An\_Introduction\_Statistics\_Python\_c02

p22

**Chapter 2 Python**

*Python* is a very popular open source programming language. At the time of writing, *codeeval* was rating *Python* “the most popular language” for the fourth year in a row (http://blog.codeeval.com/codeevalblog). There are three reasons why I have switched from other programming languages to *Python*:

1. It is the most elegant programming language that I know.

2. It is free.

3. It is powerful.

**2.1 Getting Started**

*2.1.1 Conventions*

In this book the following conventions will be used:

* Text that is to be typed in at the computer is written in Courier font, e.g., plot(x,y).
* Optional text in command-line entries is expressed with square brackets and underscores, e.g., [\_InstallationDir\_]\bin. (I use the underscores in addition, as sometimes the square brackets will be used for commands.)
* Names referring to computer programs and applications are written in italics, e.g., *IPython*.
* I will also use italics when introducing new terms or expressions for the first time.

Code samples are marked as follows:

*Python* code samples.

All the marked code samples are freely available, under http://www.quantlet.de.

Additional *Python* scripts (the listings of complete programs, as well as the *Python* code used to generate the figures) are available at *github*: https://github.com/ thomas-haslwanter/statsintro\_python.git, in the directory ISP (for “Introduction to Statistics with Python”). ISP contains the following subfolders:

**Exercise\_Solutions** contains the solutions to the exercises which are presented at the end of most chapters.

**Listings** contains programs that are explicitly listed in this book.

**Figures** lists all the code used to generate the remaining figures in the book. **Code\_Quantlets** contains all the marked code samples, grouped by book-chapter.

Packages on *github* are called *repositories*, and can easily be copied to your computer: when git is installed on your computer, simply type

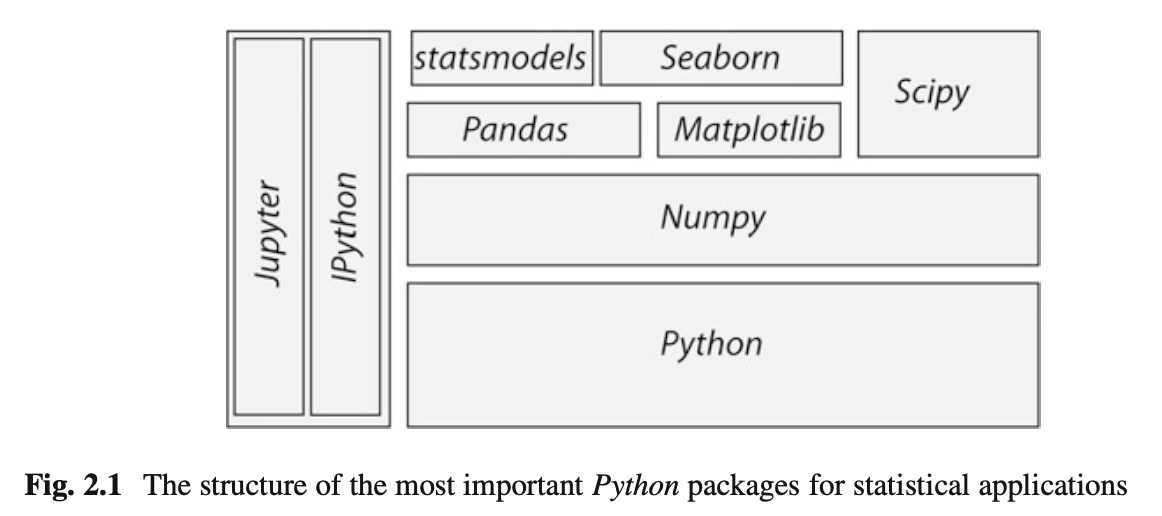
**git clone [\_RepositoryName\_]**

and the whole repository—code as well as data—will be “cloned” to your system. (See Sect. 2.4.4 for more information on *git*, *github* and code-versioning.)

***2.1.2 Distributions and Packages***

**a) 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.



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, under Windows 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/).

**b) PyPI: The Python Package Index**

The Python Package Index (*PyPI*) (Currently at https://pypi.python.org/pypi, but about to migrate to https://pypi.io) is a repository of software for the *Python* programming language. It currently contains more than 80,000 packages!

Packages from *PyPI* can be installed easily, from the Windows command shell (cmd) or the Linux terminal, with

**pip install [\_package\_]**

To update a package, use

**pip install [\_package\_] -U**

To get a list of all the *Python* packages installed on your computer, type

**pip list**

*Anaconda* uses conda, a more powerful installation manager. But *pip* also works with *Anaconda*.

***2.1.3 Installation of Python***

**a) Under Windows**

Neither *WinPython* nor *Anaconda* require administrator rights for installation.

**WinPython**

In the following, I assume that [\_WinPythonDir\_] is the installation directory for *WinPython*.

**Tip**: Do NOT install *WinPython* into the *Windows* program directory (typically C:\Program Files or C:\Program Files (x86)), because this typically leads to permission problems during the execution of *WinPython*.

* Download WinPython from https://winpython.github.io/.
* Run the downloaded .exe-file, and install *WinPython* into the  
     
  [\_WinPythonDir\_] of your choice.
* After the installation, make a change to your *Windows Environment*,  
   by typing Win -> env -> Edit environment variables for your account:

– Add[\_WinPythonDir\_]\python-3.5.1;[\_WinPythonDir\_] \python-3.5.1\Scripts\; to your PATH. (This makes *Python* and *ipython* accessible from the standard *Windows* command-line.)2

– If you do have administrator rights, you should activate

[\_WinPythonDir\_]\WinPython Control Panel.exe -> Advanced -> Register Distribution.

(This associates .py-files with this *Python* distribution.)

**Anaconda**

* Download *Anaconda* from **https://store.continuum.io/cshop/anaconda/.**
* Follow the installation instructions from the webpage. During the installation, allow *Anaconda* to make the suggested modifications to your environment PATH.
* After the installation: in the *Anaconda Launcher*, **click *update*** (besides the Apps), in order to ensure that you are running the latest version.

**Installing Additional Packages**   
**Important Note**: When I have had difficulties installing additional packages, I have been saved more than once by the pre-compiled packages from Christoph Gohlke, available under http://www.lfd.uci.edu/~gohlke/pythonlibs/: from there you can download the [\_xxx\_x].whl file for your current version of *Python*, and then install it simply with **pip install [\_xxx\_].whl**.

**b) Under Linux**

The following procedure worked on *Linux Mint 17.1*:

* Download *Anaconda* for *Python 3.5* (I used the 64 bit version, since I have a 64-bit *Linux Mint* Installation).
* Open terminal, and navigate to the location where you downloaded the file to.
* Install *Anaconda* with bash Anaconda3-4.0.0-Linux-x86.sh
* Update your Linux installation with **sudo apt-get update**

**Notes**

* You do NOT need root privileges to install *Anaconda*, if you select a user writable install location, such as ~/Anaconda.
* After the self extraction is finished, you should add the *Anaconda* binary directory to your PATH environment variable.
* As all of *Anaconda* is contained in a single directory, uninstalling *Anaconda* is easy: you simply remove the entire install location directory.
* If any problems remain, Mac and Unix users should look up Johansson’ installations tips:  
   (https://github.com/jrjohansson/scientific-python-lectures).

**c) Under Mac OS X**

Downloading *Anaconda* for *Mac OS X* is simple. Just

* go to continuum.io/downloads
* choose the Mac installer (make sure you select the *Mac OS X Python 3.x*    
  *Graphical Installer*), and follow the instructions listed beside this button.
* After the installation: in the *Anaconda Launcher*, click *update* (besides the  
   Apps), in order to ensure that you are running the latest version.

After the installation the *Anaconda* icon should appear on the desktop. No admin password is required. This downloaded version of *Anaconda* includes the *Jupyter notebook*, *Jupyter qtconsole* and the IDE *Spyder*.  
   
To see which packages (e.g., *numpy*, *scipy*, *matplotlib*, *pandas*, etc.) are featured in your installation look up the *Anaconda Package List* for your *Python* version. For example, the *Python*-installer may not include *seaborn*. To add an additional package, e.g., *seaborn*, open the terminal, and enter pip install seaborn.

***2.1.4 Installation of R and rpy2***

If you have not used *R* previously, you can safely skip this section. However, if you are already an avid *R* used, the following adjustments will allow you to also harness the power of *R* from within *Python*, using the package *rpy2*.

**a) Under Windows**

Also *R* does not require administrator rights for installation. You can download the latest version (at the time of writing *R 3.0.0*) from http://cran.r-project.org/, and install it into the [\_RDir\_] installation directory of your choice.

**With WinPython**

• After the installation of *R*, add the following two variables to your *Windows Environment*, by typing

Win -> env -> Edit environment variables for your account:

– R\_HOME=[\_RDir\_]\R-3.3.0

– R\_USER=[\_YourLoginName\_]

The first entry is required for *rpy2*. The last entry is not really necessary, just better style.

**With Anaconda**

*Anaconda* comes without *rpy2*. So after the installation of *Anaconda* and *R*, you should:

* Get *rpy2* from http://www.lfd.uci.edu/~gohlke/pythonlibs/: Christoph Gohlkes *Unofficial Windows Binaries for Python Extension Packages* are one of the mainstays of the Python community—Thanks a lot, Christoph!
* Open the *Anaconda command prompt*
* Install *rpy2* with pip. In my case, the command was  
   pip rpy2-2.6.0-cp35-none-win32.whl

**b) Under Linux**

• After the installation of *Anaconda*, install *R* and *rpy2* with conda install -c https://conda.binstar.org/r rpy2

***2.1.5 Personalizing IPython/*Jupyter**

When working on a new problem, I always start out with the *Jupyter qtconsole* (see Sect. 2.3). Once I have the individual steps working, I use the *IPython* command %history to get the sequence of commands I have used, and switch to an IDE (integrated development environment), typically *Wing* or *Spyder* (see below).

