

Thomas Haslwanter

An Introduction to Statistics with Python

With Applications in the Life Sciences

Second Edition



Springer

Statistics and Computing

Series Editor

Wolfgang Karl Härdle, Humboldt-Universität zu Berlin, Berlin, Germany

Statistics and Computing (SC) includes monographs and advanced texts on statistical computing and statistical packages.

More information about this series at <https://link.springer.com/bookseries/3022>

Thomas Haslwanter

An Introduction to Statistics with Python

With Applications in the Life Sciences

Second Edition



Springer

Thomas Haslwanter  School of Medical Engineering and Applied Social Sciences University of Applied Sciences Upper Austria Linz, Austria

ISSN 1431-8784

Statistics and Computing

ISBN 978-3-030-97370-4

<https://doi.org/10.1007/978-3-030-97371-1>

ISSN 2197-1706 (electronic)

ISBN 978-3-030-97371-1 (eBook)

1st edition: © Springer International Publishing Switzerland 2016

2nd edition: © Springer Nature Switzerland AG 2022

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

*To my two-, three-, and four-legged
household companions: Jean, Felix, and his
sister Jessica: Thank you so much for all the
support you have provided over the years!*

Preface

Preface to the First Edition

In the data analysis for my own research work, I was often slowed down by two things: (1) I did not know enough statistics, and (2) the books available would provide a theoretical background, but no real practical help. The book you are holding in your hands (or on your tablet or laptop) is intended to be the book that will solve this very problem. It is designed to provide enough basic understanding so that you know what you are doing, and it should equip you with the tools you need. I believe that the *Python* solutions provided in this book for the most basic statistical problems address at least 90% of the problems that most physicists, biologists, and medical doctors encounter in their work. So if you are the typical graduate student working on your degree, or a medical researcher analyzing your latest experiments, chances are that you will find the tools you require here—explanation and source-code included.

This is the reason I have focused on statistical basics and hypothesis tests in this book, and refer only briefly to other statistical approaches. I am well aware that most of the tests presented in this book can also be carried out using statistical modeling. But in many cases, this is not the methodology used in many life science journals. Advanced statistical analysis goes beyond the scope of this book, and—to be frank—exceeds my own knowledge of statistics.

My motivation for providing the solutions in Python is based on two considerations. One is that I would like them to be available to everyone. While commercial solutions like *Matlab*, *SPSS*, *Minitab* etc. offer powerful tools, most can only use them legally in an academic setting. In contrast, Python is completely free (as in free beer is often heard in the Python community). The second reason is that Python is the most beautiful coding language that I have yet encountered; and around 2010 Python and its documentation matured to the point where one can use it without being an serious coder. Together, this book, Python, and the tools that the Python ecosystem offers today provide a beautiful, free package that covers all the statistics that most researchers will need in their lifetime.

Preface to the Second Edition

Since the publication of the first edition, Python has continuously gained popularity and become firmly established as one of the foremost programming languages for statistical data analysis. All the core packages have matured. And thanks to the stunning development of *Jupyter* as an interactive programming environment, Python has become even more accessible for people with little programming background. To reflect these developments, and to incorporate the suggestions I have received for improving the presentation of the material, Springer has given me the opportunity to bring out a new edition of *Introduction to Statistics with Python*.

Compared to the first edition, the following changes have been made:

- The package *pandas* and its *DataFrames* have become an integral part of scientific Python, as has the *Jupyter* framework for interactive data environments. Correspondingly, a bigger amount of space has been dedicated to their introduction.
- A new package, *pingouin*, is promising a simplified and more powerful interface for many common statistics function. This package is introduced, and many application examples have been added.
- The visualization of data has been expanded, including the preparation of publication-ready graphics.
- The design of experiments and power analyses are discussed in more detail.
- A new section has been added on the confidence intervals of frequently used statistical parameters.
- A new chapter has been added on finding patterns in data, including an introduction to the correlation coefficient, cross- and autocorrelation. For an application of these concepts, a short introduction is given to time series analysis.

As for the first edition, all examples and solutions from this book are again available online. This includes code samples and example programs, Jupyter Notebooks with additional or extended information, as well as the data and Python code used to generate most of the figures. They can be downloaded from <https://github.com/thomas-haslwanter/statsintro-python-2e>.

I hope this book will help you with the statistical analysis of your data, and convey some of the often really simple ideas behind the sometimes awkwardly named statistical analysis procedures.

For Whom This Book Is

This book assumes that

- you have some basic programming experience: If you have done no programming previously, you may want to start out with Python using some of the great links provided in the text. Starting programming *and* starting statistics may be a bit

much all at once. However, solutions provided to the exercises at the end of most chapters should help you to get up to speed with Python.

- you are not a statistics expert: If you have advanced statistics experience, the online help in Python and the Python packages may be sufficient to allow you to do most of your data analysis right away. This book may still help you to get started with Python. However, the book concentrates on the basic ideas of statistics and on hypothesis tests, and only the last part introduces linear regression modeling and Bayesian statistics.

This book is designed to give you all (or at least most of) the tools that you will need for statistical data analysis. I attempt to provide the background you need to understand what you are doing. I do not prove any theorems, and do not apply mathematics unless necessary. For all tests, a working Python program is provided. In principle, you just have to define your problem, select the corresponding program, and adapt it to your needs. This should allow you to get going quickly, even if you have little Python experience. This is also the reason why I have not provided the software as one single Python package; I expect that you will have to tailor each program to your specific setup (data format, etc.).

This book is organized into three parts

Part I gives an introduction to Python: how to set it up, simple programs to get started, and tips on how to avoid some common mistakes. It also shows how to read data from different sources into Python, and how to visualize statistical data.

