>>> x = []
>>> type(x)
builtins.list
>>> x=[1,2,3,4]
>>> x
[1,2,3,4]
# 2-dimensional list (list of lists)
>>> x = [[1,2,3,4], [5,6,7,8]]
>>> x
[[1, 2, 3, 4], [5, 6, 7, 8]]
# Jagged list, not rectangular
>>> x = [[1,2,3,4] , [5,6,7]]
>>> x
[[1, 2, 3, 4], [5, 6, 7]]
# Mixed data types
>>> x = [1,1.0,1+0j,'one',None,True]
>>> x
[1, 1.0, (1+0j), 'one', None, True]
```

3 Core Native Data Types

3.1 Numeric

- Simple numbers in Python can be either integers, floats or complex. Integers correspond to either 32 bit or 64-bit integers, depending on whether the python interpreter was compiled for a 32-bit or 64-bit operating system, and floats are always 64-bit (corresponding to doubles in C/C++).

- Long integers, on the other hand, do not have a fixed size and so can accommodate numbers which are larger than maximum the basic integer type can handle.

- We will not cover all Python data types, and instead focus on those which are most relevant for data analysis and statistics.

3.1.1 Floating Point (float)

The most important (scalar) data type for numerical analysis is the float. Unfortunately, not all noncomplex numeric data types are floats. To input a floating data type, it is necessary to include a "." (full-stop /period) in the expression. This example uses the function `type()` to determine the data type of a variable.

```
>>> x = 1
>>> type(x)
int
>>> x = 1.0
>>> type(x)
float
>>> x = float(1)
>>> type(x)
float
```

This example shows that using the expression that `x = 1` produces an integer-valued variable while `x = 1.0` produces a float-valued variable. Using integers can produce unexpected results and so it is important to include ".0" when expecting a float.

3.1.2 Complex (`complex`)

Complex numbers are alsooften very important for scientific computing. Complex numbers are created in Python using `j` or the function `complex()`.

```
>>> x = 1.0
>>> type(x)
float
>>> x = 1j
>>> type(x)
complex
>>> x = 2 + 3j
>>> x
(2+3j)
>>> x = complex(1)
>>> x
(1+0j)
```

Note that `a+bj`is the same as `complex(a,b )`, while `complex(a)` is the same as `a+0j`.

3.1.3 Integers (`int` and `long`)

- Floats use an approximation to represent numbers which may contain a decimal portion. The integer data type stores numbers using an exact representation, so that no approximation is needed.

- The cost of the exact representation is that the integer data type cannot express anything that isn't an integer, rendering integers of limited use in most numerical work.

- Basic integers can be entered either by excluding the decimal, or explicitly using the `int()` function.

- The `int()` function can also be used to convert a float to an integer by round towards 0.

]

```
>>> x = 1
>>> type(x)
int
>>> x = 1.0
>>> type(x)
float
>>> x = int(x)
>>> type(x)
int
```

Integers can range from -2^{31} to $2^{31} - 1$. Python contains another type of integer, known as a long integer, which has no effective range limitation. Long integers are entered using the syntax `x = 1L` or by calling `long()`. Additionally python will automatically convert integers outside of the standard integer range to long integers.

```
>>> x = 1
>>> x
1
>>> type(x)
int
>>> x
1L
>>> type(x)
long
>>> x = long(2)
>>> type(x)
long
>>> y = 2
>>> type(y)
int
>>> x = y ** 64 # ** is denotes exponentiation, y^64 in TeX
>>> x
18446744073709551616L
```

3.1.4 Boolean (`bool`)

- The Boolean data type is used to represent true and false, using the reserved keywords `True` and `False`.

- Boolean variables are important for program flow control and are typically created as a result of logical operations , although they can be entered directly.

```
>>> x = True
>>> type(x)
bool
>>> x = bool(1)
>>> x
True
>>> x = bool(0)
>>> x
False
```

Non-zero, non-empty values generally evaluate to true when evaluated by `bool()`. Zero or empty values such as `bool(0)`, `bool(0.0)`, `bool(0.0j)`, `bool(None)`, `bool('')` and `bool([])` are all false.

3.1.5 Strings (`str`)

Strings are not usually important for numerical analysis, although they are frequently encountered when dealing with data files, especially when importing or when formatting output for human consumption. Strings are delimited using ” or ”” but not using combination of the two delimiters (i.e. do not try ”’) in a single string, except when used to express a quotation.

```
>>> x = 'abc'
>>> type(x)
34
str
>>> y = '"A quotation!"'
>>> print(y)
"A quotation!"
```

3.1.6 Lists (`list`)

- Lists are a built-in data type which require other data types to be useful.

- A list is a collection of other objects – floats, integers, complex numbers, strings or even other lists.

- Lists are essential to Python programming and are used to store collections of other values. For example, a list of floats can be used to express a vector (although the NumPy data types `array` and `matrix` are better suited).

- Lists also support slicing to retrieve one or more elements.

- Basic lists are constructed using square braces, [], and values are separated using commas.

