Chapter 9

Importing and Exporting Data

9.1 Importing Data using pandas

Pandas is an increasingly important component of the Python scientific stack, and a complete discussion of its main features is included in Chapter 17. All of the data readers in pandas load data into a pandas DataFrame (see Section 17.1.2), and so these examples all make use of the `values` property to extract a NumPy array. In practice, the DataFrame is much more useful since it includes useful information such as column names read from the data source. In addition to the three formats presented here, pandas can also read json, SQL, html tables or from the clipboard, which is particularly useful for interactive work since virtually any source that can be copied to the clipboard can be imported.

9.1.1 CSV and other formatted text files

Comma-separated value (CSV) files can be read using `read_csv`. When the CSV file contains mixed data, the default behavior will read the file into an array with an `object` data type, and so further processing is usually required to extract the individual series.

```
>>> from pandas import read_csv
>>> csv_data = read_csv('FTSE_1984_2012.csv')
>>> csv_data = csv_data.values
>>> csv_data[:4]
array([['2012-02-15', 5899.9, 5923.8, 5880.6, 5892.2, 801550000L, 5892.2],
       ['2012-02-14', 5905.7, 5920.6, 5877.2, 5899.9, 832567200L, 5899.9],
       ['2012-02-13', 5852.4, 5920.1, 5852.4, 5905.7, 643543000L, 5905.7],
       ['2012-02-10', 5895.5, 5895.5, 5839.9, 5852.4, 948790200L, 5852.4]], dtype=object)

>>> open = csv_data[:,1]
```

When the entire file is numeric, the data will be stored as a homogeneous array using one of the numeric data types, typically float64. In this example, the first column contains Excel dates as numbers, which are the number of days past January 1, 1900.

```
>>> csv_data = read_csv('FTSE_1984_2012_numeric.csv')
>>> csv_data = csv_data.values
>>> csv_data[:4,:2]
array([[ 40954. ,   5899.9],
       [ 40953. ,   5905.7],
```

```
       [ 40952. ,   5852.4],
       [ 40949. ,   5895.5]])
```

9.1.2 Excel files

Excel files, both 97/2003 (xls) and 2007/10/13 (xlsx), can be imported using `read_excel`. Two inputs are required to use `read_excel`, the filename and the sheet name containing the data. In this example, pandas makes use of the information in the Excel workbook that the first column contains dates and converts these to `datetimes`. Like the mixed CSV data, the array returned has `object` data type.

```
>>> from pandas import read_excel
>>> excel_data = read_excel('FTSE_1984_2012.xls','FTSE_1984_2012')
>>> excel_data = excel_data.values
>>> excel_data[:4,:2]
array([[datetime.datetime(2012, 2, 15, 0, 0), 5899.9],
       [datetime.datetime(2012, 2, 14, 0, 0), 5905.7],
       [datetime.datetime(2012, 2, 13, 0, 0), 5852.4],
       [datetime.datetime(2012, 2, 10, 0, 0), 5895.5]], dtype=object)

>>> open = excel_data[:,1]
```

9.1.3 STATA files

Pandas also contains a method to read STATA files.

```
>>> from pandas import read_stata
>>> stata_data = read_stata('FTSE_1984_2012.dta')
>>> stata_data = stata_data.values
>>> stata_data[:4,:2]
array([[  0.00000000e+00,   4.09540000e+04],
       [  1.00000000e+00,   4.09530000e+04],
       [  2.00000000e+00,   4.09520000e+04],
       [  3.00000000e+00,   4.09490000e+04]])
```

9.2 Importing Data without pandas

Importing data without pandas ranges from easy when files contain only numbers to difficult, depending on the data size and format. A few principles can simplify this task:

- The file imported should contain numbers *only*, with the exception of the first row which may contain the variable names.
- Use another program, such as Microsoft Excel, to manipulate data before importing.
- Each column of the spreadsheet should contain a single variable.
- Dates should be converted to YYYYMMDD, a numeric format, before importing. This can be done in Excel using the formula:
  `=10000*YEAR(A1)+100*MONTH(A1)+DAY(A1)+(A1-FLOOR(A1,1))`

- Store times separately from dates using a numeric format such as seconds past midnight or HHmmSS.sss.

