STATISTICS USING PYTHON (16CS353)

UNIT-1

Why Statistics?

Every day we are confronted with situations with uncertain outcomes, and must make decisions based on incomplete data: “Should I run for the bus? Which stock should I buy? Which man should I marry? Should I take this medication? Should I have my children vaccinated?” Some of these questions are beyond the realm of statistics (“Which person should I marry?”), because they involve too many unknown variables. But in many situations, statistics can help extract maximum knowledge from information given, and clearly spell out what we know and what we don’t know. For example, it can turn a vague statement like “This medication may cause nausea,” or “You could die if you don’t take this medication” into a specific statement like “Three patients in one thousand experience nausea when taking this medication,” or “If you don’t take this medication, there is a 95% chance that you will die.”

Without statistics, the interpretation of data can quickly become massively flawed. Take, for example, the estimated number of German tanks produced during World War II, also known as the “German Tank Problem.” The estimate of the number of German tanks produced per month from standard intelligence data was *1,550*; however, the statistical estimate based on the number of tanks observed was *327*, which was very close to the actual production number of *342* (http://en.wikipedia.org/wiki/German_tank_problem).

Similarly, using the wrong tests can also lead to erroneous results.

In general, statistics will help to

- Clarify the question.
- Identify the variable and the measure of that variable that will answer that question.
- Determine the required sample size.
- Describe variation.
- Make quantitative statements about estimated parameters.
- Make predictions based on your data.
-

Reading the Book Statistics was originally invented—like so many other things— by the famous mathematician C.F. Gauss, who said about his own work, “Ich habe fleissig sein müssen; wer es gleichfalls ist, wird eben so weit kommen.” (“I had to work hard; if you work hard as well, you, too, will be successful.”). Just as reading a book about playing the piano won’t turn you into a great pianist, simply reading this book will not teach you statistical data analysis. If you don’t have your own data to analyze, you need to do the exercises included. Should you become frustrated or stuck, you can always check the sample Solutions provided at the end of the book.

Exercises Solutions to the exercises provided can be found at the end of the book. In my experience, very few people work through large numbers of examples on their own, so I have not included additional exercises in this book.

If the information here is not sufficient, additional material can be found in other statistical textbooks and on the web:

Books There are a number of good books on statistics. My favorite is Altman (1999): it does not dwell on computers and modeling, but gives an extremely useful introduction to the field, especially for life sciences and medical applications. Many formulations and examples in this manuscript have been taken from that book.

A more modern book, which is more voluminous and, in my opinion, a bit harder to read, is Riffenburgh (2012). Kaplan (2009) provides a simple introduction to modern regression modeling. If you know your basic statistics, a very good introduction to Generalized Linear Models can be found in Dobson and Barnett (2008), which provides a sound, advanced treatment of statistical modeling.

WWW In the web, you will find very extensive information on statistics in English at
• http://www.statsref.com/
• http://www.vassarstats.net/
• http://www.biostathandbook.com/
• http://onlinestatbook.com/2/index.html
• http://www.itl.nist.gov/div898/handbook/index.htm

A good German web page on statistics and regulatory issues is http://www.reiter1.com/.
I hope to convince you that *Python* provides clear and flexible tools for most of the statistical problems that you will encounter, and that you will enjoy using it.

Python Packages for Statistics

The *Python* core distribution contains only the essential features of a general programming language. For example, it does not even contain a specialized module for working efficiently with vectors and matrices! These specialized modules are being developed by dedicated volunteers. The relationship of the most important *Python* packages for statistical applications is delineated in Fig. 2.1.