In the following, *[\_mydir\_]* has to be replaced with your home-directory (i.e., the directory that opens up when you run cmd in Windows, or terminal in Linux). And [\_myname\_] should be replaced by your name or your userID.

To start up *IPython* in a folder of your choice, and with personalized startup scripts, proceed as follows.

**a) In Windows**

* Type Win+R, and start a command shell with cmd
* In the newly created command shell, type ipython. (This will launch an *ipython* session, and create the directory [\_mydir\_]\.ipython).
* Add the Variable IPYTHONDIR to your environment (see above), and set it to [\_mydir\_]\.ipython. This directory contains the startup-commands for your *ipython*-sessions.
* Into the startup folder [\_mydir\_].ipython\profile\_default\startup  
   place a file with, e.g., the name 00\_[\_myname\_].py, containing the startup commands that you want to execute every time that you launch *ipython.* My personal startup file contains the following lines:  
     
   import pandas as pd

import os

os.chdir(r'C:\[\_mydir\_]')

This will import *pandas*, and start you working in the directory of your choice.

Note: since Windows uses \ to separate directories, but \ is also the escape character in strings, directory paths using a simple backslash have to be preceded by “r,” indicating “raw strings”.

* Generate a file “ipy.bat” in *mydir*, containing  
     
   jupyter qtconsole

To see all *Jupyter Notebooks* that come with this book, for example, do the following:

* Type Win+R, and start a command shell with cmd
* Run the commands  
     
  cd [\_ipynb-dir\_] jupyter notebook
* Again, if you want, you can put this command sequence into a batch-file.

**b) In Linux**

* Start a Linux terminal with the command terminal
* In the newly created command shell, execute the following command  
     
  **ipython**  
     
  (This generates a folder :*ipython*)
* Into the sub-folder .ipython/profile\_default/startup, place a file with  
   e.g., the name 00[\_myname\_].py, containing the lines  
     
  **import pandas as pd**

**import os**

**os.chdir([\_mydir\_])**

* In your .bashrc file (which contains the startup commands for your shell- scripts), enter the lines  
     
  **alias ipy='jupyter qtconsole'**

**IPYTHONDIR='~/.ipython'**

* To see all *Jupyter Notebooks*, do the following:  
     
  – Go to **[\_mydir\_]**  
     
  – Create the file ipynb.sh, containing the lines  
     
  #!/bin/bash  
   cd [wherever\_you\_have\_the\_ipynb\_files]

jupyter notebook  
   
– Make the file executable, with chmod 755 ipynb.sh

Now you can start “your” *IPython* by just typing ipy, and the *Jupyter Notebook* by typing ipynb.sh

**c) InMacOSX**

* + Start the *Terminal* either by manually opening *Spotlight* or the shortcut CMD + SPACE and entering Terminal and search for “Terminal.”
  + In *Terminal*, execute ipython, which will generate a folder under [\_mydir\_]/. ipython.
  + Enter the command pwd into the *Terminal*. This lists [\_mydir\_]; copy this for later use.
  + Now open *Anaconda* and launch an editor, e.g., *spyder-app* or TextEdit.3 Create a file containing the command lines you regularly use when writing code (you can always open this file and edit it). For starters you can create a file with the following command lines:  
       
    import pandas as pd  
     import os os.chdir('[\_mydir\_]/.ipython/profile\_[\_myname\_]')
  + The next steps are somewhat tricky. *Mac OS X* hides the folders that start with “.”. So to access .ipython open File -> Save as n . . . . Now open a *Finder* window, click the *Go* menu, select Go to Folder and enter

[ \_mydir\_ ]/.ipython/profile\_default/startup. This will open a *Finder* window with a header named “startup”. On the left of this text there should be a blue folder icon. Drag and drop the folder into the *Save as. . .* window open in the editor. IPython has a *README* file explaining the naming conventions. In our case the file must begin with 00-, so we could name it 00-[ \_myname\_ ].

* Open your .bash\_profile (which contains the startup commands for your shellscripts), and enter the line

alias ipy='jupyter qtconsole'

* To see all *Jupyter* Notebooks, do the following:

– Go to [\_mydir\_]  
   
– Create the file ipynb.sh, containing the lines  
   
#!/bin/bash  
 cd [wherever\_you\_have\_the\_ipynb\_files]

jupyter notebook  
   
– Make the file executable, with chmod 755 ipynb.sh

***2.1.6 Python Resources***

If you have some programming experience, this book may be all you need to get the statistical analysis of your data going. But if required, very good additional information can be found on the web, where tutorials as well as good free books are available online. The following links are all recommendable sources of information if you are starting with *Python*:

* *Python Scientific Lecture Notes* If you don’t read anything else, read this! (http://scipy-lectures.github.com)
* *NumPy for Matlab Users* Start here if you have *Matlab* experience. (https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html; also check http://mathesaurus.sourceforge.net/matlab-numpy.html)
* *Lectures on scientific computing with Python* Great *Jupyter Notebooks*, from JR Johansson!  
     
  (https://github.com/jrjohansson/scientific-python-lectures)
* *The Python tutorial* The official introduction.  
     
  (http://docs.python.org/3/tutorial)

In addition free *Python* books are available, for different levels of programming  
 skills:

* + *A Byte of Python* A very good book, at the introductory level. (http://swaroopch.com/notes/python)
  + *Learn Python the Hard Way* (3rd Ed) A popular book that you can work through. (http://learnpythonthehardway.org/book/)
* *Think Python* For advanced programmers. (http://www.greenteapress.com/thinkpython)
* *Introduction to Python for Econometrics, Statistics and Data Analysis* Introduces *Python* with a focus on statistics (Sheppard 2015).
* *Probabilistic Programming and Bayesian Methods for Hackers* An excellent introduction into Bayesian thinking. The section on Bayesian statistics in this book is also based on that book (Pilon 2015).

I have not seen many textbooks on *Python* that I have really liked. My favorite introductory books are Harms and McDonald (2010), and the more recent Scopatz and Huff (2015).  
   
When I run into a problem while developing a new piece of code, most of the time I just google; thereby I stick primarily (a) to the official *Python* documentation pages, and (b) to http://stackoverflow.com/. Also, I have found user groups surprisingly active and helpful!

***2.1.7 First Python Programs***

**a) Hello World**

**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 “Hello World” to the screen: print('Hello World'). On my Windows 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

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

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 [ ... ].  
     
  **2.2 Python Data Structures  
     
  *2.2.1 Python Datatypes***   
  *Python* offers a number of powerful data structures, and it pays off to make yourself familiar with them. One can use
* *Tuples* to group objects of different types.
* *Lists* to group objects of the same types.
* *Arrays* to work with numerical data. (*Python* also offers the data type *matrix*.  
     
  However, it is recommended to use *arrays*, since many numerical and scientific functions will not accept input data in *matrix* format.)
* *Dictionaries* for named, structured data sets.
* *DataFrames* for statistical data analysis.

**Tuple ( )** A collection of different things. Tuples are “immutable”, i.e., they cannot be modified after creation.  
   
In [1]: import numpy as np  
 In [2]: myTuple = ('abc', np.arange(0,3,0.2), 2.5)  
 In [3]: myTuple[2]

Out[3]: 2.5

**List []** Lists are “mutable”, i.e., their elements can be modified. Therefore lists are typically used to collect items of the same type (numbers, strings, : : :). Note that “+” concatenates lists.

In [4]: myList = ['abc', 'def', 'ghij']

In [5]: myList.append('klm')

In [6]: myList

Out[6]: ['abc', 'def', 'ghij', 'klm']

In [7]: myList2 = [1,2,3]

In [8]: myList3 = [4,5,6]

In [9]: myList2 + myList3 Out[9]: [1, 2, 3, 4, 5, 6]

**Array []** *vectors* and *matrices*, for numerical data manipulation. Defined in *numpy*. Note that vectors and 1-d arrays are different: vectors CANNOT be transposed! With arrays, “+” adds the corresponding elements; and the array- method *.dot* performs a scalar multiplication of two arrays. (From Python 3.5 onward, this can also be achieved with the “@” operator.).

In [10]: myArray2 = np.array(myList2)

In [11]: myArray3 = np.array(myList3)

In [12]: myArray2 + myArray3

Out[12]: array([5, 7, 9])

In [13]: myArray2.dot(myArray3)

Out[13]: 32

**Dictionary { }** Dictionaries are unordered *(key/value)* collections of content, where the content is addressed as dict['key']. Dictionaries can be created with the command dict, or by using curly brackets {...}:

In [14]: myDict = dict(one=1, two=2, info='some information')

In [15]: myDict2 = {'ten':1, 'twenty':20, 'info':'more information'}

In [16]: myDict['info']

Out[16]: 'some information'

In [17]: myDict.keys()

Out[17]: dict\_keys(['one', 'info', 'two'])

**DataFrame** Data structure optimized for working with named, statistical data. Defined in *pandas*. (See Sect. 2.5.)

***2.2.2 Indexing and Slicing***

The rules for addressing individual elements in *Python* lists or tuples or in *numpy* arrays are pretty simple really, and have been nicely summarized by Greg Hewgill on *stackoverflow*4:

a[start:end] # items start through end-1

a[start:] # items start through the rest of the array

a[:end] # items from the beginning through end-1

a[:] # a copy of the whole array

There is also the step value, which can be used with any of the above:

a[start:end:step] # start through not past end, by step

The key points to remember are that indexing starts at 0, *not* at 1; and that the :end value represents the first value that is *not* in the selected slice. So, the difference between end and start is the number of elements selected (if step is 1, the default).