```
>>> x = []
>>> type(x)
builtins.list
>>> x=[1,2,3,4]
>>> x
[1,2,3,4]
# 2dimensional
list (list of lists)
>>> x = [[1,2,3,4], [5,6,7,8]]
>>> x
[[1, 2, 3, 4], [5, 6, 7, 8]]
# Jagged list, not rectangular
>>> x = [[1,2,3,4] , [5,6,7]]
>>> x
[[1, 2, 3, 4], [5, 6, 7]]
# Mixed data types
>>> x = [1,1.0,1+0j,'one',None,True]
>>> x
[1, 1.0, (1+0j), 'one', None, True]
```

These examples show that lists can be regular, nested and can contain any mix of data types including other lists.

3.1.7 Xrange (`xrange`)

- A xrange is a useful data type which is most commonly encountered when using a `for` loop.

- `xrange(a,b,i)` creates the sequences that follows the pattern $a, a + i, a + 2i, ..., a + (m-1)i$ where m is the stepsize.

- In other words, it find all integers x starting with a such $a \leq x < b$ and where two consecutive values are separated by i .

- xrange can be called with 1 or two parameters – xrange(a,b) is the same as xrange(a,b,1) and `xrange(b)` is the same as `xrange(0,b,1)`.

```
>>> x = xrange(10)
>>> type(x)
xrange
>>> print(x)
xrange(0, 10)
>>> list(x)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x = xrange(3,10)
>>> list(x)
[3, 4, 5, 6, 7, 8, 9]
>>> x = xrange(3,10,3)
>>> list(x)
[3, 6, 9]
>>> y = range(10)
>>> type(y)
list
>>> y
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- xrange is not technically a list, which is why the statement `print(x)` returns `xrange(0,10)`.

- Explicitly converting with list produces a list which allows the values to be printed. Technically xrange is an iterator which does not actually require the storage space of a list.

- This can be seen in the differences between using `y = range(10)`, which returns a list and `y=xrange(10)` which returns an xrange object.

- Best practice is to use `xrange` instead of `range`.

3.2   inf and NaN

`inf` represents infinity and `inf` is distinct from `-inf`. `inf` can be constructed in a number for ways, for example or exp(710).

`nan` stands for **Not a Number**, and nans are created whenever a function produces a result that cannot be clearly evaluated to produce a number or infinity. For example, `inf/-inf` results in `nan`. nans often cause problems since most mathematical operations involving a `nan` produce a `nan`.

```
>>> x = nan
>>> 1.0 + x
nan
>>> 1.0 * x
nan
>>> 0.0 * x
nan
>>> mean(x)
nan
```

4   Arrays and Matrices

NumPy provides the most important data types for econometrics, statistics and numerical analysis – **arrays** and **matrices**. The difference between these two data types are:

- Arrays can have 1, 2, 3 or more dimensions, and matrices **always** have 2 dimensions. This means that a 1 by n vector stored as an array has 1 dimension and n elements, while the same vector stored as a matrix has 2-dimensions where the sizes of the dimensions are 1 and n (in either order).

- Standard mathematical operators on arrays operate element-by-element. This is not the case for matrices, where multiplication ($*$) follows the rules of *linear algebra*.

- 2-dimensional arrays can be multiplied using the rules of linear algebra using dot.

- Similarly, the function multiply can be used on two matrices for *element-by-element* multiplication.

- Arrays are more common than matrices, and all functions are thoroughly tested with arrays.

- Functions should also work with matrices, but an occasional strange result may be encountered.

- Arrays can be quickly treated as a matrix using either `asmatrix` or `mat` without copying the underlying data.

The best practice is to use arrays and to use the `asmatrix` view when writing linear algebra-heavy code. It is also important to test any custom function with both arrays and matrices to ensure that false assumptions about the behavior of multiplication have not been made.

4.1 Arrays

Arrays are the base data type in NumPy, are are arrays in some ways similar to lists since they both contain collections of elements. The focus of this section is on *homogeneous* arrays containing numeric data – that is, an array where all elements have the same numeric type

Additionally, arrays, unlike lists, are always rectangular so that all rows have the same number of elements.

Arrays are initialized from lists (or tuples) using array. Two-dimensional arrays are initialized using lists of lists (or tuples of tuples, or lists of tuples, etc.), and higher dimensional arrays can be initialized by further nesting lists or tuples.

```
>>> x = [0.0, 1, 2, 3, 4]
>>> y = array(x)
>>> y
array([ 0., 1., 2., 3., 4.])
>>> type(y)
numpy.ndarray
```

Two (or higher) -dimensional arrays are initialized using nested lists.

```
>>> y = array([[0.0, 1, 2, 3, 4], [5, 6, 7, 8, 9]])
>>> y
array([[ 0., 1., 2., 3., 4.],
[ 5., 6., 7., 8., 9.]])
```

```
>>> shape(y)
(2L, 5L)
>>> y = array([[[1,2],[3,4]],[[5,6],[7,8]]])
>>> y
array([[[1, 2],
[3, 4]],
[[5, 6],
[7, 8]]])
>>> shape(y)
(2L, 2L, 2L)
```

Array dtypes

Homogeneous arrays can contain a variety of numeric data types. The most useful is 'float64', which corresponds to the python built-in data type of float (andC/C++double). By default, calls to array will preserve the type of the input, if possible. If an input contains all integers, it will have a dtype of 'int32' (similar to the built in data type int). If an input contains integers, floats, or a mix of the two, the array's data type will be float64. If the input contains a mix of integers, floats and complex types, the array will be initialized to hold complex data.

```
>>> x = [0, 1, 2, 3, 4] # Integers
>>> y = array(x)
>>> y.dtype
dtype('int32')
>>> x = [0.0, 1, 2, 3, 4] # 0.0 is a float
>>> y = array(x)
>>> y.dtype
dtype('float64')
>>> x = [0.0 + 1j, 1, 2, 3, 4] # (0.0 + 1j) is a complex
>>> y = array(x)
>>> y
array([ 0.+1.j, 1.+0.j, 2.+0.j, 3.+0.j, 4.+0.j])
>>> y.dtype
dtype('complex128')
```