9.2.1 CSV and other formatted text files

A number of importers are available for regular (e.g. all rows have the same number of columns) comma-separated value (CSV) data. The choice of which importer to use depends on the complexity and size of the file. Purely numeric files are the simplest to import, although most files which have a repeated structure can be directly imported (unless they are very large).

loadtxt

`loadtxt` is a simple, fast text importer. The basic use is `loadtxt(`*filename*`)`, which will attempt to load the data in file name as floats. Other useful named arguments include `delim`, which allow the file delimiter to be specified, and `skiprows` which allows one or more rows to be skipped.
`loadtxt` requires the data to be numeric and so is only useful for the simplest files.

```
>>> data = loadtxt('FTSE_1984_2012.csv',delimiter=',') # Error
ValueError: could not convert string to float: Date

# Fails since CSV has a header
>>> data = loadtxt('FTSE_1984_2012_numeric.csv',delimiter=',') # Error
ValueError: could not convert string to float: Date

>>> data = loadtxt('FTSE_1984_2012_numeric.csv',delimiter=',',skiprows=1)
>>> data[0]
array([ 4.09540000e+04, 5.89990000e+03, 5.92380000e+03, 5.88060000e+03, 5.89220000e+03,
8.01550000e+08, 5.89220000e+03])
```

genfromtxt

`genfromtxt` is a slightly slower, more robust importer. `genfromtxt` is called using the same syntax as `loadtxt`, but will not fail if a non-numeric type is encountered. Instead, `genfromtxt` will return a NaN (not-a-number) for fields in the file it cannot read.

```
>>> data = genfromtxt('FTSE_1984_2012.csv',delimiter=',')
>>> data[0]
array([ nan,  nan,  nan,  nan,  nan,  nan,  nan])
>>> data[1]
array([ nan, 5.89990000e+03, 5.92380000e+03, 5.88060000e+03, 5.89220000e+03, 8.01550000e+08,
5.89220000e+03])
```

Tab delimited data can be read in a similar manner using `delimiter='\t'`.

```
>>> data = genfromtxt('FTSE_1984_2012_numeric_tab.txt',delimiter='\t')
```

csv2rec

csv2rec is an even more robust – and slower – CSV importer which imports non-numeric data such as dates. It attempts to find the best data type for each column. Note that when pandas is available, read_csv is a better option than csv2rec.

```
>>> data = csv2rec('FTSE_1984_2012.csv',delimiter=',')
>>> data[0]
(datetime.date(2012, 2, 15), 5899.9, 5923.8, 5880.6, 5892.2, 801550000L, 5892.2)
```

Unlike loadtxt and genfromtxt, which both return an array, csv2rec returns a record array (see Chapter 16) which is, in many ways, like a list. csv2rec converted each row of the input file into a datetime (see Chapter 14), followed by 4 floats for open, high, low and close, then a long integer for volume, and finally a float for the adjusted close. When the data contain non-numeric values, returned array is not homogeneous, and so it is necessary to create an array to store the numeric content of the imported data.

```
>>> open = data['open']
>>> open
array([ 5899.9,  5905.7,  5852.4, ...,  1095.4,  1095.4,  1108.1])
```

9.2.2 Excel Files

xlrd

Reading Excel files in Python is more involved, and it is simpler to convert the xls to CSV. Excel files can be read using xlrd (which is part of xlutils).

```
from __future__ import print_function
import xlrd

wb = xlrd.open_workbook('FTSE_1984_2012.xls')
# To read xlsx change the filename
# wb = xlrd.open_workbook('FTSE_1984_2012.xlsx')
sheetNames = wb.sheet_names()
# Assumes 1 sheet name
sheet = wb.sheet_by_name(sheetNames[0])
excelData = [] # List to hold data
for i in xrange(sheet.nrows):
    excelData.append(sheet.row_values(i))

# Subtract 1 since excelData has the header row
open = empty(len(excelData) - 1)
for i in xrange(len(excelData) - 1):
    open[i] = excelData[i+1][1]
```

The listing does a few things. First, it opens the workbook for reading (open_workbook('FTSE_1984_2012.xls')), then it gets the sheet names (wb.sheet_names()) and opens a sheet (wb.sheet_by_name(sheetNames[0])). From the sheet, it gets the number of rows (sheet.nrows), and fills a list with the values, row-by-row. Once the data has been read-in, the final block fills an array with the opening prices. This is substantially more complicated than importing from a CSV file, although reading Excel files is useful for automated work (e.g. you have no choice but to import from an Excel file since it is produced by some other software).