The other feature is that start or end may be a negative number, which means it counts from the end of the array instead of the beginning. So:

a[-1] # last item in the array

a[-2:] # last two items in the array

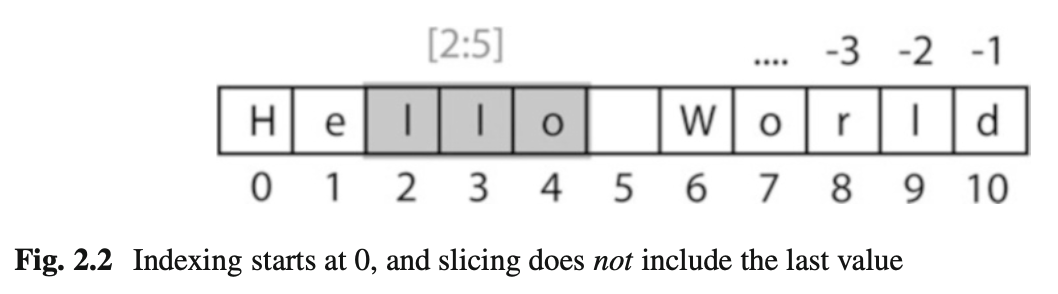
a[:-2] # everything except the last two items

As a result, a[:5] gives you the first five elements (*Hello* in Fig. 2.2), and a[-5:] the last five elements (*World*).

***2.2.3 Vectors and Arrays***

*numpy* is the *Python* module that makes working with numbers efficient. It is commonly imported with

import numpy as np



By default, it produces vectors. The commands most frequently used to generate numbers are:

**np.zeros**

generates zeros. Note that it takes only one(!) input. If you want to generate a matrix of zeroes, this input has to be a tuple, containing the number of rows/columns!

In [1]: import numpy as np

In [2]: np.zeros(3)

Out[2]: array([ 0., 0., 0.])

In [3]: np.zeros( (2,3) ) Out[3]: array([[ 0., 0., 0.],

[ 0., 0., 0.]])

**n.ones** generates ones.

**np.random.randn** generates normally distributed numbers, with a mean of 0 and a standard deviation of 1.

**np.arange** generates a range of numbers. Parameters can be

start, end, steppingInterval. Note that the end-value is excluded! While this can sometimes be a bit awkward, it has the advantage that consecutive sequences can be easily generated, without any overlap, and without missing any data points:

In [4]: np.arange(3) Out[4]: array([0, 1, 2])

In [5]: np.arange(1,3,0.5)

Out[5]: array([ 1. , 1.5, 2. , 2.5])

In [6]: xLow = np.arange(0,3,0.5) In [7]: xHigh = np.arange(3,5,0.5)

In [8]: xLow

Out[8]: array([ 0., 0.5, 1., 1.5, 2., 2.5])

In [9]: xHigh

Out[9]: array([ 3., 3.5, 4., 4.5])

**np.linspace** generates linearly spaced numbers.

In [10]: np.linspace(0,10,6)

Out[10]: array([ 0., 2., 4., 6., 8., 10.])

**np.array** generates a numpy array from given numerical data.

In [11]: np.array([[1,2], [3,4]]) Out[11]: array([ [1, 2],

[3, 4] ])

There are a few points that are peculiar to *Python*, and that are worth noting:

* Matrices are simply “lists of lists”. Therefore the first element of a matrix gives you the first row:  
     
  In [12]: Amat = np.array([ [1, 2], [3, 4] ])  
     
  In [13]: Amat[0] Out[13]: array([1, 2])
* A vector is not the same as a one-dimensional matrix! This is one of the few really un-intuitive features of *Python*, and can lead to mistakes that are hard to find. For example, vectors cannot be transposed, but matrices can

In [14]: x = np.arange(3)

In [15]: Amat = np.array([ [1,2], [3,4] ])

In [16]: x.T == x

Out[16]: array([ True, True, True], dtype=bool)

In [17]: Amat.T == Amat Out[17]: array([[ True, False],

[False, True]], dtype=bool)

**2.3 IPython/*Jupyter*: An Interactive Programming Environment**

A good workflow for source code development can make a very big difference for coding efficiency. For me, the most efficient way to write new code is as follows: I first get the individual steps worked out interactively in *IPython* (http:// ipython.org/). *IPython* provides a programming environment that is optimized for interactive computing with *Python*, similar to the command-line in *Matlab*. It comes with a command history, interactive data visualization, command completion, and lots of features that make it quick and easy to try out code. When the *pylab* mode is activated with %pylab inline, *IPython* automatically loads numpy and matplotlib.pyplot (which is the package used for generating plots) into the active workspace, and provides a very convenient, *Matlab*-like programming environment. The optional argument inline directs plots into the current *qtcon- sole/notebook*.

*IPython* uses *Jupyter* to provide different interface options, my favorite being the *qtconsole*:

**jupyter qtconsole**

A very helpful addition is the browser-based *notebook*, with support for code, text, mathematical expressions, inline plots and other rich media.

**jupyter notebook**

Note that many of the examples that come with this book are also available as *Jupyter Notebooks*, which are available at *github*: https://github.com/thomas- haslwanter/statsintro\_python.git.

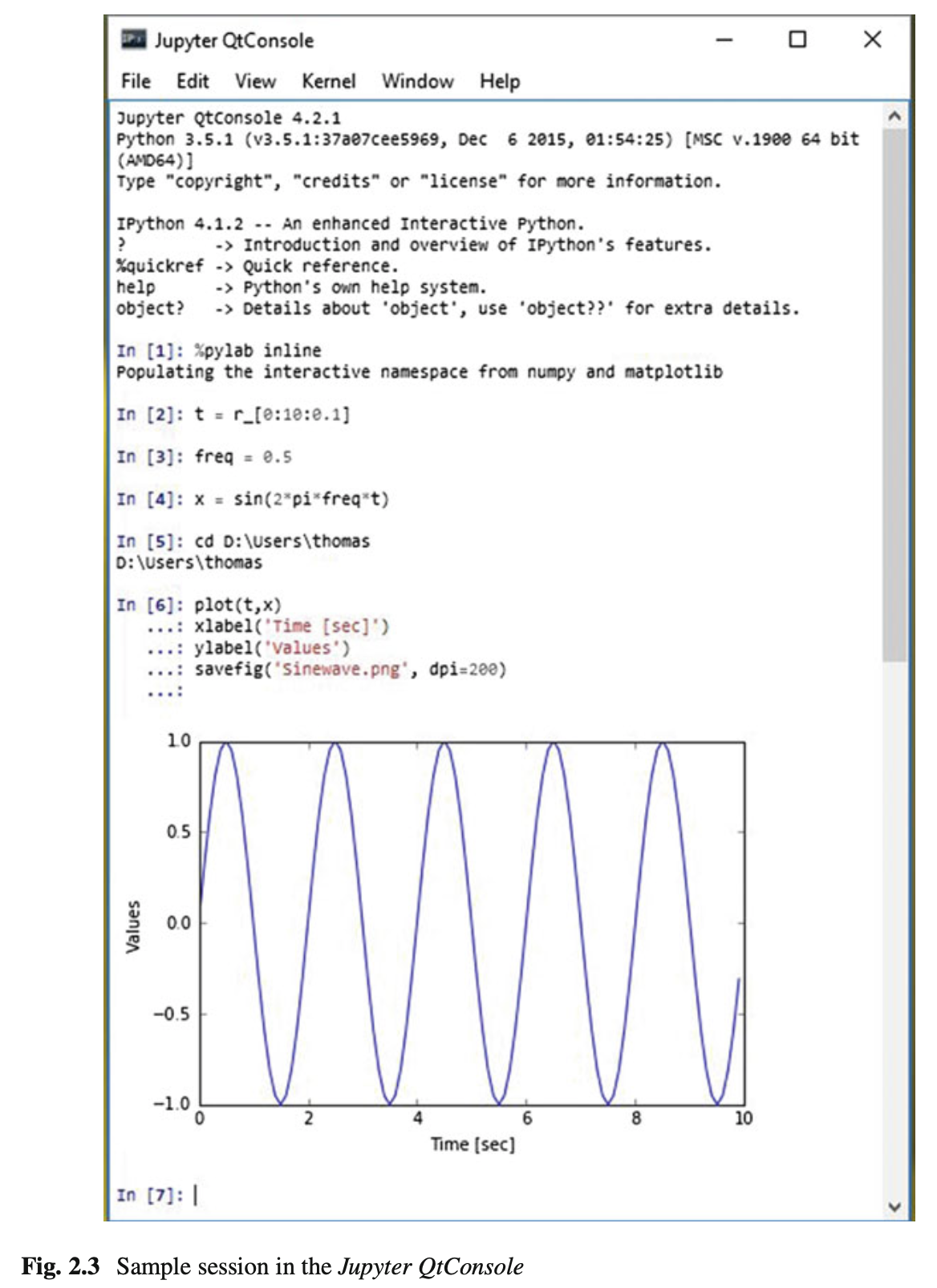
***2.3.1 First Session with the Qt Console***

An important aspect of statistical data analysis is the interactive, visual inspection of the data. Therefore I strongly recommend to start the data analysis in the *ipython qtonsole*.

For maximum flexibility, I start my *IPython* sessions from the command-line, with the command jupyter qtconsole. (Under *WinPython*: if you have problems starting *IPython* from the cmd console, use the *WinPython Command Prompt* instead—it is nothing else but a command terminal with the environment variables set such that *Python* is readily found.)