Part II provides an introduction to statistical analysis; on how to design a study, power analysis, and how best to analyze data; probability distributions; and an overview of the most important hypothesis tests. Even though modern statistics is firmly based in statistical modeling, hypothesis tests still seem to dominate the life sciences. For each test, a Python program is provided that shows how the test can be implemented.

Part III provides an introduction to correlation and regression analysis, time series analysis, and statistical modeling, and a look at advanced statistical analysis procedures. I have also included tests on discrete data in this section, such as logistic regression, as they utilize “generalized linear models” which I regard as advanced. This part ends with a presentation of the basic ideas of Bayesian statistics.

To achieve all those goals as quickly as possible, the Appendix A of the book provides hints on how to efficiently develop correct and working code. This should get you to the point where you can get things done quickly.

Acknowledgments

Python is built on the contributions from the user community, and some of the sections in this book are based on some of the excellent information available on the web. (Permission has been granted by the authors to reprint their contributions here.)

I especially want to thank the following people:

- Christiane Takacs helped me enormously by polishing the introductory statistics sections.
- Connor Johnson wrote a very nice blog explaining the results of the statsmodels OLS command, which provided the basis for the section on *Statistical Models*.
- Cam Davidson Pilon wrote the excellent open-source e-book *Probabilistic-Programming-and-Bayesian-Methods-for-Hackers*. From there, I took the example of the Challenger disaster to demonstrate Bayesian statistics.
- Fabian Pedregosa's blog on ordinal logistic regression allowed me to include this topic, which otherwise would be admittedly beyond my own skills.

I also want to thank Springer Publishing for the chance to bring out the second edition of this book, and to base the three introductory chapters (Python, Data Import, and Data Display) to a significant part on the corresponding chapters of my book *Hands-on Signal Analysis with Python*.

If you have a suggestion or correction, please send an email to my work address thomas.haslwanter@fh-ooe.at. If I make a change based on your feedback, I will add you to the list of contributors unless advised otherwise. If you include at least part of the sentence the error appears in, that makes it easy for me to search. Page and section numbers are fine, too, but not as easy to work with. Thanks!

Linz, Austria
August 2022

Thomas Haslwanter

Contents

Part I Python and Statistics

1	Introduction	3
1.1	Why Statistics?	3
1.2	Conventions	5
1.3	Accompanying Material	5
2	Python	7
2.1	Getting Started	8
2.2	Elements of Scientific Python Programming	15
2.3	Interactive Programming—IPython/Jupyter	28
2.4	Statistics Packages for Python	39
2.5	Programming Tips	43
2.6	Exercises	45
3	Data Input	49
3.1	Text	49
3.2	Excel	54
3.3	Matlab	54
3.4	Binary Data: NPZ Format	55
3.5	Other Formats	56
3.6	Exercises	56
4	Data Display	59
4.1	Introductory Example	59
4.2	Plotting in Python	62
4.3	Saving a Figure	66
4.4	Preparing Figures for Presentation	67
4.5	Display of Statistical Data Sets	70
4.6	Exercises	82

Part II Distributions and Hypothesis Tests

5 Basic Statistical Concepts	87
5.1 Populations and Samples	87
5.2 Data Types	89
5.3 Probability Distributions	90
5.4 Degrees of Freedom	94
5.5 Study Design	94
6 Distributions of One Variable	105
6.1 Characterizing a Distribution	105
6.2 Discrete Distributions	115
6.3 Normal Distribution	120
6.4 Continuous Distributions Derived from the Normal Distribution	125
6.5 Other Continuous Distributions	132
6.6 Confidence Intervals of Selected Statistical Parameters	135
6.7 Exercises	136
7 Hypothesis Tests	139
7.1 Typical Analysis Procedure	139
7.2 Hypothesis Tests and Power Analyses	144
7.3 Sensitivity and Specificity	152
7.4 Receiver-Operating-Characteristic (ROC) Curve	155
7.5 Exercises	157
8 Tests of Means of Numerical Data	159
8.1 Distribution of a Sample Mean	159
8.2 Comparison of Two Groups	164
8.3 Comparison of Multiple Groups	168
8.4 Summary: Selecting the Right Test for Comparing Groups	176
8.5 Exercises	178
9 Tests on Categorical Data	181
9.1 Proportions and Confidence Intervals	182
9.2 Tests Using Frequency Tables	183
9.3 Exercises	194
10 Analysis of Survival Times	197
10.1 Survival Distributions	197
10.2 Survival Probabilities	198
10.3 Comparing Survival Curves in Two Groups	202

Part III Statistical Modeling

11 Finding Patterns in Signals	205
11.1 Cross Correlation	205
11.2 Correlation Coefficient	208
11.3 Coefficient of Determination	211

11.4	Scatterplot Matrix	214
11.5	Correlation Matrix	214
11.6	Autocorrelation	217
11.7	Time-Series Analysis	218
12	Linear Regression Models	229
12.1	Simple Fits	230
12.2	Design Matrix and Formulas	232
12.3	Linear Regression Analysis with Python	237
12.4	Model Results of Linear Regression Models	241
12.5	Assumptions and Interpretations of Linear Regression	257
12.6	Bootstrapping	262
12.7	Exercises	262
13	Generalized Linear Models	265
13.1	Comparing and Modeling Ranked Data	265
13.2	Elements of GLMs	266
13.3	GLM 1: Logistic Regression	267
13.4	GLM 2: Ordinal Logistic Regression	270
13.5	Exercises	274
14	Bayesian Statistics	275
14.1	Bayesian Versus Frequentist Interpretation	275
14.2	The Bayesian Approach in the Age of Computers	277
14.3	Example: Markov-Chain-Monte-Carlo Simulation	278
14.4	Summing Up	280
Appendix A: Useful Programming Tools		283
Appendix B: Solutions		293
Appendix C: Equations for Confidence Intervals		321
Appendix D: Web Ressources		323
Glossary		325
Bibliography		331
Index		333