NumPy attempts to find the smallest data type which can represent the data

when constructing an array. It is possible to force NumPy to select a particular dtype by using the keyword argument dtype=datetype when initializing the array.

```
>>> x = [0, 1, 2, 3, 4] # Integers
>>> y = array(x)
>>> y.dtype
dtype('int32')
>>> y = array(x, dtype='float64') # String dtype
>>> y.dtype
dtype('float64')
>>> y = array(x, dtype=float64) # NumPy type dtype
>>> y.dtype
dtype('float64')
```

4.2 Matrix

Matrices are essentially a subset of arrays, and behave in a virtually identical manner. The two important differences are:

- Matrices always have 2 dimensions

- Matrices follow the rules of linear algebra for Matrix Multiplication (∗)

```
>>> x = [0.0, 1, 2, 3, 4] # Any float makes all float
>>> y = array(x)
>>> type(y)
numpy.ndarray
>>> y * y # Elementbyelement
array([ 0., 1., 4., 9., 16.])
>>> z = asmatrix(x)
>>> type(z)
numpy.matrixlib.defmatrix.matrix
>>> z * z # Error
ValueError: matrices are not aligned
```

4.3 1-dimensional Arrays

A vector

$$x = [1 2 3 4 5]$$

is entered as a 1-dimensional array using

```
>>> x=array([1.0,2.0,3.0,4.0,5.0])
array([ 1., 2., 3., 4., 5.])
>>> ndim(x)
1
```

If an array with 2-dimensions is required, it is necessary to use a trivial nested list.

```
>>> x=array([[1.0,2.0,3.0,4.0,5.0]])
array([[ 1., 2., 3., 4., 5.]])
>>> ndim(x)
2
```

A matrix is always 2-dimensional and so a nested list is not required to initialize a a row matrix

```
>>> x=matrix([1.0,2.0,3.0,4.0,5.0])
>>> x
matrix([[ 1., 2., 3., 4., 5.]])
>>> ndim(x)
2
```

Notice that the output matrix representation uses nested lists ([[ ]]) to emphasize the 2-dimensional structure of all matrices. The column vector,

$$x = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

is entered as a matrix or 2-dimensional array using a set of nested lists

```
>>> x=matrix([[1.0],[2.0],[3.0],[4.0],[5.0]])
>>> x
matrix([[ 1.],
 [ 2.],
 [ 3.],
 [ 4.],
 [ 5.]])
>>> x = array([[1.0],[2.0],[3.0],[4.0],[5.0]])
>>> x
array([[ 1.],
 [ 2.],
 [ 3.],
 [ 4.],
 [ 5.]])
```

4.4   2-dimensional Arrays

Matrices and 2-dimensional arrays are rows of columns, and so

$$x = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

is input by enter the matrix one row at a time, each in a list, and then encapsulate the row lists in another list.

```
>>> x = array([[1.0,2.0,3.0],[4.0,5.0,6.0],[7.0,8.0,9.0]])
>>> x
array([[ 1., 2., 3.],
[ 4., 5., 6.],
[ 7., 8., 9.]])
```

4.5 Multidimensional Arrays

- Higher dimensional arrays are useful when tracking matrix valued processes through time, such as a time varying covariance matrices.

- Multidimensional (N -dimensional) arrays are available for N up to about 30,

- depending on the size of each matrix dimension.

- Manually initializing higher dimension arrays is tedious and error prone, and so it is better to use functions such as `zeros((2, 2, 2))` or `empty((2, 2, 2))`.

4.6 Concatenation

Concatenation is the process by which one vector or matrix is appended to another. Arrays and matrices can be concatenation horizontally or vertically

4.7 Accessing Elements of an Array

- Four methods are available for accessing elements contained within an array: scalar selection, slicing, numerical indexing and logical (or Boolean) indexing.

- Scalar selection and slicing are the simplest and so are presented first.

- Numerical indexing and logical indexing both depends on specialized functions and so these methods are discussed in Chapter 12.

4.8 The `import` function

- Python, by default, only has access to a small number of built-in types and functions. The vast majority of functions are located in modules, and before a function can be accessed, the module which contains the function must be imported.

- For example, when using `ipython --pylab` (or any variants), a large number of modules are automatically imported, including NumPy and matplotlib.

- This is style of importing useful for learning and interactive use, but care is needed to make sure that the correct module is imported when designing more complex programs.

- `import` can be used in a variety of ways. The simplest is to use from module `import *` which imports all functions in module.

- This method of using import can dangerous since if you use it more than once, it is possible for functions to be hidden by later imports.

- A better method is to just import the required functions.

- This still places functions at the top level of the namespace, but can be used to avoid conflicts.

```
from pylab import log2 # Will import log2 only
from scipy import log10 # Will not import the log2 from SciPy
```

- The only difference between these two is that `import scipy` is implicitly calling `import scipy as scipy`.

- When this formof import is used, functions are used with the "as" name. For example, the load provided by NumPy is accessed using `sp.log2`, while the pylab load is `pl.log2` – and both can be used where appropriate.