To get started with *Python* and *IPython*, let me go step-by-step through the *IPython* session in Fig. 2.3:

* *IPython* starts out listing the version of *IPython* and *Python* that are used, and showing the most important help calls.
* **In [1]:** The first command %pylab inline loads *numpy* and *matplotlib* into the current workspace, and directs *matplotlib* to show plots “inline”.  
     
  To understand what is happening here requires a short detour into the structure of scientific *Python*.  
     
  Figure 2.1 shows the connection of the most important *Python* packages that are used in this book. *Python* itself is an interpretative programming language, with no optimization for working with vectors or matrices, or for producing plots. *Packages* which extend the abilities of *Python* must be loaded explicitly. The most important package for scientific applications is *numpy* , which makes working with vectors and matrices fast and efficient, and *matplotlib*, which is the most common package used for producing graphical output. *scipy* contains important scientific algorithms. For the statistical data analysis, scipy.stats contains the majority of the algorithms that will be used in this book. *pandas* is a more recent addition, which has become widely adopted for statistical data analysis. It provides *DataFrames*, which are labeled, two-dimensional data structures, making work with data more intuitive. *seaborn* extends the plotting



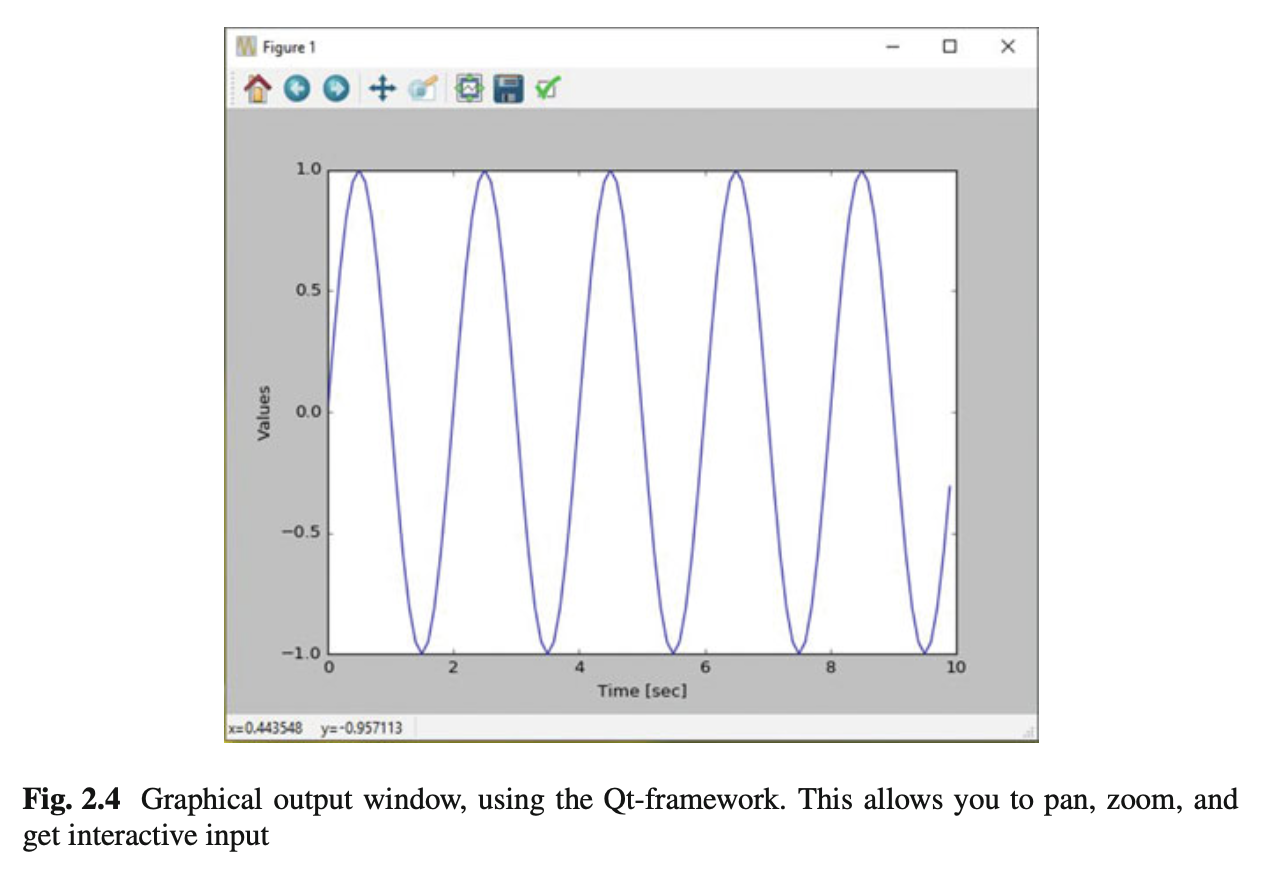
abilities of *matplotlib*, with a focus on statistical graphs. And *statsmodels* contains many modules for statistical modeling, and for advanced statistical analysis. Both *seaborn* and *statsmodels* make use of *pandas* DataFrames.

*IPython* provides the tools for interactive data analysis. It lets you quickly dis- play graphs and change directories, explore the workspace, provides a command history etc. The ideas and base structure of *IPython* have been so successful that the front end has been turned into a project of its own, *Jupyter*, which is now also used by other languages like *Julia*, *R*, and *Ruby*.

* **In [2]:** The command t = r\_[0:10:0.1] is a shorthand version for  
   t = arange(0, 10, 0.1), and generates a vector from 0 to 10, with a step size of 0.1. r\_ (and arange) are commands in the *numpy* package. (r\_ generates *row* vectors, and c\_ is the corresponding *numpy* command to generate *column* vectors.) However, since numpy has already been imported into the current workspace by %pylab inline, we can use these commands right away.
* **In [4]:** Since *t* is a vector, and *sin* is a function from *numpy*, the sine-value is calculated automatically for each value of *t*.
* **In [5]:** In *Python* scripts, changes of the current folder have to be performed with os.chdir(). However, tasks common with interactive computing, such as directory changes (%cd), bookmarks for directories (%bookmark), inspection of the workspace (%who and %whos), etc., are implemented as “IPython magic functions”. If no *Python* variable with the same name exists, the “%” sign can be left away, as here.
* **In [6]:** Since we have started out with the command %pylab inline, *IPython* generates plots in the *Jupyter QtConsole*, as shown in Fig. 2.3. To enter multi-line commands in *IPython*, one can use CTRL+Enter for additional command lines, indicated in the terminal by .... (The command sequence gets executed after the next empty line.)  
     
  Note that also generating graphics files is very simple: here I generate the PNG- file “Sinewave.png”, with a resolution of 200 dots-per-inch.  
     
  I have mentioned above that *matplotlib* handles the graphics output. In the *Jupyter QtConsole*, you can switch between inline graphs and output into an external graphics-window with %matplotlib inline and %matplotlib qt4 (see Fig.2.4). (Depending on your version of *Python*, you may have to replace %matplotlib qt4 with %matplotlib tk.) An external graphics window allows to zoom and pan in the figure, get the cursor position (which can help to find outliers), and get interactive input with the command ginput. *matplotlib*’s plotting commands closely follow the *Matlab* conventions.

***2.3.2 Notebook and rpy2***

Many of the code samples accompanying this book are also available as *Jupyter Notebooks*, and can be downloaded from https://github.com/thomas-haslwanter/ statsintro\_python.git. Therefore the concept of *Notebooks* and their integration with the *R*-language are briefly presented here.