Abbreviations

ACF	Auto-Correlation Function
AIC	Akaike Information Criterion
ANOVA	ANalysis Of VAriance
ARIMA	Autoregressive Integrated Moving Average model
ARMA	Autoregressive Moving Average model
BIC	Bayesian Information Criterion
CDF	Cumulative Distribution Function
CI	Confidence Interval
CQ	Code Quantlet
DF/DOF	Degrees of Freedom
EOL	End Of Line
GLM	Generalized Linear Models
GUI	Graphical User Interface
HDF5	Hierarchical Data Format 5
HSD	Honest Significant Difference
HTML	HyperText Markup Language
IDE	Integrated Development Environment
IQR	Inter-Quartile-Range
ISF	Inverse Survival Function
ISO	International Organization for Standardization
ISP2e	https://github.com/thomas-haslwanger/statsintro-python-2e/tree/master/src/code_quantlets
JPEG	Joint Photographic Experts Group
KDE	Kernel Density Estimation
MCMC	Markov Chain-Monte Carlo
NAN	Not-A-Number
NPV	Negative Predictive Value
OLS	Ordinary Least Squares
PACF	Partial Auto-Correlation Function
PDF	Probability Density Function
PMF	Probability Mass Function

PNG	Portable Network Graphics
PPF	Percentile Point Function
PPV	Positive Predictive Value
QQ-Plot	Quantile-Quantile Plot
ROC	Receiver Operating Characteristic
RVS	Random Variate Sample
SARIMAX	Seasonal Autoregressive Integrated Moving Average Exogenous model
SD	Standard Deviation
SE/SEM	Standard Error (of the Mean)
SF	Survival Function
SQL	Structured Query Language
SS	Sum of Squares
SVG	Scalable Vector Graphics
TSA	Time Series Analysis
Tukey HSD	Tukey Honest Significant Difference test

Part I

Python and Statistics

The first part of the book presents an introduction to statistics based on Python. It is impossible to cover the whole language in thirty or forty pages, so if you are a beginner, please see one of the excellent Python introductions available on the Internet for details. Links are given below. This part is a kick-start for Python; it shows how to install Python under Windows, Linux, or MacOS, and walks step-by-step through documented programming examples. Tips are included to help avoid some of the problems frequently encountered while learning Python.

Because most of the data for statistical analysis are commonly obtained from text files, Excel files, or data preprocessed by Matlab, the third chapter presents simple ways to import these types of data into Python.

The last chapter of Part I illustrates various ways of visualizing data in Python. Since the flexibility of Python for interactive data analysis has led to a certain complexity that can frustrate new Python programmers, code samples for various types of interactive plots should help future Pythonistas to avoid these problems.

Chapter 1

Introduction



Statistics is the explanation of variance in the light of what remains unexplained (anon).

1.1 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 1550; 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 in Appendix.

Exercises: Solutions to the exercises provided at the end of most chapters can be found in Appendix. 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 (2018), which provides a sound, advanced treatment of statistical modeling.

WWW: On 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.

1.2 Conventions

The following conventions will be used.

- Text that is to be typed in on the computer is written in Courier font, e.g.,
`plot(x,y)`.
- Optional text in command-line entries is expressed with `<...>`,
e.g., `<InstallationDir>`.
- Names referring to computer programs and applications are written in italics, e.g.,
Jupyter.
- Italics will also be used when introducing new terms or expressions for the first time.
- Really important points that should definitely be remembered are indicated as follows:

This will be important!

1.3 Accompanying Material

All the examples and solutions shown in this book are available online. This includes code samples and example programs, Jupyter Notebooks with additional or extended information, as well as the data and Python code used to generate many of the figures. They can be downloaded from *GitHub*

<https://github.com/thomas-haslwanger/statsintro-python-2e>

which is organized in folders:

data	Raw data required for running the programs.
resources	Images used by this repository.
ipynbs	Jupyter Notebooks with examples, and with additional or extended information that goes beyond the content presented in the book.
src/exercise_solutions	Solutions to the exercises that are presented at the end of most chapters.
src/listings	Programs that are explicitly listed in this book.
src/figures	Code used to generate the Python figures in the text. Unless noted otherwise, the source code for Python figures is available in the source file F<chapter->_<figure->_xxxx.py. For example, Fig. 8.4 can be generated with the Python code in F8_4_anovaOneway.py
src/code_quantlets	Additional code samples.

In this book, references to the `src/code_quantlets`-directory in this repository will be abbreviated with `<ISP2e>`. Make sure also to look at the file `Errata.pdf` in the top folder of that archive, which will be kept up to date with corrections to any mistakes that are discovered after publication of the book.

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>` (here, the repository name of `statsintro-python-2e` given above) in a command terminal, and the whole repository—code as well as data—will be “cloned” to your system. Alternatively, you can download a ZIP-archive from there to your local system.

Chapter 2

Python



Python is free, consistently and completely object oriented, and has a large number of (free) scientific toolboxes (e.g., <http://www.scipy.org/>). It is used by Google, NASA, and many others. Information on Python can be found under <http://www.python.org/>. If you want to use Python for scientific applications, currently the best way to get started is with a Python distribution, either *WinPython*, or *Anaconda* from *Continuum Analytics*. These distributions contain the complete scientific and engineering development software for numerical computations, data analysis, and data visualization based on Python. They also come with *Qt* graphical user interfaces, and the interactive scientific/development environment Spyder. If you already have experience with *Matlab*, the article *NumPy for Matlab Users* (<https://numpy.org/devdocs/user/numpy-for-matlab-users.html>) provides an overview of the similarities and differences between the two languages.

