Contents

An Introduction to Pandas

This tutorial will get you started with Pandas - a data analysis library for Python that is great for data preparation, joining, and ultimately generating well-formed, tabular data that's easy to use in a variety of visualization tools or (as we will see here) machine learning applications.

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.

references

1 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.

1.1 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]
```

2 Core Native Data Types

2.1 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.

2.1.1 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.

2.1.2 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`.

2.1.3 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
```

2.1.4 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.

2.1.5 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!"
```

2.1.6 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.

2.1.7 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 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.

3.1 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)
```

3.2 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 $*$

3.3 1-dimensional Arrays

A vector

$$x = [12345]$$

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.]])
```

3.4 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.

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.]])
```

3.5 Multidimensional Arrays

Higher dimensional arrays are useful when tracking matrix valued processes through time, such as a timevarying covariance matrices. Multidimensional (N -dimensional) arrays are available forN 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)).

3.6 Concatenation

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

3.7 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.

3.8 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 byNumPy 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.

3.9 Special Arrays

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

3.9.1 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).

3.9.2 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).

3.9.3 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
```

3.10 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).

3.10.1 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).

4 Data Structures

pandas introduces two new data structures to Python - Series and DataFrame, both of which are built on top of NumPy (this means it's fast).

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
pd.set_option('max_columns', 50)
```

4.1 Series

Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the index. The basic method to create a Series is to call:

```
s = Series(data, index=index)
```

Here, data can be many different things:

- a Python dict
- an ndarray
- a scalar value (like 5)

A Series is a one-dimensional object similar to an array, list, or column in a table. It will assign a labeled index to each item in the Series. By default, each item will receive an index label from 0 to N, where N is the length of the Series minus one.

```
# create a Series with an arbitrary list
s = pd.Series([7, 'Heisenberg', 3.14, -1789710578, 'Happy Eating!'])
s
```

```
0                7
1       Heisenberg
2             3.14
3      -1789710578
4    Happy Eating!
dtype: object
```

Alternatively, you can specify an index to use when creating the Series.

```
s = pd.Series([7, 'Heisenberg', 3.14, -1789710578, 'Happy Eating!'],
              index=['A', 'Z', 'C', 'Y', 'E'])
s
```

```
A                7
Z       Heisenberg
C             3.14
Y      -1789710578
E    Happy Eating!
dtype: object
```

The Series constructor can convert a dictonary as well, using the keys of the dictionary as its index.

```
d = {'Chicago': 1000, 'New York': 1300, 'Portland': 900, 'San Francisco': 1100,
     'Austin': 450, 'Boston': None}
cities = pd.Series(d)
cities
Out[4]:
Austin            450
Boston            NaN
Chicago          1000
New York         1300
```

```
Portland          900
San Francisco    1100
dtype: float64
```

You can use the index to select specific items from the Series ...

```
cities['Chicago']
Out[5]:
1000.0
```

```
cities[['Chicago', 'Portland', 'San Francisco']]
Out[6]:
Chicago          1000
Portland          900
San Francisco    1100
dtype: float64
```

Or you can use boolean indexing for selection.

```
cities[cities < 1000]
Out[7]:
Austin      450
Portland    900
dtype: float64
```

That last one might be a little weird, so let's make it more clear - cities ¡ 1000 returns a Series of True/False values, which we then pass to our Series cities, returning the corresponding True items.

```
less_than_1000 = cities < 1000
print less_than_1000
```

```
print '\n'
print cities[less_than_1000]
Austin            True
Boston           False
Chicago          False
New York         False
Portland          True
San Francisco    False
dtype: bool


Austin      450
Portland    900
dtype: float64

You can also change the values in a Series on the fly.

\begin{framed}
\begin{verbatim}
# changing based on the index
print 'Old value:', cities['Chicago']
cities['Chicago'] = 1400
print 'New value:', cities['Chicago']
Old value: 1000.0
New value: 1400.0

\begin{framed}
\begin{verbatim}
# changing values using boolean logic
print cities[cities < 1000]
print '\n'
cities[cities < 1000] = 750

print cities[cities < 1000]
Austin      450
Portland    900
dtype: float64


Austin      750
Portland    750
dtype: float64
```

```
```