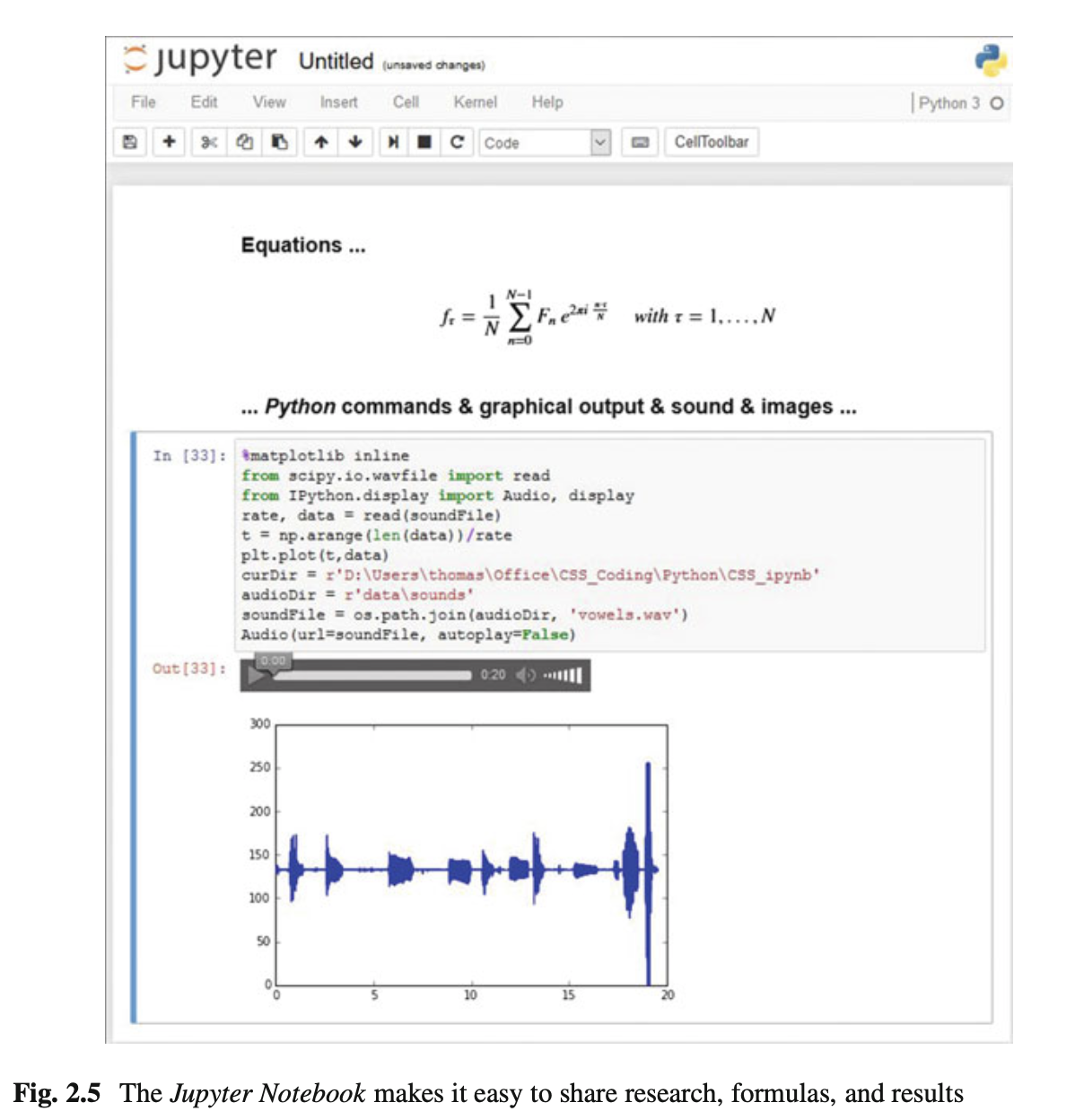
**a) The Notebook**

Since approximately 2013 the *IPython Notebook* has become a very popular way to share research and results in the *Python* community. In 2015 the development of the interface has become its own project, called *Jupyter*, since the *notebook* can be used not only with *Python* language, but also with *Julia*, *R*, and 40 other programming languages. The *notebook* is a browser based interface, which is especially well suited for teaching and for documentation. It allows to combine a structured layout, equations in the popular LaTeX format, and images, and can include resulting graphs and videos, as well as the output from *Python* commands (see Fig. 2.5).

**b) rpy2**

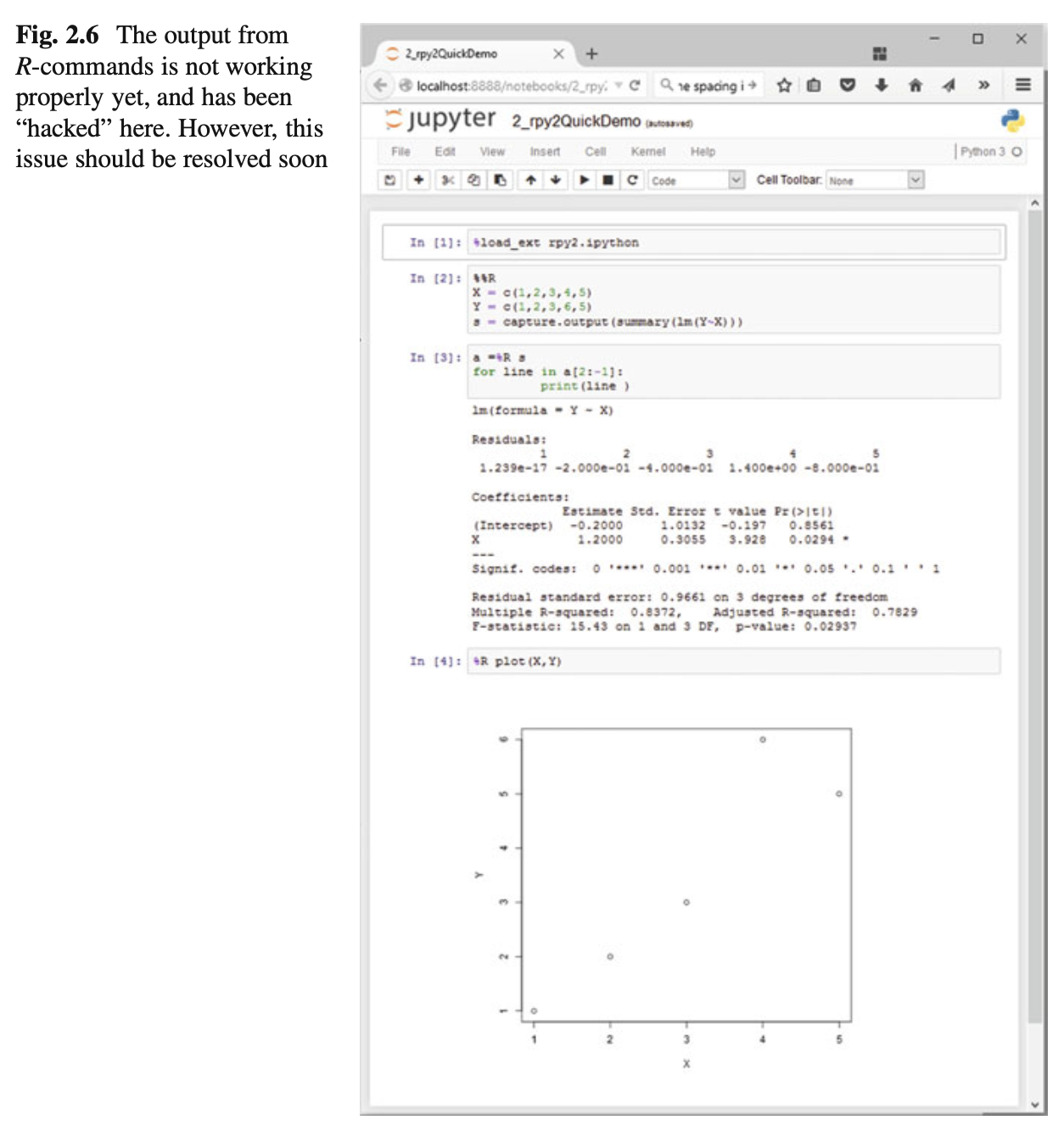
While *Python* is my preferred programming language, the world of advanced statistics is clearly dominated by *R*. Like *Python*, *R* is completely free and has a very active user community. While *Python* is a general programming language, *R* is optimized for the interactive work with statistical data. Many users swear that *ggplot* provides the best-looking graphs for statistical data.

To combine the best of both worlds, the package *rpy2* provides a way to transfer data from *Python* into *R*, execute commands in *R*, and transfer the results back into *Python*. In the *Jupyter Notebook*, with *rpy2* even *R* graphics can be fully utilized (see Fig. 2.6)!



***2.3.3 IPython Tips***

1. Use *IPython* in the *Jupyter QtConsole*, and customize your startup as described in Sect. 2.1.5: it will save you time in the long run!
2. For help on e.g., plot, use help(plot) or plot?. With one question mark the help gets displayed, with two question marks (e.g., plot??) also the source code is shown.
3. Check out the help tips displayed at the start of *IPython*.
4. Use TAB-completion, for file- and directory names, variable names, AND for commands.
5. To switch between inline and external graphs, use %matplotlib inline and  
      
   %matplotlib qt4.
6. By default, *IPython* displays data with a very high precision. For a more concise display, use %precision 3.
7. You can use edit [\_fileName\_] to edit files in the local directory, and  
      
   %run [\_fileName\_] to execute *Python* scripts in your current workspace.



**2.4 Developing Python Programs**

***2.4.1 Converting Interactive Commands into a Python Program***

*IPython* is very helpful in working out the command syntax and sequence. The next step is to turn these commands into a *Python* program with comments, that can be run from the command-line. This section introduces a fairly large number of *Python* conventions and syntax.

An efficient way to turn *IPython* commands into a function is to

* first obtain the command history with the command %hist or %history.
* copy the history into a good IDE (integrated development environment): I either use *Wing* (my clear favorite *Python* IDE, although it is commercial; see Fig. 2.7) or *Spyder* (which is good and free; see Fig. 2.8). *PyCharm* is another IDE with a good debugger, and with very good *vim*-emulation.
* turn it into a working *Python* program by adding the relevant package informa- tion, etc.

Converting the commands from the interactive session in Fig. 2.3 into a program, we get

**Listing 2.2 L2\_4\_pythonScript.py**

* 1 '''   
  2 Short demonstration of a Python script.