- While this method is the most general, it does require slightly more typing.

4.9 Special Arrays

Functions are available to construct a number of useful, frequently encountered arrays.

4.9.1 ones

ones generates an array of 1s and is generally called with one argument, a tuple, containing the size of each dimension. ones takes an optional second argument (dtype) to specify the data type. If omitted, the data type is float.

```
>>> M, N = 5, 5
>>> x = ones((M,N)) # M by N array of 1s
>>> x = ones((M,M,N)) # 3D array
>>> x = ones((M,N), dtype='int32') # 32bit integers
```

ones_like creates an array with the same shape and data type as the input. Calling ones_like(x) is equivalent to calling ones(x.shape,x.dtype).

4.9.2 zeros

zeros produces an array of 0s in the same way ones produces an array of 1s, and commonly used to initialize an array to hold values generated by another procedure. zeros takes an optional second argument (dtype) to specify the data type. If omitted, the data type is float.

```
>>> x = zeros((M,N)) # M by N array of 0s
>>> x = zeros((M,M,N)) # 3D array of 0s
>>> x = zeros((M,N),dtype='int64') # 64 bit integers
```

zeros_like creates an array with the same size and shape as the input. Calling zeros_like(x) is equivalent to calling zeros(x.shape,x.dtype).

4.9.3 empty

empty produces an empty (uninitialized) array to hold values generated by another procedure. empty takes an optional second argument (dtype) which specifies the data type. If omitted, the data type is float.

```
>>> x = empty((M,N)) # M by N empty array
>>> x = empty((N,N,N,N)) # 4D empty array
>>> x = empty((M,N),dtype='float32') # 32bit
```

4.10 floats (single precision)

Using empty is slightly faster than calling zeros since it does not assign 0 to all elements of the array – the "empty" array created will be populated with (essential random) non-zero values. empty_like creates an array with the same size and shape as the input. Calling empty_like(x) is equivalent to calling empty(x.shape,x.dtype).

4.10.1 eye, identity

eye generates an identity array – an array with ones on the diagonal, zeros everywhere else. Normally, an identity array is square and so usually only 1 input is required. More complex zero-padded arrays containing an identity matrix can be produced using optional inputs.

```
>>> In = eye(N)
```

identity is a virtually identical function with similar use, In = identity(N).

5 Basic Functions and Numerical Indexing

5.1 Mathematics

`sum, cumsum`

`sum` sums elements in an array. By default, it will sum all elements in the array, and so the second argument is normally used to provide the axis to use – 0 to sum down columns, 1 to sum across rows. `cumsum` produces the cumulative sum of the values in the array, and is also usually used with the second argument to indicate the axis to use.

```
>>> x = randn(3,4)
>>> x
array([[0.08542071,
2.05598312,
2.1114733 , 0.7986635 ],
[0.17576066,
0.83327885, 0.64064119,
0.25631728],
[0.38226593,
```

```
1.09519101,
0.29416551, 0.03059909]])
>>> sum(x) # all elements
0.62339964288008698
>>> sum(x, 0) # Down rows, 4 elements
array([0.6434473
, 2.31789529,
1.76499762, 0.57294532])
>>> sum(x, 1) # Across columns, 3 elements
array([ 0.76873297, 0.23944028,
1.15269233])
>>> cumsum(x,0) # Down rows
array([[0.08542071,
2.05598312,
2.1114733 , 0.7986635 ],
[0.26118137,
1.22270427,
1.47083211, 0.54234622],
[0.6434473
, 2.31789529,
1.76499762, 0.57294532]])
```

sum and `cumsum` can both be used as function or as methods. When used as methods, the first input is the axis so that `sum(x,0)` is the same as `x.sum(0)`.

prod, cumprod

prod and `cumprod` behave similarly to `sum` and `cumsum` except that the product and cumulative product are returned. `prod` and `cumprod` can be called as function or methods.

diff

diff computes the finite difference of a vector (also array) and returnsn-1 an element vectorwhenused on an n element vector. diff operates on the last axis by default, and so diff(x) operates across columns and returns `x[:,1:size(x,1)]x[:,: size(x,1)1]` for a 2-dimensional array.

diff takes an optional keyword argument axis so that diff(x, axis=0) will operate across rows. diff can also be used to produce higher order differences (e.g. double difference).

```
>>> x= randn(3,4)
>>> x
array([[0.08542071,
2.05598312,
2.1114733 , 0.7986635 ],
[0.17576066,
0.83327885, 0.64064119,
0.25631728],
[0.38226593,
1.09519101,
0.29416551, 0.03059909]])
>>> diff(x) # Same as diff(x,1)
0.62339964288008698
>>> diff(x, axis=0)
array([[0.09033996,
2.88926197, 2.75211449,
1.05498078],
[0.20650526,
1.92846986,
0.9348067 , 0.28691637]])
>>> diff(x, 2, axis=0) # Double difference, columnbycolumn
array([[0.11616531,
4.81773183,
3.68692119, 1.34189715]])
```

exp

exp returns the element-by-element exponential (e^x) for an array.

log

log returns the element-by-element natural logarithm $(ln(x))$ for an array.

log10

log10 returns the element-by-element base-10 logarithm (log10 (x)) for an array.