Python is a very-high-level dynamic object-oriented programming language (Fig. 2.1). It is designed to be easy to program and easy to read. It was started in 1980, and has since gained immense popularity in a broad range of fields from web development, system administration, and in science and engineering. Python is open source, and has become one of the most successful programming languages. 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.



Fig. 2.1 The Python Logo

2.1 Getting Started

2.1.1 Distributions and Packages

The Python core distribution contains only the essential features of a general programming language. 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 packages for scientific applications are *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. And *pandas* has become widely adopted for statistical data analysis. It provides *DataFrames* which are labeled, 2D data structures, making work with data more flexible and intuitive. Python can be used with different front-ends (Fig. 2.2): from the command line or terminal; interactively, using Jupyter and IPython; and from “Integrated Development Environments” (IDEs).

IPython provides the tools for interactive data analysis, and *Jupyter* provides the different frontends for *IPython*. *IPython* lets you quickly display graphs and change directories, explore the workspace, provides a command history, etc.

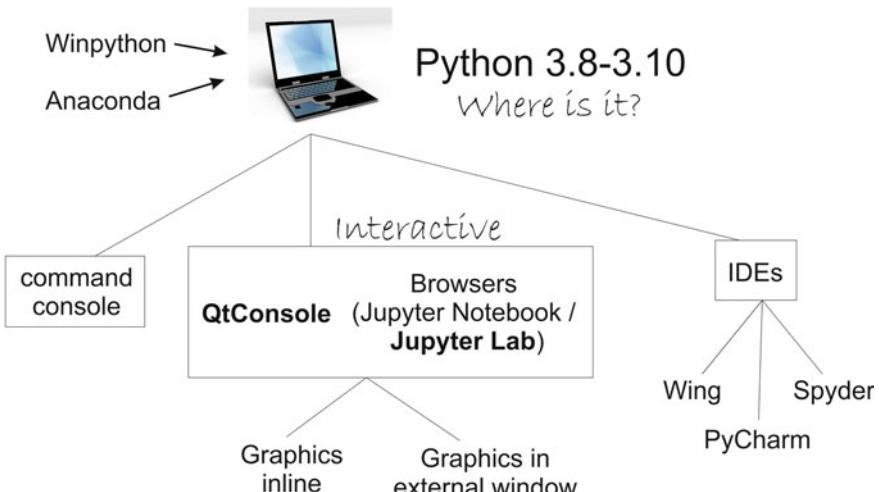


Fig. 2.2 Python can be used from different front-ends: from the command line (left), interactively (center), and from an IDE (right)

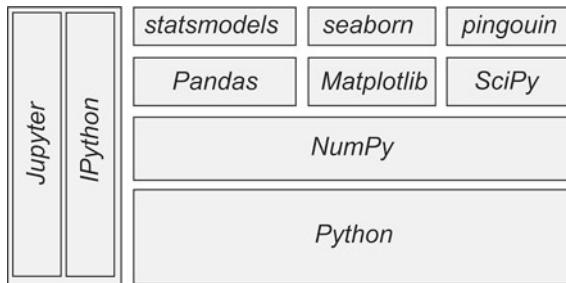


Fig. 2.3 The structure of the most important Python packages for statistics

Figure 2.3 shows the modular structure of the most important Python packages that are used in this book.

To facilitate the use of Python, 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 3.10.5
<https://winpython.github.io/>
- *Anaconda* by Continuum, for Windows, Mac, and Linux. The latest *Anaconda* version at the time of writing was 2020.11, with Python 3.9.
<https://www.anaconda.com/products/individual>

I am presently using *WinPython*, which is free and customizable. *Anaconda* also runs under Mac OS and Linux, and is free for educational purposes.

The Python code samples in this book expect a Python version ≥ 3.6 .

The programs included in this book have been tested under Windows and Linux. Under Windows, the following package versions have been used:

- *python 3.8.9* ... basic Python installation.
- *numpy 1.20.2+mkl* ... for working with vectors and arrays.
- *scipy 1.6.2* ... all the essential scientific algorithms.
- *matplotlib 3.4.1* ... the de facto standard module for plotting and visualization.
- *pandas 1.2.4* ... adds DataFrames (imagine powerful spreadsheets) to Python.
- *seaborn 0.11.1* ... statistical visualization package and visualization.
- *pingouin 0.4.0* ... easy-to-use statistics functions to perform the most widely used statistical tests.
- *statsmodels 0.12.2* ... for advanced statistical modeling.
- *ipython 7.22.0* ... for interactive work.
- *jupyter 1.0.0* ... for interactive work environments, e.g., the *JupyterLab*, *Jupyter Notebook*, or the *Qt console*.

All of these packages come with the *WinPython* and *Anaconda* distributions. Additional packages, which may be required by individual applications, can easily be installed using pip or conda.

a) PyPI—The Python Package Index

The Python Package Index (*PyPI*) (<https://pypi.org/>) is a repository of software for the Python programming language and contains more than 390'000 projects!