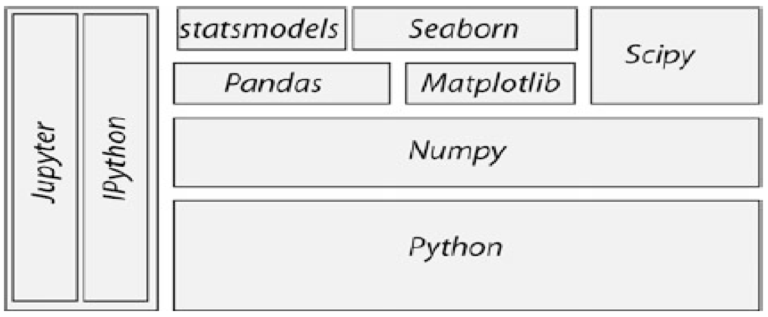


Fig. 2.1 The structure of the most important *Python* packages for statistical applications

To facilitate the use of *Python*, the so-called *Python distributions* collect matching versions of the most important packages, and I strongly recommend using one of these distributions when getting started. Otherwise one can easily become overwhelmed by the huge number of *Python* packages available. My favorite *Python* distributions are

• *WinPython* recommended for Windows users. At the time of writing, the latest version was 3.5.1.3 (newer versions also ok).
https://winpython.github.io/

• *Anaconda* by Continuum. For Windows, Mac, and Linux. Can be used to install Python 2.x and 3.x, even simultaneously! The latest *Anaconda* version at time of writing was 4.0.0 (newer versions also ok).
https://store.continuum.io/cshop/anaconda/

Neither of these two distributions requires administrator rights. I am presently using *WinPython*, which is free and customizable. *Anaconda* has become very popular recently, and is free for educational purposes.

Unless you have a specific requirement for 64-bit versions, you may want to install a 32-bit version of *Python*: it facilitates many activities that require compilation of module parts, e.g., for Bayesian statistics (PyMC), or when you want to speed up your programs with *Cython*. Since all the *Python* packages required for this course are now available for *Python* 3.x, I will use *Python* 3 for this book.

However, all the scripts included should also work for *Python* 2.7. Make sure that you use a current version of *IPython/Jupyter* (4.x), since the *Jupyter Notebooks* provided with this book won't run on *IPython* 2.x.[1]

The programs included in this book have been tested with Python 2.7.10 and 3.5.1, underWindows and Linux, using the following package versions:

- *ipython 4.1.2* : : : For interactive work.
- *numpy 1.11.0* : : : For working with vectors and arrays.
- *scipy 0.17.1* : : : All the essential scientific algorithms, including those for basic
- statistics.
- *matplotlib 1.5.1* : : : The de-facto standard module for plotting and visualization.
- *pandas 0.18.0* : : : Adds *DataFrames* (imagine powerful spreadsheets) to *Python*.
- *patsy 0.4.1* : : : For working with statistical formulas.
- *statsmodels 0.8.0* : : : For statistical modeling and advanced analysis.
- *seaborn 0.7.0* : : : For visualization of statistical data.
- In addition to these fairly general packages, some specialized packages have also
- been used in the examples accompanying this book:
- *xlrd 0.9.4* : : : For reading and writing MS Excel files.
- *PyMC 2.3.6* : : : For Bayesian statistics, including Markov chain Monte Carlo
- simulations.

- *scikit-learn 0.17.1* : : : For machine learning.
- *scikits.bootstrap 0.3.2* : : : Provides bootstrap confidence interval algorithms for
- scipy.
- *lifelines 0.9.1.0* : : : Survival analysis in *Python*.
- *rpy2 2.7.4* : : : Provides a wrapper for *R*-functions in *Python*.

Most of these packages come either with the *WinPython* or *Anaconda* distributions, or can be installed easily using pip or conda. To get *PyMC* to run, you may need to install a C-compiler. On my *Windows* platform, I installed *Visual Studio 15*, and set the environment variable SET VS90COMNTOOLS=%VS14COMNTOOLS%.

To use *R*-function from within *Python*, you also have to install *R*. Like *Python*, *R* is available for free, and can be downloaded from the *Comprehensive R Archive Network*, the latest release at the time of writing being *R-3.3.0* (http://cran.r-project.org/).