3

4 author:ThomasHaslwanter  
   
5 date: May-2015  
   
6 ver: 1.0  
   
7 '''

8

9 #Importstandardpackages  
   
10 importnumpyasnp  
   
11 importmatplotlib.pyplotasplt

12

13 #Generatethetime-values  
   
14 t=np.r\_[0:10:0.1]

15

16 #Setthefrequency,andcalculatethesine-value  
   
17 freq=0.5  
   
18 x=np.sin(2\*np.pi\*freq\*t)

19

20 #Plotthedata  
   
21 plt.plot(t,x)

22

23 #Formattheplot  
   
24 plt.xlabel('Time[sec]')  
   
25 plt.ylabel('Values')

26

27 #Generateafigure,onedirectoryup

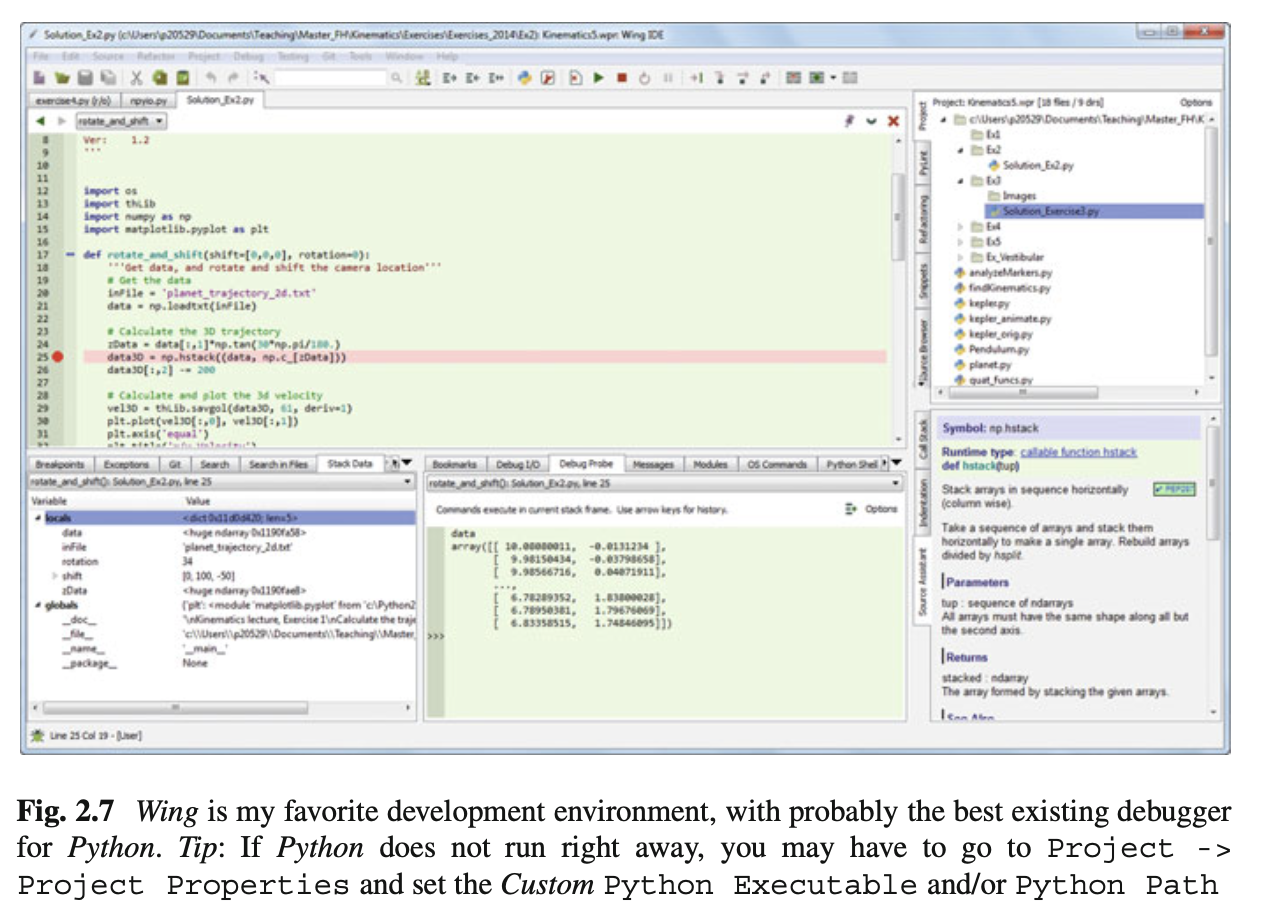
28 plt.savefig(r'..\Sinewave.png',dpi=200)

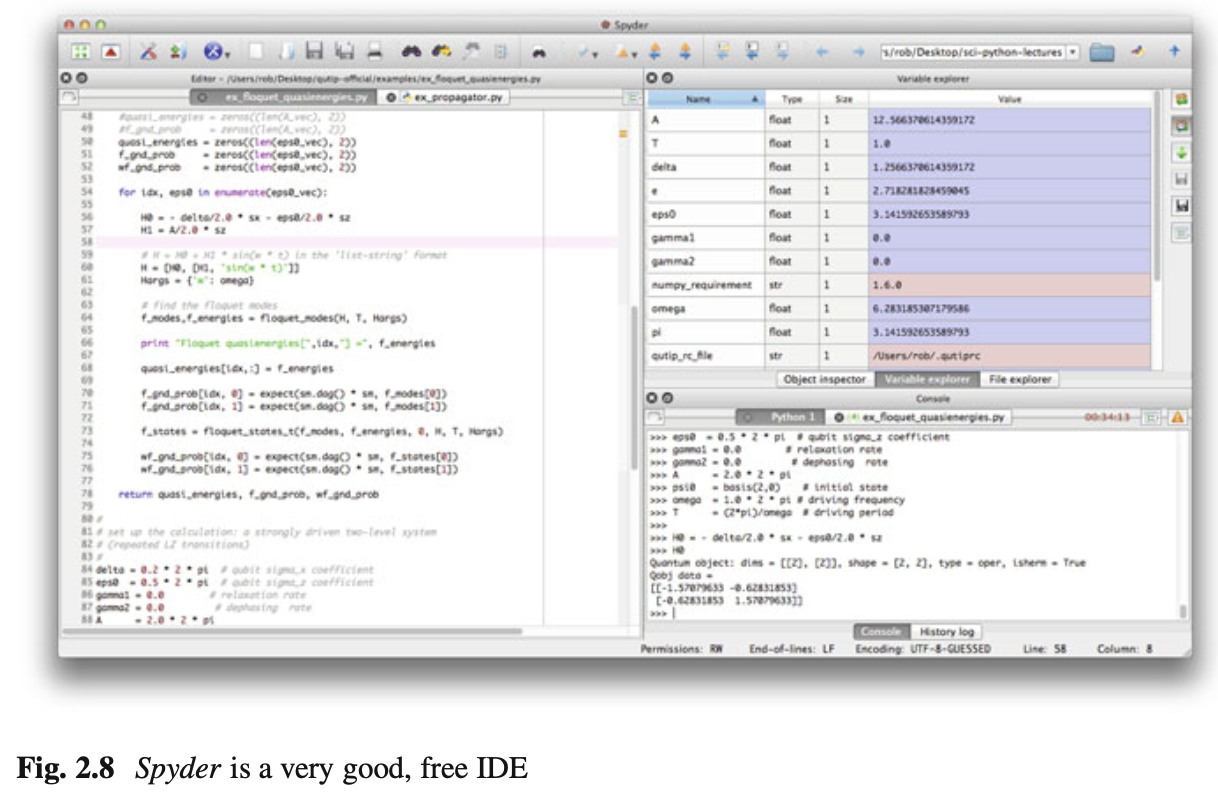
29

30 #Putitonthescreen  
   
31 plt.show()

The following modifications were made from the *IPython* history:

* The commands were put into a files with the extension “.py”, a so-called *Python module*.
* 1–7: It is common style to precede a *Python* module with a header block. Multi- line comments are given between triple inverted commas ''' [\_ xxx \_] '''. The first comment block describing the module should also contain information about author, date, and version number.





* 9: Single-line comments use “#”.
* 10–11: The required *Python* packages have to be imported explicitly. (In IPython, this is done for *numpy* and *matplotlib.pyplot* by the command %pylab.) It is customary to import *numpy as np*, and *matplotlib.pyplot*, the matplotlib module containing all the plotting commands, as *plt*.
* 14 etc: The numpy command r\_ has to be addressed through the corresponding package name, i.e., np.r\_. (In *IPython*, %pylab took care of that.)
* 18: Note that also “pi” is in *numpy*, so np.pi is needed!
* 21 etc: All the plotting commands are in the package plt.
* 28: Care has to be taken with backslashes in pathnames: in *Windows*, directories in path-names are separated by "\", which is also used as the escape-character in strings. To take "\" literally, a string has to be preceded by “r” (for “r”aw string), e.g., r'C:\Users\Peter' instead of 'C:\\Users\\Peter'.
* 34: While *IPython* automatically shows graphical output, *Python* programs don’t show the output until this is explicitly requested by plt.show(). The idea behind this is to optimize the program speed, only showing the graphical output when required. The output looks the same as in Fig. 2.4.

***2.4.2 Functions, Modules, and Packages***

*Python* has three different levels of modularization:

**Function** A *function* is defined by the keyword def, and can be defined anywhere in *Python*. It returns the object in the return statement, typically at the end of the function.

**Modules** A *module* is a file with the extension “.py”. Modules can contain function and variable definitions, as well as valid *Python* statements.

**Packages** A *package* is a folder containing multiple *Python* modules, and must have a file named \_\_init\_\_.py. For example, *numpy* is a *Python* package. Since *packages* are mainly important for grouping a larger number of modules, they won’t be discussed in this book.

**a) Functions**

The following example shows how functions can be defined and used.

**Listing 2.3 L2\_4\_pythonFunction.py**

1 '''DemonstrationofaPythonFunction

2   
3 author:thomashaslwanter,date:May-2015  
   
4 '''

5

6 #Importstandardpackages

* 7 importnumpyasnp

8

9 10

def incomeAndExpenses(data):

'''Find the sum of the positive numbers, and the sum of

the negative ones.'''

income = np.sum(data[data>0]) expenses = np.sum(data[data<0])

return (income, expenses) if\_\_name\_\_=='\_\_main\_\_':

testData = np.array([-5, 12, 3, -6, -4, 8])

# If only real banks would be so nice ;)

if testData[0] < 0:

print('Your first transaction was a loss, and will be

dropped.')

testData = np.delete(testData, 0)

else:

print('Congratulations: Your first transaction was a

gain!')

