An\_Introduction\_Statistics\_Python\_c03

p59

**Chapter 3 Data Input**

This chapter shows how to read data into *Python*. Thus it forms the link between the chapter on *Python*, and the first chapter on statistical data analysis. It may be surprising, but reading data into the system in the correct format and checking for erroneous or missing entries is often one of the most time consuming parts of the data analysis.

Data input can be complicated by a number of problems, like different separators between data entries (such as spaces and/or tabulators), or empty lines at the end of the file. In addition, data may have been saved in different formats, such as MS Excel, *Matlab*, HDF5 (which also includes the *Matlab*-format), or in databases. Understandably, we cannot cover all possible input options. But I will try to give an overview of where and how to start with data input.

**3.1 Input from Text Files**

***3.1.1 Visual Inspection***

When the data are available as ASCII-files, you should always start out with a visual inspection of the data! In particular, you should check

* Do the data have a header and/or a footer?
* Are there empty lines at the end of the file?
* Are there white-spaces before the first number, or at the end of each line? (The  
   latter is a lot harder to see.)
* Are the data separated by tabulators, and/or by spaces? (Tip: you should use a  
   text-editor which can visualize tabs, spaces, and end-of-line (EOL) characters.)

***3.1.2 Reading ASCII-Data into Python***

In *Python*, I strongly suggest that you start out reading in and inspecting your data in the *Jupyter QtConsole* or in an *Jupyter Notebook*. It allows you to move around much more easily, try things out, and quickly get feedback on how successful your commands have been. When you have your command syntax worked out, you can obtain the command history with %history, copy it into your favorite IDE, and turn it into a program.

While the a numpy command np.loadtxt allows to read in simply formatted text data, most of the time I go straight to *pandas*, as it provides significantly more powerful tools for data-entry. A typical workflow can contain the following steps:

* Changing to the folder where the data are stored.
* Listing the items in that folder.
* Selecting one of these items, and reading in the corresponding data.
* Checking if the data have been read in completely, and in the correct format.  
     
  These steps can be implemented in *IPython* with the following commands:  
     
  In [1]: import pandas as pd  
   In [2]: cd 'C:\Data\storage'  
   In [3]: pwd # Check if you were successful  
   In [4]: ls # List the files in that directory In [5]: inFile = 'data.txt'  
   In [6]: df = pd.read\_csv(inFile)  
   In [7]: df.head() # Check if first line is ok In [8]: df.tail() # Check the last line

After “In [7]” I often have to adjust the options of pd.read\_csv, to read in all the data correctly. Make sure you check that the number of column headers is equal to the number of columns that you expect. It can happen that everything gets read in—but into one large single column!

**a) Simple Text-Files**

For example, a file data.txt with the following content

1, 1.3, 0.6

2, 2.1, 0.7

3, 4.8, 0.8

4, 3.3, 0.9

can be read in and displayed with

In [9]: data = np.loadtxt('data.txt', delimiter=',')

In [10]: data Out[10]:

array([[ 1. , 1.3, 0.6], [ 2. , 2.1, 0.7], [ 3. , 4.8, 0.8], [ 4. , 3.3, 0.9]])

where data is a *numpy array*. Without the flag delimiter=',', the function *np.loadtxt* crashes. An alternative way to read in these data is with

In [11]: df = pd.read\_csv('data.txt', header=None)

In [12]: df Out[12]:

012 0 1 1.3 0.6 1 2 2.1 0.7 2 3 4.8 0.8 3 4 3.3 0.9

where df is a *pandas DataFrame*. Without the flag header=None, the entries of the first row are falsely interpreted as the column labels!

In [13]: df = pd.read\_csv('data.txt')

In [14]: df

Out[14]:

1 1.3 0.6

0 2 2.1 0.7

1 3 4.8 0.8

2 4 3.3 0.9

The *pandas* routine has the advantage that the first column is recognized as integer, whereas the second and third columns are float.

**b) More Complex Text-Files**

The advantage of using *pandas* for data input becomes clear with more complex files. Take for example the input file “data2.txt,” containing the following lines:

ID, Weight, Value 1, 1.3, 0.6

2, 2.1, 0.7

3, 4.8, 0.8

4, 3.3, 0.9

Those are dummy values, created by ThH. June, 2015

One of the input flags of pd.read\_csv is skipfooter, so we can read in the data easily with

In [15]: **df2 = pd.read\_csv('data.txt', skipfooter=3, delimiter='[ ,]\*')**

The last option, delimiter='[ ,]\*', is a *regular expression* (see below) specifying that “one or more spaces and/or commas may be used to separate entry values.” Also, when the input file includes a header row with the column names, the data can be accessed immediately with their corresponding column name, e.g.:

In [16]: df2 Out[16]:

ID Weight Value 0 1 1.3 0.6 1 2 2.1 0.7 2 3 4.8 0.8 3 4 3.3 0.9

In [17]:

Out[17]:

0 0.6

1 0.7

2 0.8

3 0.9

Name: Value, dtype: float64

**c) Regular Expressions**

Working with text data often requires the use of simple *regular expressions*. Regular expressions are a very powerful way of finding and/or manipulating text strings. Many books have been written about them, and good, concise information on regular expressions can be found on the web, for example at:

* https://www.debuggex.com/cheatsheet/regex/python provides a convenient cheat sheet for regular expressions in *Python*.
* http://www.regular-expressions.info gives a comprehensive description of regu- lar expressions.