5.2 Rounding

`around, round`

around rounds to the nearest integer, or to a particular decimal place when called with two arguments.

```
%---------------%
\begin{framed}
\begin{verbatim}
>>> x = randn(3)
array([ 0.60675173, 0.3361189
, 0.56688485])
>>> around(x)
array([ 1., 0., 1.])
>>> around(x, 2)
array([ 0.61, 0.34,
0.57])
```

`around` can also be used as a method on an ndarray – except that the method is named round. For example, `x.round(2)` is identical to `around(x, 2)`. The change of names is needed to avoid conflicting with the Python built-in function round.

`floor`

floor rounds to the next smallest integer.

```
>>> x = randn(3)
array([ 0.60675173, 0.3361189
, 0.56688485])
>>> floor(x)
array([ 0., 1.,
1.])
```

`ceil`

ceil rounds to the next largest integer.

```
>>> x = randn(3)
array([ 0.60675173, 0.3361189
, 0.56688485])
>>> ceil(x)
array([ 1., 0.,
0.])
```

Note that the values returned are still floating points and so 0. is the same as 0..

5.3   Generating Arrays and Matrices

`linspace`

`linspace(l,u,n)` generates a set of n points uniformly spaced between l, a lower bound (inclusive) and u, an upper bound (inclusive).

```
>>> x = linspace(0, 10, 11)
>>> x
array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9., 10.])
```

`logspace`

`logspace(l,u,n)` produces a set of logarithmically spaced points between 10l and 10u . It is identical to 10**linspace(l,u,n).

`arange`

`arange(l,u,s)` produces a set of points spaced by s between l, a lower bound (inclusive) and u, an upper bound (exclusive). arange can be used with a single parameter, so that arange(n) is equivalent to arange(0,n,1). Note that arange will return integer data type if all inputs are integer.

```
>>> x = arange(11)
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
>>> x = arange(11.0)
```

```
array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9., 10.])
>>> x = arange(4, 10, 1.25)
array([ 4. , 5.25, 6.5 , 7.75, 9. ])
```

5.3.1 `meshgrid`

`meshgrid` broadcasts two vectors to produce two 2-dimensional arrays, and is a useful function when plotting 3-dimensional functions.

```
>>> x = arange(5)
>>> y = arange(3)
>>> X,Y = meshgrid(x,y)
>>> X
array([[0, 1, 2, 3, 4],
[0, 1, 2, 3, 4],
[0, 1, 2, 3, 4]])
>>> Y
array([[0, 0, 0, 0, 0],
[1, 1, 1, 1, 1],
[2, 2, 2, 2, 2]])
```

5.3.2 `r_`

`r_` is a convenience function which generates 1-dimensional arrays from slice notation. While `r_` is highly flexible, the most common use it r_[start : end : stepOrCount] where start and end are the start and end points, and stepOrCount can be either a step size, if a real value, or a count, if complex.

```
>>> r_[0:10:1] # arange equiv
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> r_[0:10:.5] # arange equiv
array([ 0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ,
5.5, 6. , 6.5, 7. , 7.5, 8. , 8.5, 9. , 9.5])
>>> r_[0:10:5j] # linspace equiv, includes end point
array([ 0. , 2.5, 5. , 7.5, 10. ])
```

r₋ can also be used to concatenate slices using commas to separate slice notation blocks.

```
>>> r_[0:2, 7:11, 1:4]
array([ 0, 1, 7, 8, 9, 10, 1, 2, 3])
```

Note that r₋ is not a function and that is used with [].

5.3.3 c₋

c₋ is virtually identical to r₋ except that column arrays are generates, which are 2-dimensional (second dimension has size 1).

```
>>> c_[0:5:2]
array([[0],
[2],
[4]])
>>> c_[1:5:4j]
array([[ 1. ],
[ 2.33333333],
[ 3.66666667],
[ 5. ]])
```

c₋, like r₋, is not a function and is used with [].

5.3.4 ix₋

ix₋(a,b) constructs an n-dimensional open mesh from n 1-dimensional lists or arrays. The output of ix₋ is an n-element tuple containing 1-dimensional arrays. The primary use of ix₋ is to simplify selecting slabs inside a matrix. Slicing can also be used to select elements froman array as long as the slice pattern is regular. ix₋ is particularly useful for selecting elements froman array using indices which are not regularly spaced, as in the final example.

```
>>> x = reshape(arange(25.0),(5,5))
>>> x
```

```
array([[ 0., 1., 2., 3., 4.],
[ 5., 6., 7., 8., 9.],
[ 10., 11., 12., 13., 14.],
[ 15., 16., 17., 18., 19.],
[ 20., 21., 22., 23., 24.]])
>>> x[ix_([2,3],[0,1,2])] # Rows 2 & 3, cols 0, 1 and 2
array([[ 10., 11., 12.],
[ 15., 16., 17.]])
>>> x[2:4,:3] # Same, standard slice
array([[ 10., 11., 12.],
[ 15., 16., 17.]])
>>> x[ix_([0,3],[0,1,4])] # No slice equiv
```

5.3.5 `mgrid`

`mgrid` is very similar to meshgrid but behaves like `r_` and `c_` in that it takes slices as input, and uses a real valued variable to denote step size and complex to denote number of values. The output is an $n + 1$ dimensional vector where the first index of the output indexes the meshes.

```
>>> mgrid[0:3,0:2:.5]
>>>
>>> mgrid[0:3:3j,0:2:5j]
>>>
```