openpyxl

openpyxl reads and write the modern Excel file format that is the default in Office 2007 or later. openpyxl also supports a reader and writer which is optimized for large files, a feature not available in xlrd. Unfortunately, openpyxl uses a different syntax from xlrd, and so some modifications are required when using openpyxl.

```
from __future__ import print_function
import openpyxl

wb = openpyxl.load_workbook('FTSE_1984_2012.xlsx')
sheetNames = wb.get_sheet_names()
# Assumes 1 sheet name
sheet = wb.get_sheet_by_name(sheetNames[0])
rows = sheet.rows

# Subtract 1 since excelData has the header row
open = empty(len(rows) - 1)
for i in xrange(len(excelData) - 1):
    open[i] = rows[i+1][1].value
```

The strategy with 2007/10/13 xlsx files is essentially the same as with 97/2003 files. The main difference is that the command `sheet.rows()` returns a tuple containing the all of the rows in the selected sheet. Each row is itself a tuple which contains `Cells` (which are a type created by openpyxl), and each cell has a `value` (Cells also have other useful attributes such as `data_type` and methods such as `is_date()`) .

Using the optimized reader is similar. The primary differences are:

- The workbook must be opened using the keyword argument `use_iterators = True`
- The rows are sequentially accessible using `iter_rows()`.
- `value` is not available, and so `internal_value` must be used.
- The number of rows is not known, and so it isn't possible to pre-allocate the storage variable with the correct number of rows.

```
from __future__ import print_function
import openpyxl

wb = openpyxl.load_workbook('FTSE_1984_2012.xlsx', use_iterators = True)
sheetNames = wb.get_sheet_names()
# Assumes 1 sheet name
sheet = wb.get_sheet_by_name(sheetNames[0])

# Use list to store data
open = []

# Changes since access is via memory efficient iterator
# Note () on iter_rows
for row in sheet.iter_rows():
    # Must use internal_value
```

```
    open.append(row[1].internal_value)

# Burn first row and convert to array
open = array(open[1:])
```

9.2.3 MATLAB Data Files (.mat)

SciPy enables MATLAB data files (mat files) to be read excluding except the latest V7.3 format, which can be read using PyTables or h5py. Data from compatible mat files can be loaded using `loadmat`. The data is loaded into a dictionary, and individual variables are accessed using the keys of the dictionary.

```
>>> import scipy.io as sio
>>> matData = sio.loadmat('FTSE_1984_2012.mat')
>>> type(matData)
dict

>>> matData.keys()
['volume',
 '__header__',
 '__globals__',
 'high',
 'adjclose',
 'low',
 'close',
 '__version__',
 'open']

>>> open = matData['open']
```

MATLAB data files in the newest V7.3 format can be easily read using PyTables.

```
>>> import tables
>>> matfile = tables.openFile('FTSE_1984_2012_v73.mat')
>>> matfile.root
/ (RootGroup) ''
  children := ['volume' (CArray), 'high' (CArray), 'adjclose' (CArray), 'low' (CArray), '
      close' (CArray), 'open' (CArray)]

>>> matfile.root.open
/open (CArray(1, 7042), zlib(3)) ''
  atom := Float64Atom(shape=(), dflt=0.0)
  maindim := 0
  flavor := 'numpy'
  byteorder := 'little'
  chunkshape := (1, 7042)

>>> open = matfile.root.open.read()
open = matfile.root.open.read()

>>> matfile.close() # Close the file
```

9.2.4 Reading Complex Files

Python can be programmed to read any text file format since it contains functions for directly accessing files and parsing strings. Reading poorly formatted data files is an advanced technique and should be avoided if possible. However, some data is only available in formats where reading in data line-by-line is the only option. For example, the standard import methods fail if the raw data is very large (too large for Excel) and is poorly formatted. In this case, the only possibility may be to write a program to read the file line-by-line (or in blocks) and to directly process the raw text.