Let me give two examples how *pandas* can make use of regular expressions:

1. Reading in data from a file, separated by a combination of commas, semicolons, or white-spaces:

**df = pd.read\_csv(inFile, sep='[ ;,]+')**

The square brackets indicate a *combination* (“[: : :]”) of : : : The plus indicates *one or more* (“+”)

2. Extracting columns with certain name-patterns from a *pandas* DataFrame. In the following example, I will extract all the columns starting with Vel: In [18]: data = np.round(np.random.randn(100,7), 2)

In [19]: df = pd.DataFrame(data, columns=['Time',

'PosX', 'PosY', 'PosZ', 'VelX', 'VelY', 'VelZ'])

In [20]: df.head()

Out[20]:

Time PosX PosY PosZ VelX VelY VelZ 0 0.30 -0.13 1.42 0.45 0.42 -0.64 -0.86 1 0.17 1.36 -0.92 -1.81 -0.45 -1.00 -0.19 2 -3.03 -0.55 1.82 0.28 0.29 0.44 1.89 3 -1.06 -0.94 -0.95 0.77 -0.10 -1.58 1.50 4 0.74 -1.81 1.23 1.82 0.45 -0.16 0.12

In [21]: vel = df.filter(regex='Vel\*')

In [22]: vel.head()

Out[22]:

VelX VelY VelZ 0 0.42 -0.64 -0.86 1 -0.45 -1.00 -0.19 2 0.29 0.44 1.89 3 -0.10 -1.58 1.50 4 0.45 -0.16 0.12

**3.2 Input from MS Excel**

There are two approaches to reading a *Microsoft Excel* file in *pandas*: the function read\_excel, and the class ExcelFile.1

* read\_excel is for reading one file with file-specific arguments (i.e., identical data formats across sheets).
* ExcelFile is for reading one file with sheet-specific arguments (i.e., different data formats across sheets).

Choosing the approach is largely a question of code readability and execution speed.

The following commands show equivalent class and function approaches to read a single sheet:

# using the ExcelFile class

xls = pd.ExcelFile('path\_to\_file.xls') data = xls.parse('Sheet1', index\_col=None,

na\_values=['NA'])

# using the read\_excel function

data = pd.read\_excel('path\_to\_file.xls', 'Sheet1',

index\_col=None, na\_values=['NA'])

If this fails, give it a try with the *Python* package *xlrd*.

The following advanced script shows how to directly import data from an Excel file which is stored in a zipped archive on the web:

**Listing 3.1 L3\_2\_readZip.py**

1 '''GetdatafromMS-Excelfiles,whicharestoredzippedon

the WWW. '''

2

3 #author:ThomasHaslwanter,date:Nov-2015 4

* 5 #Importstandardpackages
* 6 importpandasaspd

7

* 8 #additionalpackages
* 9 importio
* 10 importzipfile

11

* 12 #Python2/3usedifferentpackagesfor"urlopen"
* 13 importsys
* 14 ifsys.version\_info[0]==3:
* 15 from urllib.request import urlopen
* 16 else:
* 17 from urllib import urlopen

18

* 19 defgetDataDobson(url,inFile):
* 20 '''Extract data from a zipped-archive on the web'''

21

* 22 # get the zip-archive
* 23 GLM\_archive = urlopen(url).read()

24

* 25 # make the archive available as a byte-stream
* 26 zipdata = io.BytesIO()
* 27 zipdata.write(GLM\_archive)

28

* 29 # extract the requested file from the archive, as a  
     
  pandas XLS-file
* 30 myzipfile = zipfile.ZipFile(zipdata)
* 31 xlsfile = myzipfile.open(inFile)

32

33 # read the xls-file into Python, using Pandas, and return

the extracted data

* 34 xls = pd.ExcelFile(xlsfile)
* 35 df = xls.parse('Sheet1', skiprows=2)

36

37 return df 38

* 39 if\_\_name\_\_=='\_\_main\_\_':
* 40 # Select archive (on the web) and the file in the archive
* 41 url = 'http://cdn.crcpress.com/downloads/C9500/GLM\_data.  
     
  zip'
* 42 inFile = r'GLM\_data/Table 2.8 Waist loss.xls'

43

* 44 df = getDataDobson(url, inFile)
* 45 print(df)

46

47 input('All done!')

**3.3 Input from Other Formats**

**Matlab** Support for data input from *Matlab* files is built into *scipy*, with the command scipy.io.loadmat.

**Clipboard** If you have data in your clipboard, you can import them directly with pd.read\_clipboard()

**Other file formats** Also SQL databases and a number of additional formats are supported by *pandas*. The simplest way to access them is typing pd.read\_ + TAB, which shows all currently available options for reading data into *pandas* DataFrames.

***3.3.1 Matlab***

The following commands return string, number, vector, and matrix variables from a *Matlab* file “data.mat”, as well as the content of a structure with two entries (a vector and a string). The *Matlab* variables containing the scalar, string, vector, matrix, and structure are called *number, text, vector, matrix*, and *structure*, respectively.

from scipy.io import loadmat

data =

loadmat('data.mat')

= data['number'][0,0] = data['text'][0]

= data['vector'][0] = data['matrix']

number

text

vector

matrix

struct\_values = data['structure'][0,0][0][0] strunct\_string = data['structure'][0,0][1][0]