First Python Programs

a) HelloWorld

Python Shell

Python is an interpreted language. The simplest way to start *Python* is to type python on the command line. (When I say *command line* I refer in *Windows* to the command shell started with cmd, and in *Linux* or *Mac OS X* to the terminal.) Then you can already start to execute *Python* commands, e.g., the command to print “HelloWorld” to the screen: print('Hello World'). On myWindows computer, this results in

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:54:25) [
MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.

>>> print('Hello World')
Hello World
>>>
```

However, I never use the basic *Python* shell any more, but always start out with the *IPython/Jupyter qtconsole* described in more detail in Sect. 2.3. The *Qt console* is an interactive programming environment which offers a number of advantages. For example, when you type print(in the *Qt console*, you immediately see information about the possible input arguments for the command print.

Python Modules

Often we want to store our commands in a file for later reuse. *Python* files have the extension .py, and are referred to as *Python modules*. Let us create a new file with the name helloWorld.py, containing the line print('Hello World')

This file can now be executed by typing python helloWorld.py on the command line. In *Windows* you can actually run the file by double-clicking it, or by simply typing helloWorld.py if the extension .py is associated with the *Python* program installed on your computer. In *Linux* and *Mac OS X* the procedure is slightly more involved. There, the file needs to contain an additional first line specifying the path to the *Python* installation.

```
#! \usr\bin\python
print('Hello World')
```

On these two systems, you also have to make the file executable, by typing chmod +x helloWorld.py, before you can run it with helloWorld.py.

b) SquareMe

To increase the level of complexity, let us write a *Python* module which prints out the square of the numbers from zero to five. We call the file squareMe.py, and it contains the following lines
Listing 2.1 squareMe.py

```
 # This file shows the square of the numbers from 0 to 5.

def squared(x):
return x**2
for ii in range(6):
print(ii, squared(ii))
print('Done')
```

Let me explain what happens in this file, line-by-line:

1 The first line starts with “#”, indicating a comment-line.

3–4 These two lines define the function *squared*, which takes the variable *x* as input, and returns the square (x**2) of this variable.

Note: The range of the function is defined by the indentation! This is a feature loved by many *Python* programmers, but often found confusing by newcomers. Here the last indented line is *line 4*, which ends the function definition.

6–7 Here the program loops over the first 6 numbers. Also the range of the for loop is defined by the indentation of the code.

In *line 7*, each number and its corresponding square are printed to the output.

9 This command is not indented, and therefore is executed after the for-loop has ended.

Notes

- Since*Python* starts at 0, the loop in *line 6* includes the numbers from 0 to 5.

- ➢ In contrast to some other languages *Python* distinguishes the syntax for function calls from the syntax for addressing elements of an array etc: function calls, as in *line 7*, are indicated with round brackets (...); and individual elements of arrays or vectors are addressed by square brackets [...].

Pandas: Data Structures for Statistics

pandas is a widely used *Python* package which has been contributed by Wes McKinney. It provides data structures suitable for statistical analysis, and adds functions that facilitate data input, data organization, and data manipulation. It is common to import pandas as pd, which reduces the typing a bit (http://pandas. pydata.org/).
A good introduction to pandas has been written by Olson (2012).

Data Handling

a) Common Procedures

In statistical data analysis, labeled data structures have turned out to be immensely useful. To handle labeled data in *Python, pandas* introduces the so-called *DataFrame* objects. A DataFrame is a two-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table. DataFrames are the most commonly used pandas objects

Let me start with a specific example, by creating a DataFrame with three columns, called “Time,” “x,” and “y”:

```
import numpy as np
import pandas as pd
t = np.arange(0,10,0.1)
x = np.sin(t)
y = np.cos(t)
df = pd.DataFrame({'Time':t, 'x':x, 'y':y})
```

In *pandas*, rows are addressed through indices and columns through their name.
To address the first column only, you have two options:

```
df.Time
df['Time']
```

If you want to extract two columns at the same time, ask for several variables in a list:

```
data = df[['Time', 'y']]
```

To display the first or last rows, use

```
data.head()
data.tail()
```

To extract the six rows from 5 to 10, use

data[4:10] as 10 _ 4 D 6. (I know, the array indexing takes some time to get used to. Just keep in mind that *Python* addresses the *locations between* entries, not the entries, and that it starts at 0!)

The handling of DataFrames is somewhat different from the handling of *numpy* arrays. For example, (numbered) rows and (labeled) columns can be addressed simultaneously as follows:
df[['Time', 'y']][4:10]

You can also apply the standard row/column notation, by using the method iloc:
df.iloc[4:10, [0,2]]

Finally, sometimes you want to have direct access to the data, not to the DataFrame. You can do this with data.values which returns a *numpy* array.

b) Notes on Data Selection

While *pandas*' DataFrames are similar to numpy arrays, their philosophy is different, and I have wasted a lot of nerves addressing data correctly. Therefore I want to explicitly point out the differences here:

numpy handles "rows" first. E.g., data[0] is the first row of an array **pandas** starts with the columns. E.g., df['values'][0] is the first element of the column 'values'.

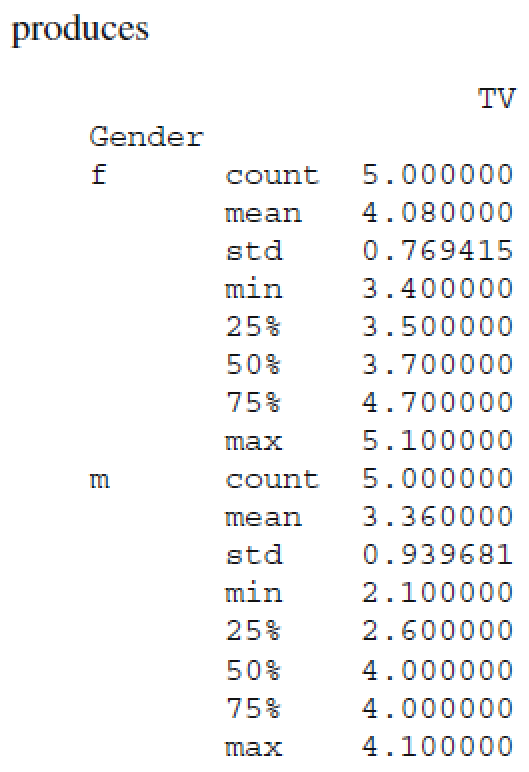
If a DataFrame has labeled rows, you can extract for example the row "rowlabel" with df.loc['rowlabel']. If you want to address a row by its number, e.g., row number "15," use df.iloc[15]. You can also use iloc to address "rows/columns," e.g., df.iloc[2:4,3].

Slicing of rows also works, e.g., df[0:5] for the first 5 (!) rows. A sometimes confusing convention is that if you want to slice out a single row, e.g., row "5," you have to use df[5:6]. If you use df[5] alone, you get an error!

2.5.2 Grouping

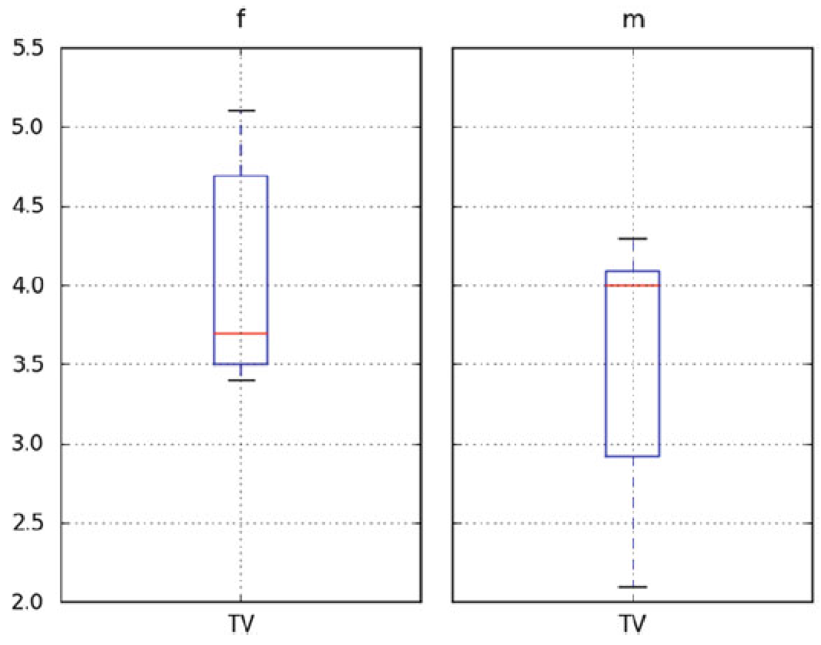
pandas offers powerful functions to handlemissing data which are often replaced by *nan's ("Not-A-Number")*. It also allows more complex types of data manipulation like pivoting. For example, you can use data-frames to efficiently group objects, and do a statistical evaluation of each group. The following data are simulated (but realistic) data of a survey on how many hours a day people watch the TV, grouped into "m"ale and "f"emale responses:

```
import pandas as pd
import matplotlib.pyplot as plt
data = pd.DataFrame({
'Gender': ['f', 'f', 'm', 'f', 'm',
'm', 'f', 'm', 'f', 'm', 'm'],
'TV': [3.4, 3.5, 2.6, 4.7, 4.1, 4.1,
5.1, 3.9, 3.7, 2.1, 4.3]
})
#------------------------------------------

# Group the data
grouped = data.groupby('Gender')
# Do some overview statistics
print(grouped.describe())
# Plot the data:
grouped.boxplot()
plt.show()

#------------------------------------------
# Get the groups as DataFrames
df_female = grouped.get_group('f')
# Get the corresponding numpy-array
values_female = grouped.get_group('f').values
```



produces

```
                      TV
Gender
f       count   5.000000
        mean    4.080000
        std     0.769415
        min     3.400000
        25%     3.500000
        50%     3.700000
        75%     4.700000
        max     5.100000
m       count   5.000000
        mean    3.360000
        std     0.939681
        min     2.100000
        25%     2.600000
        50%     4.000000
        75%     4.000000
        max     4.100000
```



For statistical analysis, *pandas* becomes really powerful if you combine it with *statsmodels* (see below).

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:

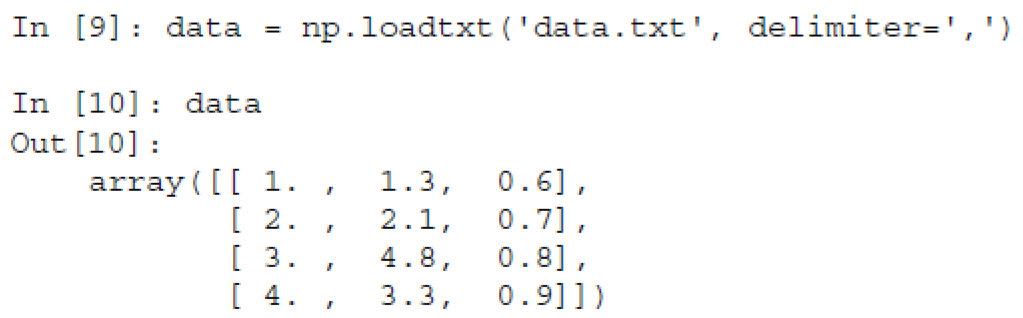
```
In [1]: import pandas as pd
In [2]: cd 'C:\Data\storage'
In [3]: pwd # Check if you were successful
In [4]: ls # List the files in that directory
In [5]: inFile = 'data.txt'
In [6]: df = pd.read_csv(inFile)
In [7]: df.head() # Check if first line is ok
In [8]: df.tail() # Check the last line
```

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!

Simple Text-Files
For example, a file data.txt with the following content

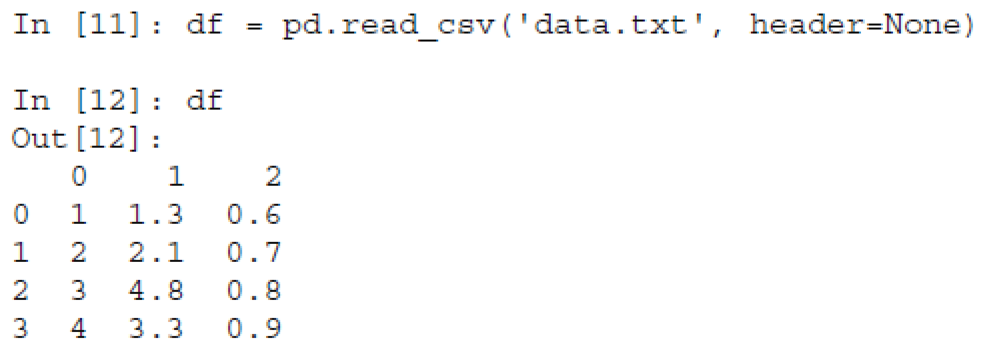
```
1, 1.3, 0.6
2, 2.1, 0.7
3, 4.8, 0.8
4, 3.3, 0.9
```

can be read in and displayed with



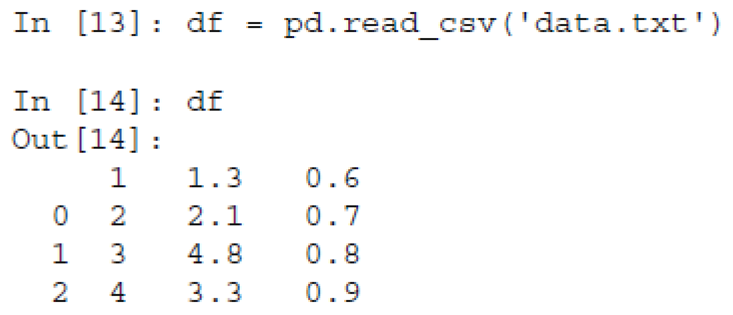
```
In [9]: data = np.loadtxt('data.txt', delimiter=',')