5.3.6 `ogrid`

`ogrid` is identical to mgrid except that the arrays returned are always 1-dimensional. `ogrid` output is generally more appropriate for looping code, while `mgrid` is usually more appropriate for vectorized code. When the size of the arrays is large, then `ogrid` uses much less memory.

```
>>> ogrid[0:3,0:2:.5]
[array([[ 0.],
[ 1.],
```

```
[ 2.]]), array([[ 0. , 0.5, 1. , 1.5]])]
>>> ogrid[0:3:3j,0:2:5j]
[array([[ 0. ],
[ 1.5],
[ 3. ]]),
array([[ 0. , 0.5, 1. , 1.5, 2. ]])]
```

6   Useful Array and Matrix Functions

Many functions operate exclusively on array inputs, including functions which are mathematical in nature, for example computing the eigenvalues and eigenvectors and functions for manipulating the elements of an array.

6.1   Views

Views are computationally efficient methods to produce objects of one type which behave as other objects of another type without copying data. For example, an array x can always be converted to a matrix using `matrix(x)`, which will copy the elements in x. View "fakes" the call to matrix and only inserts a thin layer so that x viewed as a matrix behaves like a matrix.

6.1.1   `view`

`view` can be used to produce a representation of an array, matrix or recarray as another type without copying the data. Using view is faster than copying data into a new class.

```
>>> x = arange(5)
>>> type(x)
numpy.ndarray
>>> x.view(matrix)
matrix([[0, 1, 2, 3, 4]])
>>> x.view(recarray)
rec.array([0, 1, 2, 3, 4])
```

6.1.2 `asmatrix`, `mat`

asmatrix and mat can be used to view an array as a matrix. This view is useful since matrix views will use matrix multiplication by default.

```
>>> x = array([[1,2],[3,4]])
>>> x * x # Elementbyelement
array([[ 1, 4],
[ 9, 16]])
>>> mat(x) * mat(x) # Matrix multiplication
matrix([[ 7, 10],
[15, 22]])
```

Both commands are equivalent to using view(matrix).

6.1.3 `asarray`

asarray work in a similar matter as asmatrix, only that the view produced is that of ndarray. Calling asarray is equivalent to using view(ndarray)

6.2 Shape Information and Transformation

6.2.1 `shape`

`shape` returns the size of all dimensions or an array or matrix as a tuple. shape can be called as a function or an attribute. `shape` can also be used to reshape an array by entering a tuple of sizes. Additionally, the new shape can contain -1 which indicates to expand along this dimension to satisfy the constraint that the number of elements cannot change.

```
>>> x = randn(4,3)
>>> x.shape
(4L, 3L)
>>> shape(x)
(4L, 3L)
>>> M,N = shape(x)
>>> x.shape = 3,4
>>> x.shape
(3L, 4L)
```

```
>>> x.shape = 6,-1
>>> x.shape
(6L, 2L)
```

6.2.2 `reshape`

`reshape` transforms an array with one set of dimensions and to one with a different set, preserving the number of elements. Arrays with dimensions M by N can be reshaped into an array with dimensions K by L as long as M N = K L. The most useful call to reshape switches an array into a vector or vice versa.

```
>>> x = array([[1,2],[3,4]])
>>> y = reshape(x,(4,1))
>>> y
array([[1],
[2],
[3],
[4]])
>>> z=reshape(y,(1,4))
>>> z
array([[1, 2, 3, 4]])
>>> w = reshape(z,(2,2))
array([[1, 2],
[3, 4]])
```

The crucial implementation detail of reshape is that arrays are stored using row-major notation. Elements in arrays are counted first across rows and then then down columns. reshape will place elements of the old array into the same position in the new array and so after calling reshape, `x (1) = y (1)`, `x (2) = y (2)`, and so on.

6.2.3 `size`

`size` returns the total number of elements in an array or matrix. `size` can be used as a function or an attribute.

```
>>> x = randn(4,3)
>>> size(x)
12
>>> x.size
12
```

6.2.4 ndim

ndim returns the size of all dimensions or an array or matrix as a tuple. ndim can be used as a function or an attribute .

```
>>> x = randn(4,3)
>>> ndim(x)
2
>>> x.ndim
2
```

6.2.5 tile

tile, along with reshape, are two of the most useful non-mathematical functions. tile replicates an array according to a specified size vector. To understand how tile functions, imagine forming an array composed of blocks. The generic form of tile is tile(X , (M, N)) where X is the array to be replicated, M is the number of rows in the new block array, and N is the number of columns in the new block array.

```
>>> x = array([[1,2],[3,4]])
>>> z = hstack((x,x,x))
>>> y = vstack((z,z))
```

However, tile provides a much easier method to construct y

```
>>> w = tile(x,(2,3))
>>> y w
array([[0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0]])
```

`tile` has two clear advantages over manual allocation:

- First, tile can be executed using parameters determined at run-time, such as the number of explanatory variables in a model,

- second tile can be used for arbitrary dimensions. Manual array construction becomes tedious and error prone with as few as 3 rows and columns.

`repeat` is a related function which copies data is a less useful manner.

6.2.6 `ravel`

`ravel` returns a flattened view (1-dimensional) of an array or matrix. `ravel` does not copy the underlying data (when possible), and so it is very fast.

```
>>> x = array([[1,2],[3,4]])
>>> x
array([[ 1, 2],
[ 3, 4]])
>>> x.ravel()
array([1, 2, 3, 4])
>>> x.T.ravel()
array([1, 3, 2, 4])
```

6.2.7 `flatten`

`flatten` works much like `ravel`, only that is copies the array when producing the flattened version.