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
```

And to show information about a particular package type

```
pip show <package>
```

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

2.1.2 Installation of Python

While Python and the required packages can be installed manually, it is typically much easier to start out with a complete Python distribution.

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 can lead 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. (On my own system, I place all programs that do not modify the Windows Registry, such as *WinPython*, *vim*, and *ffmpeg*, into a folder C:\Programs.)
- After the installation, make a change to your *Windows Environment*, by typing Win -> env -> Edit environment variables for your account (Note that this is different from the system environment!):

- Add the directories
`<WinPythonDir>\python-3.8.9.amd64;`
`<WinPythonDir>\python-3.8.9.amd64\Scripts\;`
 (or whatever your Python version number is) to your PATH. (This makes Python and *IPython* accessible from the standard *Windows* command line, which can be reached quickly by typing Win+cmd.)
- Remove the default
`%USERPROFILE%\AppData\Local\Microsoft\WindowsApps` from the PATH (since it contains a misleading `python.exe-link`).
- 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://www.anaconda.com/distribution/>.
- Follow the installation instructions from the web page. 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 When I have had difficulties installing additional packages, I have been saved more than once by the pre-compiled packages Christoph Gohlke, available under <http://www.lfd.uci.edu/~gohlke/pythonlibs/>: from there you can download the <program>.whl file for your current version of Python, and then install it simply with `pip install <program>.whl`.

a) Under Linux

The following procedure works on *Linux Mint 20.1*:

- Download the most recent version of *Anaconda*.
- Open terminal, and navigate to the location where you downloaded the file to.
- Install *Anaconda* with `bash Anaconda<xx>-y.y.y-Linux-x86.sh`
- Update your Linux installation with `sudo apt-get update`.

Notes to Anaconda

- 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's installations tips:
<https://github.com/jrjohansson/scientific-python-lectures>

b) Under Mac OS X

- Go to <https://www.anaconda.com/distribution/>.
- 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 Qt console*, and the IDE *Spyder*.

To see which packages (e.g., *numpy*, *scipy*, *matplotlib*, and *pandas*) 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`.

2.1.3 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* user, 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 4.1.0*) from <http://cran.r-project.org/>, and install it into the `<RDir>` installation directory of your choice.

- 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-4.1.0  
- R_USER=<YourLoginName>
```

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

with Anaconda While *WinPython* comes with *rpy2* installed, *Anaconda* comes without *rpy2*. So after the installation of *Anaconda* and *R*, you should install *rpy2* with

- `conda install -c conda-forge rpy2`.

b) Under Linux

- After the installation of *Anaconda*, install *R* and *rpy2* with `conda install -c conda-forge rpy2`.

2.1.4 Python Resources

My favorite introductory book for scientific applications of Python is Scopatz and Huff (2015). However, that book does not provide any information on statistics. If you have some programming experience, the book you are currently reading may be all you need to get the statistical analysis of your data going. 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 for starting with Python:

- *Python Scientific Lecture Notes* If you don't read anything else, read this!
<http://scipy-lectures.org/>
- *NumPy for Matlab Users* Start here if you have Matlab experience
<https://numpy.org/doc/stable/user/numpy-for-matlab-users.html>
also check
<http://mathesaurus.sourceforge.net/matlab-numpy.html>
- *Lectures on scientific computing with Python* Great IPython notebooks, from JR Johansson.
<https://github.com/jrjohansson/scientific-python-lectures>
- *The Python tutorial* The official introduction
<http://docs.python.org/3/tutorial>
- *7 Steps to Python* My own efforts to smoothen the first steps into Python
https://work.thaslwanter.at/py_intro/

When running into a problem while developing a new piece of code, most of the time I just google; thereby, I stick primarily to the official Python documentation pages and to <http://stackoverflow.com>. Also, I have found Python user groups surprisingly active and helpful!

2.1.5 A Simple Python Program

a) Hello World

Python Shell 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 3.8.9 (tags/v3.8.9:0a7dcbd, May  3 2021, 17:27:52)...
Type "help", "copyright", "credits" or "license" for more...
>>> print('Hello World')
Hello World
>>>
```

However, most of the time it is more recommendable to start with the *IPython/Jupyter Qt console* described in more detail in Sect. 2.3. For example, the *Jupyter Qt console* is an interactive programming environment which offers a number of advantages. When you type `print` (in the *Qt console*, you immediately see information about the possible input arguments for the command `print`.

Python Modules are files with the extension `.py`, and are used to store Python commands in a file for later use. Let us create a new file with the name `helloWorld.py`, which contains the single line

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

This file can now be executed by typing `python helloWorld.py` on the command line.

On *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 local Python installation. On *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) square_me

To increase the level of complexity, let us write a Python module that includes a *function definition* and prints out the square of the numbers from zero to five. (More on functions in Sect. 2.2.5.) We call the file `L2_1_square_me.py`, and it contains the following lines:

Listing 2.1: square_me.py

```
1 # This file shows the square of the numbers 0-5.
2
3 def squared(x=10):
4     return x**2
5
6 for ii in range(6):
7     print(ii, squared(ii))
8
9 print(squared())
```

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. If the function is called with no input, `x` is by default set to `10`. This notation makes it very simple to define default values for function inputs.

Note: The range of the function is defined by the indentation! This is a feature loved by many Python programmers, but often found a bit 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. It tests if the function call with “`()`”, which uses the default parameter for `x`, also works, and prints the result.

Notes:

- Since Python starts at 0, the loop in *line 6* includes the six 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 Elements of Scientific Python Programming

Compared to the simple example above, real-world applications involve not only individual numbers but also vectors and matrices. These, together with the most important Python data- and file-structures, will be described in this section.

2.2.1 Python Datatypes

Python offers a number of powerful data structures, and it pays off to make yourself familiar with them. The most common ones are

- *Lists* to group objects of the same types.
- *Numpy Arrays* to work with numerical data. (numpy also offers the data type `np.matrix`. However, in my experience `np.array` is the way to go, since many numerical and scientific functions will not accept input data in `matrix` format.)
- *Tuples* to group objects of different types.
- *Dictionaries* for named, structured data sets.
- *pandas DataFrames* for simple import and export of data, and for statistical data analysis.

For simple programs, you will mainly work with lists and arrays. Dictionaries are used to group related information together. And tuples are used primarily to return multiple parameters from functions.

In the following, we will use for Python code the input/output formatting of *IPython* which will be presented in Sect. 2.3.

List [] Lists are typically used to collect items of the same type (numbers, strings, ...). They are “mutable”, i.e., their elements can be modified.

Note that “+” concatenates lists.