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]:
   0    1    2
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.

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]: df2.Value
Out[17]:
0    0.6
1    0.7
2    0.8
3    0.9
Name: Value, dtype: float64
```

c) Regular Expressions

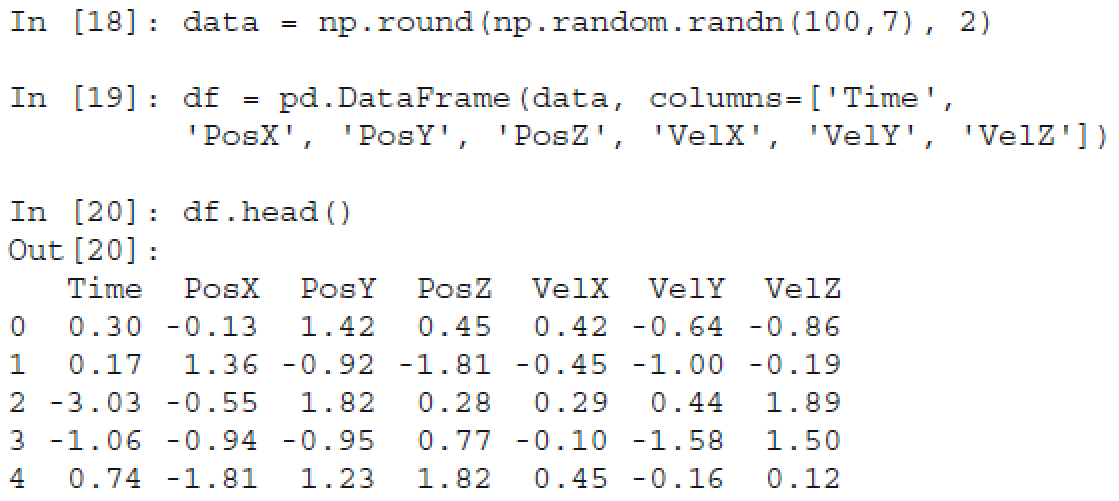
Working with text data often requires the use of simple *regular expressions*. Regular expressions are a very powerful way of finding and/or manipulating text strings. Many books have been written about them, and good, concise information on regular expressions can be found on the web, for example at:

• 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 regular expressions.

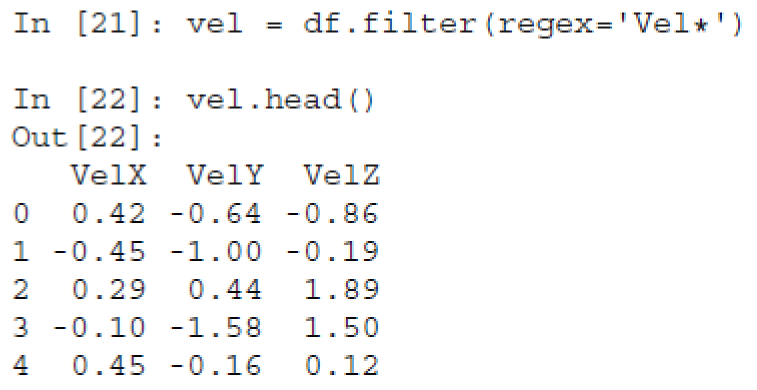
Let me give two examples how *pandas* can make use of regular expressions:

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