6.2.8 `flat`

`flat` produces a `numpy.flatiter` object (`flat` iterator) which is an iterator over a flattened view of an array. Because it is an iterator, it is especially fast and

memory friendly. flat can be used as an iterator in a for loop or with slicing notation.

```
>>> x = array([[1,2],[3,4]])
>>> x.flat
<numpy.flatiter at 0x6f569d0>
>>> x.flat[2]
3
>>> x.flat[1:4] = 1
>>> x
array([[ 1, 1],
[1,
1]])
```

6.2.9 `split`, `vsplit`, `hsplit`

`vsplit` and `hsplit` split arrays and matrices vertically and horizontally, respectively. Both can be used to split an array into n equal parts or into arbitrary segments, depending on the second argument. If scalar, the array is split into n equal sized parts.

If a 1 dimensional array, the array is split using the elements of the array as break points. For example, if the array was [2,5,8], the array would be split into 4 pieces using [:2] , [2:5], [5:8] and [8:]. Both `vsplit` and `hsplit` are special cases of split, which can split along an arbitrary axis.

```
>>> x = reshape(arange(20),(4,5))
>>> y = vsplit(x,2)
>>> len(y)
2
>>> y[0]
array([[0, 1, 2, 3, 4],
[5, 6, 7, 8, 9]])
>>> y = hsplit(x,[1,3])
>>> len(y)
3
>>> y[0]
array([[ 0],
[ 5],
```

```
[10],
[15]])
>>> y[1]
array([[ 1,  2],
[ 6,  7],
[11, 12],
[16, 17]])
```

6.2.10   `delete`

`delete` removes values from an array, and is similar to splitting an array, and then concatenating the values which are not deleted. The form of `delete` is `delete(x,rc, axis)` where rc are the row or column indices to delete, and axis is the axis to use (0 or 1 for a 2-dimensional array). If axis is omitted, `delete` operated on the flattened array.

```
>>> x = reshape(arange(20),(4,5))
>>> delete(x,1,0) # Same as x[[0,2,3]]
array([[ 0,  1,  2,  3,  4],
[10, 11, 12, 13, 14],
[15, 16, 17, 18, 19]])
>>> delete(x,[2,3],1) # Same as x[:,[0,1,4]]
array([[ 0,  1,  4],
[ 5,  6,  9],
[10, 11, 14],
[15, 16, 19]])
>>> delete(x,[2,3]) # Same as hstack((x.flat[:2],x.flat[4:]))
array([ 0,  1,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19])
```

6.2.11   `squeeze`

`squeeze` removes *singleton* dimensions from an array, and can be called as a function or a method.

```
>>> x = ones((5,1,5,1))
>>> shape(x)
(5L, 1L, 5L, 1L)
>>> y = x.squeeze()
>>> shape(y)
(5L, 5L)
>>> y = squeeze(x)
```

6.2.12 fliplr, flipud

fliplr and flipud flip arrays in a left-to-right and up-to-down directions, respectively. flipud reverses the elements in a 1-dimensional array, and flipud(x) is identical to x[::1]. fliplr cannot be used with 1-dimensional arrays.

```
>>> x = reshape(arange(4),(2,2))
>>> x
array([[0, 1],
[2, 3]])
>>> fliplr(x)
array([[1, 0],
[3, 2]])
>>> flipud(x)
array([[2, 3],
[0, 1]])
```

6.2.13 diag

The behavior of diag differs depending depending on the form of the input.

- If the input is a square array, it will return a column vector containing the elements of the diagonal.

- If the input is an vector, it will return an array containing the elements of the vector along its diagonal.

Consider the following example:

```
>>> x = array([[1,2],[3,4]])
>>> x
array([[1, 2],
[3, 4]])
>>> y = diag(x)
>>> y
array([1, 4])
>>> z = diag(y)
>>> z
array([[1, 0],
[0, 4]])
```

6.2.14 triu, tril

triu and tril produce upper and lower triangular arrays, respectively.

```
>>> x = array([[1,2],[3,4]])
>>> triu(x)
array([[1, 2],
[0, 4]])
>>> tril(x)
array([[1, 0],
[3, 4]])
```

6.3   Some Useful Linear Algebra Functions

`det`

`det` computes the determinant of a square matrix or array.

```
>>> x = matrix([[1,.5],[.5,1]])
>>> det(x)
0.75
```

`solve`

`solve` solves the system $Ax=b$ when X is square and invertible so that the solution is exact.

```
>>> A = array([[1.0,2.0,3.0],[3.0,3.0,4.0],[1.0,1.0,4.0]])
>>> b = array([[1.0],[2.0],[3.0]])
>>> solve(A,b)
array([[ 0.625],
[1.125],
[ 0.875]])
```

`eig`

`eig` computes the eigenvalues and eigenvectors of a square matrix. When used with one output, the eigenvalues and eigenvectors are returned as a tuple.

```
>>> x = matrix([[1,.5],[.5,1]])
>>> val,vec = eig(x)
>>> vec*diag(val)*vec.T
matrix([[ 1. , 0.5],
[ 0.5, 1. ]])
```

`eigvals` can be used if only eigenvalues are needed.