```
In [1]: myList = ['abc', 'def', 'ghij']

In [2]: myList.append('klm')

In [3]: myList
Out[3]: ['abc', 'def', 'ghij', 'klm']

In [4]: myList2 = [1,2,3]

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

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

Array [] *vectors* and *matrices*, for numerical data manipulation. They are defined in *numpy*. Note that vectors and 1D arrays are different: vectors CANNOT be transposed! With arrays, “+” adds the corresponding elements; and the array-method .dot performs a scalar multiplication. (Since Python 3.5, scalar multiplications can also be performed with the operator “@”.)

```
In [7]: import numpy as np
....: myArray2 = np.array(myList2)
....: myArray3 = np.array(myList3)

In [8]: myArray2 + myArray3
Out[8]: array([5, 7, 9])

In [9]: myArray2.dot(myArray3)
Out[9]: 32

In [10]: myArray2 @ myArray3
Out[10]: 32
```

Tuple () A collection of different things. Once created, tuples cannot be modified. (This really irritated me when I started to work with Python. But since I use tuples almost exclusively to return parameters from functions, this has not turned out to be any real limitation.)

```
In [11]: import numpy as np

In [12]: myTuple = ('abc', np.arange(0,3,0.2), 2.5)
In [13]: myTuple[2]
Out[13]: 2.5
```

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. It is defined in *pandas*. (See Sect. 2.2.4.)

2.2.2 Indexing and Slicing

The rules for addressing individual elements in Python lists, tuples, or *numpy* arrays have been nicely summarized by Greg Hewgill on *stackoverflow*¹:

```
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 up to end, by step
```

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

`start` or `end` may be a negative number. In that case the count goes 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. 2.4), and `a[-5:]` the last five elements (*World*).

¹ <http://stackoverflow.com/questions/509211/explain-pythons-slice-notation>.

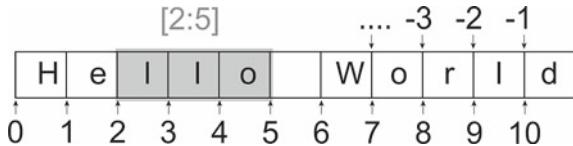


Fig. 2.4 The expressions in the top row indicate possible notations to select elements of a list or an array. Indexing starts at 0, and slicing does *not* include the last value

2.2.3 Numpy 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 as follows:

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

```
In [1]: import numpy as np

In [2]: np.zeros(3)
# by default numpy-functions generate 1D-vectors
Out[2]: array([ 0.,  0.,  0.])

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

np.ones generates numpy arrays containing ones.

np.random.randn generates normally distributed numbers, with a mean of 0 and a standard deviation of 1. To produce reproducible random numbers, you have to specify the starting point for the random number generation, for example, with `np.random.seed(...)`, using an integer number of your choice.

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]: xLow = np.arange(0, 3, 0.5)
```

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

In [7]: xLow
Out[7]: array([ 0.,  0.5,  1.,  1.5,  2.,  2.5])

In [8]: xHigh
Out[8]: array([ 3.,  3.5,  4.,  4.5])
```

np.linspace generates linearly spaced numbers

```
In [9]: np.linspace(0, 10, 6)
Out[9]: array([ 0.,  2.,  4.,  6.,  8.,  10.])
```

np.array generates a numpy array from given numerical data, and is a convenient notation to enter small matrices

```
In [10]: np.array([[1,2], [3,4]])
Out[10]: 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, the second element the second row, etc.:

```
In [11]: Amat = np.array([[1, 2],
                           [3, 4]])

In [12]: Amat[0]
Out[12]: array([1, 2])
```

Warning: A vector is not the same as a 1D matrix! This is one of the few features of Python that is not intuitive (at least to me), and can lead to mistakes that are hard to find. For example, vectors cannot be transposed, but matrices can.

```
In [13]: x = np.arange(3)

In [14]: Amat = np.array([[1,2], [3,4]])

In [15]: x.T == x
Out[15]: array([ True,  True,  True])
# This indicates that a vector stays a vector, and that
# the transposition with '''.T''' has no effect on
# its shape

In [16]: Amat.T == Amat
Out[16]: array([[ True, False],
                [False,  True]])
```

np.r_ Useful command to quickly construct small row vectors. But I only use it to try things out quickly. In my programs, I prefer the clearer but equivalent `np.array([...])`

```
In [17]: np.r_[1,2,3]
Out[17]: array([1, 2, 3], dtype=int32)
```

np.c_ Useful command to quickly build up small column vectors. Note that column-vectors can also be generated with the command `np.newaxis`:

```
In [18]: np.c_[[1.5,2,3]] # note the double brackets!
Out[18]: array([[1.5],
   [2. ],
   [3. ]])
```

```
In [19]: x[:, np.newaxis]
Out [19]: array([[0],
   [1],
   [2]])
```

np.atleast_2d Converts a vector (which cannot be transposed; see above) to the corresponding 2D array (which can be transposed):

```
In [20]: x = np.arange(5)
```

```
In [21]: x
Out[21]: array([0, 1, 2, 3, 4])
```

```
In [22]: x.T
Out[22]: array([0, 1, 2, 3, 4]) # no effect on 1D-vectors
```

```
In [23]: x_2d = np.atleast_2d(x)
```

```
In [24]: x_2d.T
Out[24]: array([[0],
   [1],
   [2],
   [3],
   [4]])
```

np.column_stack An elegant command to generate column matrices:

```
In [25]: x = np.arange(3)
```

```
In [26]: y = np.arange(3,6)
```

```
In [27]: np.column_stack( (x,y) )
Out[27]: array([[0, 3],
   [1, 4],
   [2, 5]])
```