What if you aren't sure whether an item is in the Series? You can check using idiomatic Python.

```
print 'Seattle' in cities
print 'San Francisco' in cities
False
True
```

Mathematical operations can be done using scalars and functions.

```
# divide city values by 3
cities / 3
Out[12]:
Austin           250.000000
Boston                  NaN
Chicago          466.666667
New York         433.333333
Portland         250.000000
San Francisco    366.666667
dtype: float64

\begin{framed}
\begin{verbatim}
# square city values
np.square(cities)
Out[13]:
Austin            562500
Boston               NaN
Chicago          1960000
New York         1690000
Portland          562500
San Francisco    1210000
dtype: float64
```

You can add two Series together, which returns a union of the two Series with the addition occurring on the shared index values. Values on either Series that did not have a shared index will produce a NULL/NaN (not a number).

```
print cities[['Chicago', 'New York', 'Portland']]
print'\n'
print cities[['Austin', 'New York']]
print'\n'
print cities[['Chicago', 'New York', 'Portland']] + cities[['Austin', 'New York']]
```

```
Chicago     1400
New York    1300
Portland     750
dtype: float64


Austin       750
New York    1300
dtype: float64


Austin       NaN
Chicago      NaN
New York    2600
Portland     NaN
dtype: float64
```

Notice that because Austin, Chicago, and Portland were not found in both Series, they were returned with NULL/NaN values.

NULL checking can be performed with isnull and notnull.

```
# returns a boolean series indicating which values aren't NULL
cities.notnull()
Out[15]:
Austin            True
Boston           False
Chicago           True
New York          True
Portland          True
San Francisco     True
dtype: bool
In [16]:
```

```
# use boolean logic to grab the NULL cities
print cities.isnull()
print '\n'
print cities[cities.isnull()]
Austin           False
Boston            True
Chicago          False
New York         False
Portland         False
San Francisco    False
dtype: bool


Boston   NaN
dtype: float64
```

4.2 DataFrame

A DataFrame is a tablular data structure comprised of rows and columns, akin to a spreadsheet, database table, or R's data.frame object. You can also think of a DataFrame as a group of Series objects that share an index (the column names).

For the rest of the tutorial, we'll be primarily working with DataFrames.

4.3 Panel

`Panel` is a somewhat less-used, but still important container for 3-dimensional data. The term panel data is derived from econometrics and is partially responsible for the name pandas: pan(el)-da(ta)-s. The names for the 3 axes are intended to give some semantic meaning to describing operations involving panel data and, in particular, econometric analysis of panel data. However, for the strict purposes of slicing and dicing a collection of DataFrame objects, you may find the axis names slightly arbitrary:

- items: axis 0, each item corresponds to a DataFrame contained inside
- major_axis: axis 1, it is the index (rows) of each of the DataFrames
- minor_axis: axis 2, it is the columns of each of the DataFrames

5 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.

5.1 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.

5.1.1 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])
```

5.1.2 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).

5.1.3 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)

5.2 Shape Information and Transformation

5.2.1 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)
```

5.2.2 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.

5.2.3 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
```

5.2.4 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
```

5.2.5 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.

For example, suppose X was an array MATRIX HERE and MATRIX HERE was required. This could be accomplished by manually constructing y using hstack and vstack. an attribute .

```
>>> 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.

5.2.6 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])
```

5.2.7 flatten

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

5.2.8 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]])
```

5.2.9 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]])
```

5.2.10 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])
```

5.2.11 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)
```

5.2.12 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]])
```

5.2.13 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]])
```

5.2.14 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]])
```

5.3 Some Useful Linear Algebra Functions

5.3.1 det

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

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

5.3.2 solve

`solve` solves the system $\boldsymbol{Ax}$=$\boldsymbol{b}$ 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]])
```

5.3.3 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.

6 Logical Operators

Logical expressions can be combined using four logical devices,

```
% BEGIN TABLE
Keyword (Scalar) & Function & Bitwise & True if . . . \\ \hline
and & logical_and & Both & True \\ \hline
or  & logical_or & Either or Both True \\ \hline
not & logical_not & ~ & Not True \\ \hline
& logical_xor & ^ & One True and One False \\ \hline

% END OF TABLE
```

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.

6.1 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)
```

6.2 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)
```

6.2.1 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).

6.2.2 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.

6.2.3 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
```

6.3 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
```