(myIncome, myExpenses) = incomeAndExpenses(testData)

print('You have earned {0:5.2f} EUR, and spent {1:5.2f} EUR.'.format(myIncome, -myExpenses))

* 1–4: Comment header.
* 6: Since *numpy* will be required in that module, it has to be imported. To reduce the writing to a minimum, it is conventionally called *np*.
* 9/10: Function definition, and a comment describing the function. Note that in *Python* the function block is defined by the indentation, not by any brackets or *end* statements! This is a feature that irritates many *Python* novices, but really helps to keep code clear and nicely formatted. Important: *Python* makes a difference between a tab and the equivalent amount of spaces. This can lead to errors which are really hard to detect, so use a good IDE that automatically converts tabs to spaces!
* 11:  
     
  – The sum command is taken from *numpy*, so it has to be preceded by .np.  
     
  – In *Python*, function arguments are indicated by round brackets (...), whereas elements of lists, tuples, vectors, and arrays are indicated by square brackets  
   [...].  
     
  – In *numpy* you can select elements of an array either with an index (see line 20), or with a boolean array (line 11).
* 14: *Python* also uses round brackets to form groups of elements, the so-called *tuples*. And the return statement does the obvious things: it returns elements from a function.
* 16: Here quite a few new aspects of *Python* come together:

– Just like function definitions, if-loops or for-loops use indentation to define their context.

– *Python* conventionally uses underscores (\_) to indicate private variables, which are not used for typical programming tasks.

– Here we check the variable with the name \_\_name\_\_, which is denoting the context of a module evaluation. If the module is run as a *Python* script, \_\_name\_\_ is set to \_\_main\_\_. But if a module is imported, it is set to the name of the importing module. This way it is possible to add code to a function that is only used when the module is executed, but not when the functions in this module are imported by other modules (see below).

* 17: Definition of a *numpy* array.
* 26: The two elements returned as a tuple from the function incomeAndExpenses can be immediately assigned to two different *Python* objects  
     
  (myIncome, myExpenses).
* 27: While there are different ways to produce formatted strings, this is probably the most elegant one: curly brackets { ... } indicate values that will be inserted, and can also contain formatting statements. The corresponding values are then passed into the string by the method format, e.g., print('The value of pi is {0}'.format(np.py)).

**b) Modules**  
   
To execute the module pythonFunction.py from the command-line, type  
 python pythonFunction.py. In Windows, if the extension “.py” is associated  
 with the *Python* program, it suffices to double-click the module, or to type pythonFunction.py on the command-line. In WinPython the association of the extension “.py” with the *Python* function is set by the *WinPython Control Panel.exe*, by the command *Register Distribution* : : : in the menu *Advanced*.

To run a module in *IPython*, use the magic function %run:  
   
In [56]: %run pythonFunction  
 Your first transaction was a loss, and will be dropped. You have earned 23.00 EUR, and spent 10.00 EUR.  
   
Note that you either have to be in the directory where the function is defined, or you have to give the full pathname.  
   
If you want to use a function or variable that is defined in a different module, you have to import that module. This can be done in three different ways. For the following example, assume that the other module is called newModule.py, and the function that we want from there newFunction.

* import newModule: The function can then be accessed with newModule.newFunction().
* from newModule import newFunction: In this case, the function can be called directly newFunction().
* from newModule import \*: This imports all variables and functions from newModule into the current workspace; again, the function can be called directly with newFunction(). However, use of this syntax is discouraged as it clutters up the current workspace.

If you import a module multiple times, *Python* recognizes that the module is already known, and skips later imports. If you want to override this, and explicitly want to re-import a module that has changed, you have to use the command reload from the package importlib:

**from importlib import reload**

**reload(pythonFunction)**

**Python 2.x:** reload does NOT need to be imported from importlib, but is available as a core module.

The next example shows you how to import functions from one module into another module:

**Listing 2.4 L2\_4\_pythonImport.py**

1 '''DemonstrationofimportingaPythonmodule 2

3 author:ThH,date:May-2015'''

4   
5 #Importstandardpackages  
   
6 importnumpyasnp

7

8 #additionalpackages:thisimportsthefunctiondefined  
   
above  
   
9 importL2\_4\_pythonFunction

10   
11 #Generatetest-data  
   
12 testData=np.arange(-5,10)

13   
14 #Useafunctionfromtheimportedmodule  
   
15 out=L2\_4\_pythonFunction.incomeAndExpenses(testData)

16

17 #Showsomeresults

18 print('Youhaveearned{0:5.2f}EUR,andspent{1:5.2f}EUR.'  
   
 .format(out[0], -out[1]))

* 9: Here the module pythonFunction (that we have just discussed above) is imported. Note that the code in the section if \_\_name\_\_ == '\_\_main\_\_' in pythonFunction.py is NOT executed when the module is imported!
* 15: To access the function incomeAndExpenses from the module pythonFunction, module- and function-name have to be given: incomeAndExpenses.pythonFunction(...)

***2.4.3 Python Tips***

1. Stick to the standard conventions.  
   * Every function should have a documentation string on the line below the function definition.
   * Packages should be imported with their commonly used names:  
        
     import numpy as np  
      import matplotlib.pyplot as plt import scipy as sp  
      import pandas as pd  
      import seaborn as sns
2. To get the current directory, use os.path.abspath(os.curdir). And in Python modules a change of directories can NOT be executed with cd (as in *IPython*), but instead requires the command os.chdir(...).
3. Everything in *Python* is an object: to find out about “obj”, use type(obj) and dir(obj).
4. Learn to use the debugger. Personally, I always use the debugger from the IDE, and rarely resort to the built-in debugger *pdb*.
5. Know lists, tuples, and dictionaries; also, know about *numpy* arrays and *pandas* DataFrames.
6. Use functions a lot, and understand the if \_\_name\_\_=='\_\_main\_\_': construct.
7. If you have all your personal functions in the directory mydir, you can add this directory to your PYTHONPATH with the command import sys  
      
   sys.path.append('mydir')
8. If you are using non-ASCII characters, such as the German\"{o}\"{a}\"{u}{\ss} or the French \`{e}\'{e}, you have to let *Python* know, by adding  
    # -\*- coding: utf-8 -\*-  
    in the first or second line of your *Python* module. This has to be done, even if the non-ASCII characters only appear in the comments! This requirement arises from the fact that *Python* will default to ASCII as standard encoding if no other encoding hints are given.

***2.4.4 Code Versioning***

Computer programs rarely come out perfect at the first try. Typically they are developed iteratively, by successively eliminating the known errors. *Version control* programs, also known as *revision control* programs, allow tracking only the modifications, and storing previous versions of the program under development. If the latest changes cause a new problem, it is then easy to compare them to earlier versions, and to restore the program to a previous state.

I have been working with a number of version control programs, and *git* is the first one I am really happy with. *git* is a version control program, and *github* is a central source code repository. If you are developing computer software, I strongly recommend the use of *git*. It can be used locally, with very little overhead. And it can also be used to maintain and manage a remote backup copy of the programs. While the real power of *git* lies in its features for collaboration, I have been very happy with it for my own data and software. An introduction to *git* goes beyond the scope of this book, but a very good instruction is available under https://git-scm. com/. Good, short, and simple starting instructions—in many languages—can be found at http://rogerdudler.github.io/git-guide/.

I am mostly working under Windows, and *tortoisegit* (https://tortoisegit.org/) provides a very useful Windows shell interface for *git*. For example, in order to clone a repository from *github* to a computer where *tortoisegit* is installed, simply right- click in the folder on your computer where you want the repository to be installed, select Git Clone ..., and enter the repository name—and the whole repository will be cloned there. Done!

*github* (https://github.com/) is an online project using *git*, and the place where the source code for the majority of *Python* packages is hosted.

**2.5 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).

***2.5.1 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 = 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 handle missing 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())

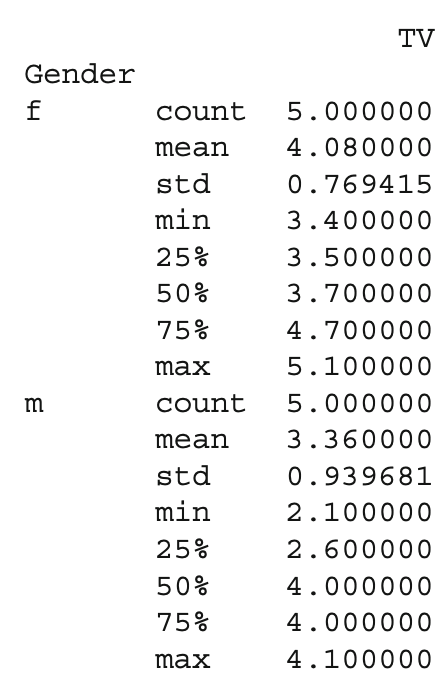
# Plot the data: grouped.boxplot() plt.show()