7 Logical and Relational Operators

Logical operators are useful when writing code. Logical operators, when combined with flow control, allow for complex choices to be compactly expressed.

7.1 Relational Operators

The core logical operators are

Symbol	Function	Definition
>	greater	Greater than
>=	greater_equal	Greater than or equal to
<	less	Less than
<=	less_equal	Less than or equal to
==	equal	Equal to
!=	not_equal	Not equal to

```
>>> x = array([[1,2],[3,
4]])
>>> x > 0
array([[ True, True],
[False, False]], dtype=bool)
>>> x == 3
array([[False, False],
[ True, False]], dtype=bool)
>>> y = array([1,1])
>>> x < y # y broadcast to be (2,2)
array([[False, False],
[ True, True]], dtype=bool)
>>> z = array([[1,1],[1,
1]])
# Same as broadcast y
>>> x < z
array([[False, False],
[ True, True]], dtype=bool)
```

7.2 Logical Operators

Logical expressions can be combined using four logical devices,

Keyword (Scalar)	Function	Bitwise	True if . . .
and	logical_and	Both	True
or	logical_or		Either or Both True
not	logical_not		Not True
	logical_xor	∧	One True and One False

There are three versions of all operators except XOR. The keyword version (e.g. and) can only be used with scalars and so it not useful when working with NumPy. Both the function and bitwise operators can be used with NumPy arrays, although care is requires when using the bitwise operators.

```
>>> x = arange(2.0,4)
>>> y = x >= 0
>>> z = x < 2
>>> logical_and(y, z)
array([False, False, True, True, False, False], dtype=bool)
>>> y & z
array([False, False, True, True, False, False], dtype=bool)
>>> (x > 0) & (x < 2)
array([False, False, True, True, False, False], dtype=bool)
>>>
>>> x > 0 & x < 4 # Error
TypeError: ufunc 'bitwise_and' not supported for the input types,
and the inputs could not
be safely coerced to any supported types
according to the casting rule ''safe''
>>> ~(y & z) # Not
array([ True, True, False, False, True, True], dtype=bool)
>>>
```

7.3 Bitwise operators

Bitwise operators have high priority – higher than logical comparisons – and so parentheses are requires around comparisons. For example, $(x > 1)\&(x < 5)$ is a valid statement, while $x > 1\&x < 5$, which is evaluated as $(x > (1\&x)) < 5$, produces an error.

```
>>> x = arange(2.0,4)
>>> y = x >= 0
>>> z = x < 2
>>> logical_and(y, z)
array([False, False, True, True, False, False], dtype=bool)
>>> y & z
array([False, False, True, True, False, False], dtype=bool)
>>> (x > 0) & (x < 2)
array([False, False, True, True, False, False], dtype=bool)
```

7.4 Multiple tests: `all` and `any`

The commands all and any take logical input and are self-descriptive. all returns True if all logical elements in an array are 1.

- If `all` is called without any additional arguments on an array, it returns True if all elements of the array are logical true and 0 otherwise.

- any returns logical(True) if any element of an array is True.

Both `all` and `any` can be also be used along a specific dimension using a second argument or the keyword argument axis to indicate the axis of operation (0 is column-wise and 1 is row-wise).

When used column- or row-wise, the output is an array with one less dimension than the input, where each element of the output contains the truth value of the operation on a column or row.

```
>>> x = array([[1,2][3,4]])
>>> y = x <= 2
>>> y
array([[ True, True],
[False, False]], dtype=bool)
>>> any(y)
True
>>> any(y,0)
array([[ True, True]], dtype=bool)
>>> any(y,1)
array([[ True],
[False]], dtype=bool)
```

7.4.1 allclose

allclose can be used to compare two arrays for near equality. This type of function is important when comparing floating point values which may be effectively the same although not identical.

```
>>> eps = np.finfo(np.float64).eps
>>> eps
2.2204460492503131e16
```

```
>>> x = randn(2)
>>> y = x + eps
115
>>> x == y
array([False, False], dtype=bool)
>>> allclose(x,y)
True
```

The tolerance for being close can be set using keyword arguments either relatively (rtol) or absolutely (atol).

7.4.2 array_equal

array_equal tests if two arrays have the same shape and elements. It is safer than comparing arrays directly since comparing arrays which are not broadcastable produces an error.

7.4.3 array_array_equiv

array_equiv tests if two arrays are equivalent, even if they do not have the exact same shape. Equivalence is defined as one array being broadcastable to produce the other.

```
>>> x = randn(10,1)
>>> y = tile(x,2)
>>> array_equal(x,y)
False
>>> array_equiv(x,y)
True
```

7.5   is∗

A number of special purpose logical tests are provided to determine if an array has special characteristics. Some operate element-by-element and produce an array of the same dimension as the input while other produce only scalars. These functions all begin with is.

Operator	True if . . .	Method of operation
isnan	1 if nan	element-by-element
isinf	1 if inf	element-by-element
isfinite	1 if not inf and not nan	element-by-element
isposfin,isnegfin	1 for positive or negative inf	element-by-element
isreal	1 if not complex valued	element-by-element
iscomplex	1 if complex valued	element-by-element
isreal	1 if real valued	element-by-element
is_string_like	1 if argument is a string	scalar
is_numlike	1 if is a numeric type	scalar
isscalar	1 if scalar	scalar
isvector	1 if input is a vector	scalar