The file *IBM_TAQ.txt* contains a simple example of data that is difficult to import. This file was downloaded from Wharton Research Data Services and contains all prices for IBM from the TAQ database between January 1, 2001 and January 31, 2001. It is too large to use in Excel and has both numbers, dates and text on each line. The following code block shows one method for importing this data set.

```
import io
from numpy import array

f = io.open('IBM_TAQ.txt', 'r')
line = f.readline()
# Burn the first list as a header
line = f.readline()

date = []
time = []
price = []
volume = []
while line:
    data = line.split(',')
    date.append(int(data[1]))
    price.append(float(data[3]))
    volume.append(int(data[4]))
    t = data[2]
    time.append(int(t.replace(':','')))
    line = f.readline()

# Convert to arrays, which are more useful than lists
# for numeric data
date = array(date)
price = array(price)
volume = array(volume)
time = array(time)

allData = array([date,price,volume,time])

f.close()
```

This block of code does a few thing:

- Open the file directly using `file`
- Reads the file line by line using `readline`

- Initializes lists for all of the data
- Rereads the file parsing each line by the location of the commas using `split(',')` to split the line at each comma into a list
- Uses `replace(':','')` to remove colons from the times
- Uses `int()` and `float()` to convert strings to numbers
- Closes the file directly using `close()`

9.3 Saving or Exporting Data using pandas

Pandas supports writing to CSV, general delimited text files, Excel files, json, html tables, HDF5 and STATA. An understanding of the pandas' DataFrame is required prior to using pandas file writing facilities, and Chapter 17 provides further information.

9.4 Saving or Exporting Data without pandas

Native NumPy Format

A number of options are available for saving data. These include using native npz data files, MATLAB data files, CSV or plain text. Multiple numpy arrays can be saved using `savez_compressed` (`numpy.savez_compressed`).

```
x = arange(10)
y = zeros((100,100))
savez_compressed('test',x,y)
data = load('test.npz')
# If no name is given, arrays are generic names arr_1, arr_2, etc
x = data['arr_1']

savez_compressed('test',x=x,otherData=y)
data = load('test.npz')
# x=x provides the name x for the data in x
x = data['x']
# otherDate = y saves the data in y as otherData
y = data['otherData']
```

A version which does not compress data but is otherwise identical is `savez`. Compression is usually a good idea and is very helpful for storing arrays which have repeated values and are large.

9.4.1 Writing MATLAB Data Files (.mat)

SciPy enables MATLAB data files to be written. Data can be written using `savemat`, which takes two inputs, a file name and a dictionary containing data, in its simplest form.

```
from __future__ import print_function
import scipy.io as sio
```

```
x = array([1.0,2.0,3.0])
y = zeros((10,10))
# Set up the dictionary
saveData = {'x':x, 'y':y}
sio.savemat('test',saveData,do_compression=True)
# Read the data back in
matData = sio.loadmat('test.mat')
```

`savemat` uses the optional argument `do_compression = True`, which compresses the data, and is generally a good idea on modern computers and/or for large datasets.

9.4.2 Exporting Data to Text Files

Data can be exported to a tab-delimited text files using `savetxt`. By default, `savetxt` produces tab delimited files, although then can be changed using the names argument `delimiter`.

```
x = randn(10,10)
# Save using tabs
savetxt('tabs.txt',x)
# Save to CSV
savetxt('commas.csv',x,delimiter=',')
# Reread the data
xData = loadtxt('commas.csv',delimiter=',')
```

9.5 Exercises

Note: There are no exercises using pandas in this chapter. For exercises using pandas to read or write data, see Chapter 17.

1. The file *exercise3.xls* contains three columns of data, the date, the return on the S&P 500, and the return on XOM (ExxonMobil). Using Excel, convert the date to YYYYMMDD format and save the file.

2. Save the file as both CSV and tab delimited. Use the three text readers to read the file, and compare the arrays returned.

3. Parse loaded data into three variables, `dates`, `SP500` and `XOM`.

4. Save NumPy, compressed NumPy and MATLAB data files with all three variables. Which files is the smallest?

5. Construct a new variable, `sumreturns` as the sum of `SP500` and `XOM`. Create another new variable, `outputdata` as a horizontal concatenation of `dates` and `sumreturns`.

6. Export the variable `outputdata` to a new CSV file using `savetxt`.

7. (Difficult) Read in *exercise3.xls* directly using xlrd.

8. (Difficult) Save *exercise3.xls* as *exercise3.xlsx* and read in directly using openpyxl.