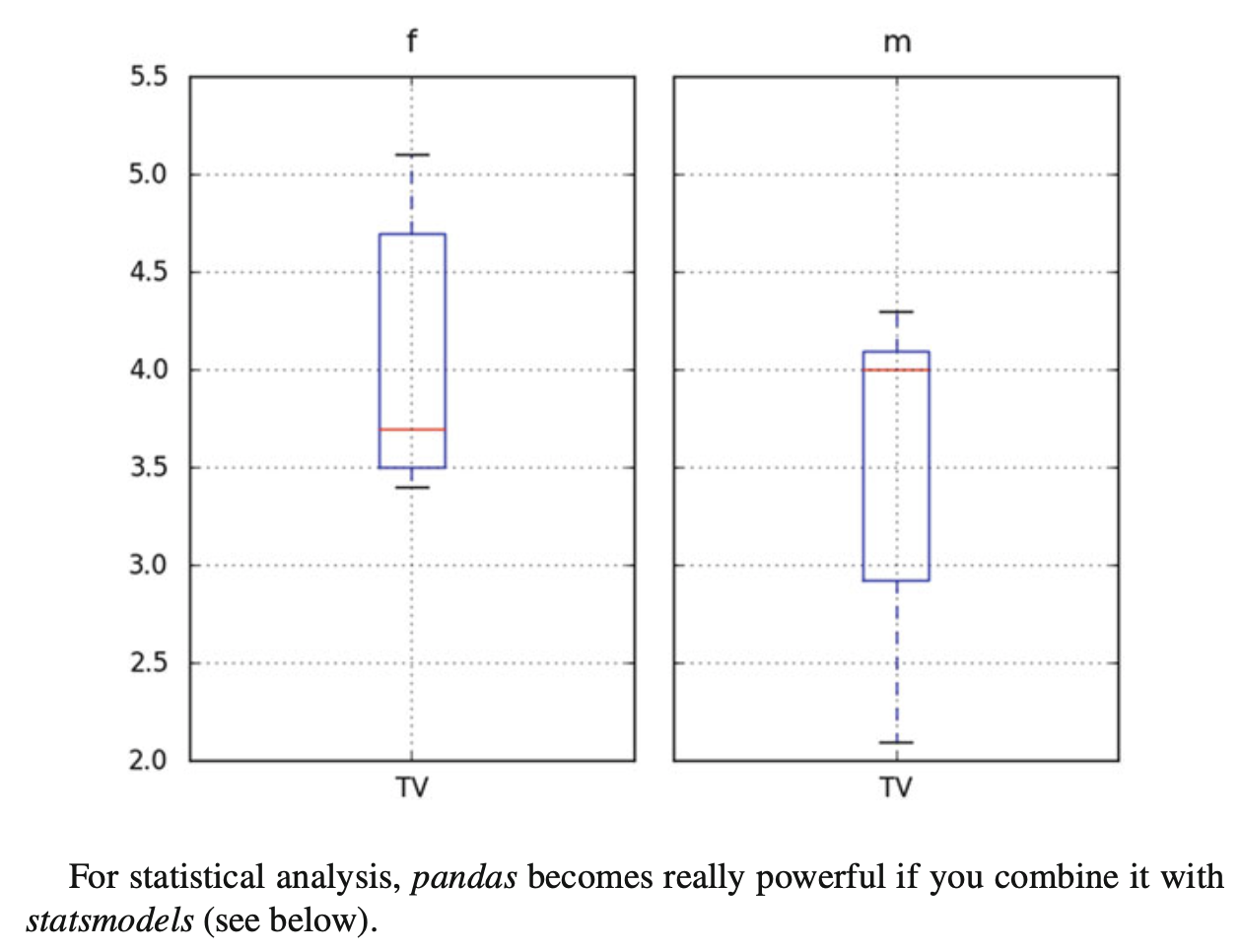
#-------------------------------------------- # Get the groups as DataFrames

df\_female = grouped.get\_group('f')

# Get the corresponding numpy-array values\_female = grouped.get\_group('f').values

produces





**2.6 Statsmodels: Tools for Statistical Modeling**

*statsmodels* is a *Python* package contributed to the community by the *statsmodels* development team (http://www.statsmodels.org/). It has a very active user commu- nity, and has in the last five years massively increased the functionality of *Python* for statistical data analysis. *statsmodels* provides classes and functions for the estimation of many different statistical models, as well as for conducting statistical tests and statistical data exploration. An extensive list of result statistics are available for each estimator.

*statsmodels* also allows the formulation of models with the popular formula language based on the notation introduced by Wilkinson and Rogers (1973), and also used by *S* and *R*. For example, the following example would fit a model that assumes a linear relationship between *x* and *y* to a given dataset:

import numpy as np

import pandas as pd

import statsmodels.formula.api as sm

# Generate a noisy line, and save the data in a pandas-DataFrame x = np.arange(100)

y = 0.5\*x - 20 + np.random.randn(len(x))

df = pd.DataFrame({'x':x, 'y':y})

# Fit a linear model, using the "formula" language # added by the package "patsy"

model = sm.ols('y~x', data=df).fit()

print( model.summary() )

Another example would be a model that assumes that “success” is determined by intelligence” and “diligence,” as well as the interaction of the two. Such a model could be described by

*success* 􏰙 *intelligence* 􏰚 *diligence*

More information on that topic is presented in Chap. 11 (“Statistical Models”).

An extensive list of result statistics are available for each estimator. The results of all *statsmodels* commands have been tested against existing statistical packages to ensure that they are correct. Features include:

* Linear Regression
* Generalized Linear Models
* Generalized Estimating Equations
* Robust Linear Models
* Linear Mixed Effects Models
* Regression with Discrete Dependent Variables
* ANOVA
* Time Series analysis
* Models for Survival and Duration Analysis
* Statistics (e.g., Multiple Tests, Sample Size Calculations, etc.)
* Nonparametric Methods
* Generalized Method of Moments
* Empirical Likelihood
* Graphics functions
* A Datasets Package

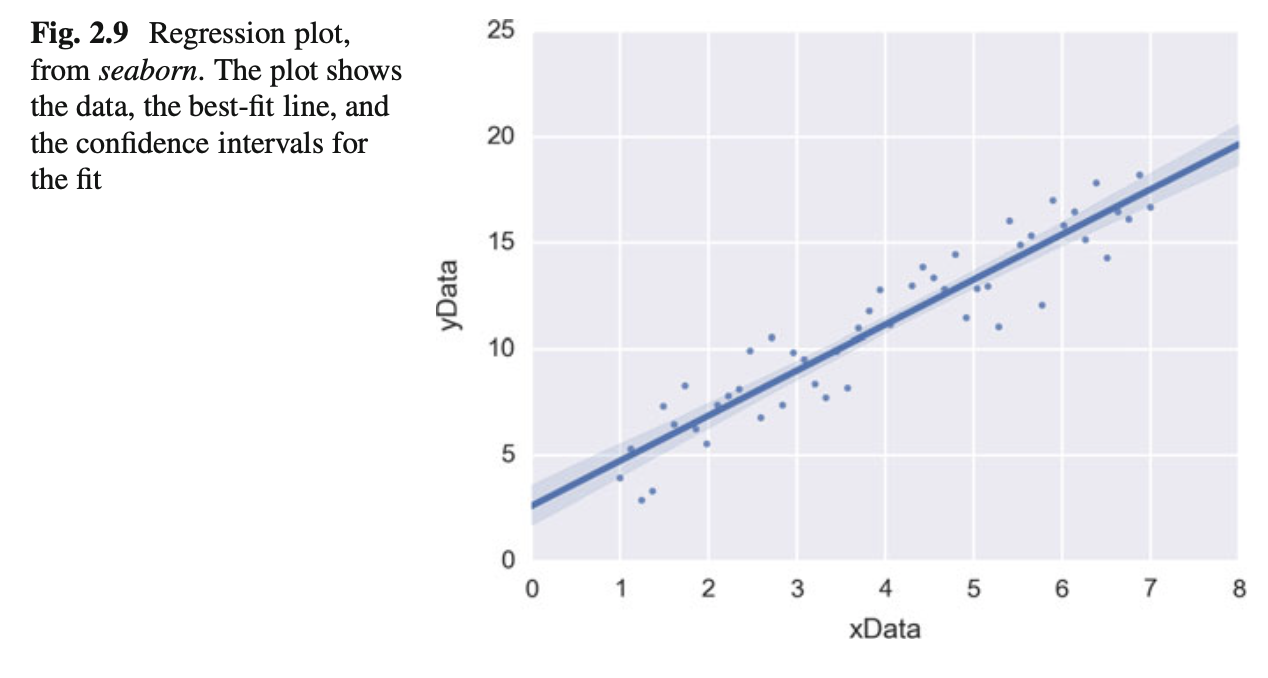
**2.7 Seaborn: Data Visualization**

*seaborn* is a *Python* visualization library based on *matplotlib*. Its primary goal is to provide a concise, high-level interface for drawing statistical graphics that are both informative and attractive http://stanford.edu/~mwaskom/software/seaborn/ (Fig. 2.9).

For example, the following code already produces a nice regression plot (Fig. 2.9), with line-fit and confidence intervals:

import numpy as np

import matplotlib.pyplot as plt



import pandas as pd

import seaborn as sns

x = np.linspace(1, 7, 50)

y = 3 + 2\*x + 1.5\*np.random.randn(len(x)) df = pd.DataFrame({'xData':x, 'yData':y}) sns.regplot('xData', 'yData', data=df) plt.show()

**2.8 General Routines**

In the examples used later in this book, a few tasks come up repeatedly: reading in data, setting the desired font size and formatting parameters, and generating graphical output files. The two following modules handle those tasks. If you are interested you can check them out; but their understanding is not really required:

Code: “ISP\_mystyle.py”5: sets commonly used formatting options, and provides functions for standardized graphics-output into files.

**2.9 Exercises**

**2.1 DataInput**

Read in data from different sources:

* A CVS-file with a header (’.\Data\data\_kaplan\swim100m.csv’). Also show the first 5 data points.
* An MS-Excel file (’.\Data\data\_others\Table 2.8 Waist loss.xls’). Show the last five data points.
* Read in the same file, but this time from the zipped archive http://cdn.crcpress. com/downloads/C9500/GLM\_data.zip.

**2.2 First Steps with *Pandas***

* + Generate a *pandas* DataFrame, with the x-column time stamps from 0 to 10 s, at a rate of 10 Hz, the y-column data values with a sine with 1.5 Hz, and the z-column the corresponding cosine values. Label the x-column “Time”, and the y-column “YVals”, and the z-column “ZVals”.
  + Show the head of this DataFrame.
  + Extract the data in lines 10–15 from “Yvals” and “ZVals”, and write them to the file “out.txt”.
  + Let the user know where the data have been written to.