2.2.4 pandas DataFrames

pandas (<http://pandas.pydata.org/>) is a widely used Python package, and provides data structures suitable for statistical analysis and data manipulation. It also adds

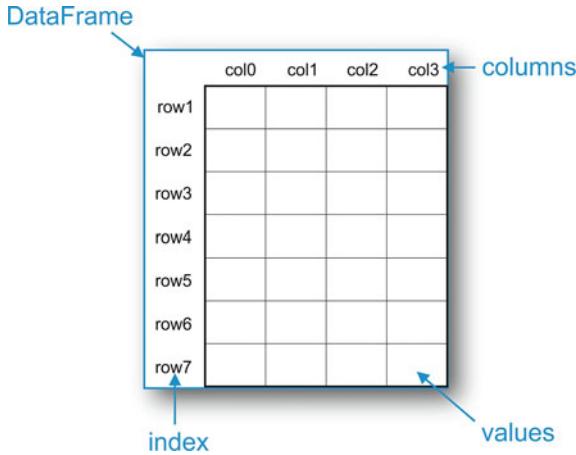


Fig. 2.5 pandas DataFrame

functions that facilitate data input, data organization, and data manipulation. *pandas* is commonly imported with

```
import pandas as pd
```

The official pandas documentation contains a very good “Getting started” section: https://pandas.pydata.org/docs/getting_started/.

a) Basic Syntax of DataFrames

Especially in statistical data analysis (read “data science”), *panelled data structures* (→ “pandas”) have turned out to be immensely useful. To handle such labeled data in Python, *pandas* introduces so-called “DataFrame” objects. A DataFrame is a 2D labeled data structure with columns of potentially different types. You can think of it as a spreadsheet or SQL table (see Fig. 2.5). DataFrames are the most commonly used *pandas* objects.

For statistical analysis, *pandas* becomes really powerful when combined with the package *statsmodels* (<https://www.statsmodels.org/>).

pandas DataFrames can have some distinct advantages over *numpy* arrays:

- A *numpy array* requires homogeneous data. In contrast, with a *pandas DataFrame* you can have a different data type (float, int, string, datetime, etc.) in each column (Fig. 2.6).
- *pandas* has built-in functionality for a lot of common data-processing applications: for example, easy grouping by syntax, easy joins (which are also really efficient in *pandas*), and rolling windows.
- DataFrames, where the data can be addressed with column names, can help a lot in keeping track of your data.

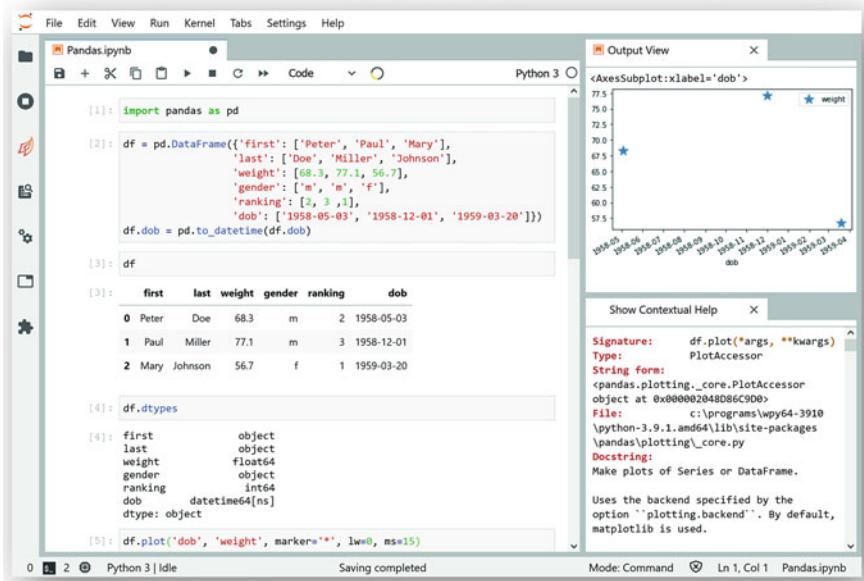


Fig. 2.6 Demonstration of some features of pandas DataFrames in *JupyterLab* (see Sect. 2.3.2). Unlike `np.array`, a `pd.DataFrame` can be used to combine different data types (here, strings, floats, integers, and dates: commands 2–4). It also can work efficiently with date and time data (commands 2 and 6, also note the output of “6” in the “Output View” on the right. This is a new feature of *JupyterLab*)

In addition, *pandas* has excellent tools for data input and output.

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']

```

To extract two columns at the same time, put the variable names in a list. With the following command, a new DataFrame `data` is generated, containing the columns Time and `y`:

```

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

```

After reading in the data, it is good practice to check if the data have been read in correctly. The first or last few rows can be displayed with

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

For example, the following statement shows the 5th – 10th row (note that these are 6 rows):

```
data[4:10]
```

as $10 - 4 = 6$. (I know, the array indexing takes some time to get used to. It helps me to think of the indices as pointers *to* the elements, and that they start at 0. see Fig. 2.4)

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]
```

The standard row/column notation can be used by applying the method `iloc`:

```
df.iloc[4:10, [0,2]]
```

Finally, sometimes one wants direct access to the data, not to the DataFrame. This can be achieved with

```
data.values
```

which returns a *numpy* array if all data have the same data type.

b) Note: Data Selection

While *pandas*' DataFrames are similar to *numpy* arrays, their philosophy is different. The *numpy* syntax comes from the mathematical description of n-dimensional matrices. In contrast, *pandas* has its origin in the data analysis of column-oriented database information. Some of the differences between the two that you should watch out for are

- *numpy* handles “rows” first. For example, `data[0]` is the first row of an array.
- *pandas* starts with the columns. For example, `df['values'][0]` is the first element of the column '`values`'.
- If a DataFrame has labeled rows, one can extract, for example, the row “row_label” with `df.loc['row_label']`. If one wants to address a row by its number, e.g., row number “15”, one can use `df.iloc[15]`. And `iloc` can be used 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]`. `df[5]` raises an error!