```
'''Get data from MS-Excel files, which are stored zipped on the WWW. '''

# author: Thomas Haslwanter, date: Nov-2015

# Import standard packages
import pandas as pd

# additional packages
import io
import zipfile

# Python 2/3 use different packages for "urlopen"
import sys
if sys.version_info[0] == 3:
from urllib.request import urlopen
else:
from urllib import urlopen

def getDataDobson(url, inFile):
'''Extract data from a zipped-archive on the web'''

# get the zip-archive
GLM_archive = urlopen(url).read()

# make the archive available as a byte-stream
zipdata = io.BytesIO()
zipdata.write(GLM_archive)

# extract the requested file from the archive, as a pandas XLS-file
myzipfile = zipfile.ZipFile(zipdata)
xlsfile = myzipfile.open(inFile)
# read the xls-file into Python, using Pandas, and return
```

```
the extracted data
xls = pd.ExcelFile(xlsfile)
df = xls.parse('Sheet1', skiprows=2)

return df

if __name__ == '__main__':
# Select archive (on the web) and the file in the archive
url = 'http://cdn.crcpress.com/downloads/C9500/GLM_data.zip'
inFile = r'GLM_data/Table 2.8 Waist loss.xls'

df = getDataDobson(url, inFile)
print(df)

input('All done!')
```

4.1 Datatypes

The choice of appropriate statistical procedure depends on the data type. Data can be *categorical* or *numerical*. If the variables are numerical, we are led to a certain statistical strategy. In contrast, if the variables represent qualitative categorizations, then we follow a different path.

In addition, we distinguish between *univariate*, *bivariate*, and *multivariate* data. *Univariate data* are data of only one variable, e.g., the size of a person. *Bivariate data* have two parameters, for example, the x=y position in a plane, or the income as a function of age. *Multivariate data* have three or more variables, e.g., the position of a particle in space, etc.

4.1.1 Categorical

a) Boolean
Boolean data are data which can only have two possible values.

For example,
• female/male
• smoker/nonsmoker
• True/False

b) Nominal
Many classifications require more than two categories. Such data are called *nominal data*. An example is *married/single/divorced*.

c) Ordinal
In contrast to nominal data, *ordinal data* are ordered and have a logical sequence, e.g., *very few/few/some/many/very many*.

4.1.2 Numerical

a) Numerical Continuous

Whenever possible, it is best to record the data in their original continuous format, and only with a meaningful number of decimal places. For example, it does not make sense to record the body size with more than 1mm accuracy, as there are larger changes in body height between the size in the morning and the size in the evening, due to compression of the intervertebral disks.

b) Numerical Discrete

Some numerical data can only take on integer values. These data are called *numerical discrete*. For example *Number of children: 0 1 2 3 4 5 ….*