Code: `ISP_showPandas.py`² demonstrates some of *pandas* powerful functions for handling missing data, and for grouping and pivoting data items.

2.2.5 Functions, Modules, and Packages

Python has three different levels of modularization:

Functions are defined by the keyword `def`, and can be defined anywhere in Python. They return the object in the `return` statement, typically at the end of the function.

Modules are files with the extension `.py`. Modules can contain function- and variable- definitions, as well as valid Python statements.

Packages are folders containing multiple Python modules, and must contain 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

A function is a set of statements that take inputs, do some specific computation, and produce output. The idea is to group commonly or repeatedly done tasks and make a function, so that instead of writing the same code again and again for different inputs we can call the function. In Python, functions can be declared at any point in a program with the command `def`.

A short application example is given in Listing 2.2. Note that in the function definition so-called “type hints” are used (line 11) to indicate input and return type. They are optional, but make the code easier to read and understand.

Listing 2.2: `python_module.py`

```

1 """ Demonstration of a Python Function """
2
3 # author: Thomas Haslwanter
4 # date: June-2022
5
6 # Import standard packages
7 import numpy as np
8 from typing import Tuple
9
10
11 def income_and_expenses(data : np.ndarray) -> Tuple[float,
float]:
```

² [<ISP2e>/02_GettingStarted/ISP_showPandas.py](https://ISP2e/02_GettingStarted/ISP_showPandas.py).

```

12     """Find the sum of the positive numbers, and the sum of
13         the negative ones.
14
15     Parameters
16     -----
16     data : numpy array (,n)
17         Incoming and outgoing account transactions
18
19     Returns
20     -----
21     income : Sum of the incoming transactions
22     expenses : Sum of the outgoing account transactions
23     """
24
25     income = np.sum(data[data>0])
26     expenses = np.sum(data[data<0])
27
28     return (income, expenses)
29
30
31 if __name__=='__main__':
32     testData = np.array([-5, 12, 3, -6, -4, 8])
33
34     # If only real banks would be so nice ;
35     if testData[0] < 0:
36         print('Your first transaction was a loss and is
37             dropped.')
38     testData = np.delete(testData, 0)
39 else:
40     print('Congratulations: Your first transaction is a
41         gain!')
42
43     (my_income, my_expenses) = income_and_expenses(testData)
44     print(f'You have earned {my_income:.2f} EUR, ' +
45         f'and spent {-my_expenses:.2f} EUR.')

```

A detailed description of what happens in this piece of code is given below.
The example in Listing 2.2 shows how functions can be defined and used.

- **1:** Module header, commonly written as a multiline comment ("""<xxx>""").
- **3/4:** Author and date information (should be separate from the module header).³
- **7:** The required Python packages have to be imported explicitly. Here, *numpy* will be required, and it is customary to import *numpy* as *np*.
- **8:** The command *Tuple* from the package *typing* will be used in the “type hints” for the upcoming function. *Type hints* give hints on the type of the object(s) the function is using and for its return. They are optional, but improve the readability of code.
- **9/10:** Keep 2 empty lines before function definitions.

³ For the rest of the book, the “author/date” information will be left away, to keep the program listings more compact.

- **11:** Function signature.
- **12–23:** Multiline comment describing the function. It should also include information about the parameters the function takes, and about the return elements.
- **11–28:** Function definition. 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!
- **25:**
 - 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 **35**), or with a Boolean array (lines **25–26**).
- **28:** Python also uses round brackets to form groups of elements, so-called “tuples”. And the `return` statement does the obvious things: it returns elements from a function.
- **31:** Here, quite a few new aspects of Python are introduced:
 - Just like function definitions, `if`-loops or `for`-loops use indentation to define their context.
 - A convention followed by most Python coders is to prefix variables or methods that are supposed to be treated as a non-public part of the Python code with an underscore, for example, `_geek` or `__name__`.
 - Here, we check the variable with the name `__name__`, which is automatically generated by the Python interpreter and indicates the context of a module evaluation. If the module is run as a Python script, `__name__` is automatically set to `__main__`. But if a module is imported (see, e.g., Listing 2.3), 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). This is a nice way to test functions defined in the same module.
- **32:** Definition of a *numpy* array.
- **41:** The two elements returned as a tuple from the function `income_and_expenses` can be simultaneously assigned to two different Python variables, here to `(my_income, my_expenses)`.
- **42:** While there are different ways to produce formatted strings, the “f-strings” that were introduced with Python 3.6 are probably the most elegant: curly brackets `{ }` indicate values that will be inserted. The optional expression after the colon contains formatting statements: here `:5.2f` indicates “express this number as a

float, with 5 digits, 2 of which are after the comma”.⁴ The corresponding values are then passed into the f-string for formatted output. And the ‘\’ at the end of the line indicates a line continuation.

b) Modules

To execute the module `L2_2_python_module.py` from the command line, type `python L2_2_python_module.py`. In Windows, if the extension “.py” is associated with the Python program, it suffices to double-click the module, or to type `python_module.py` on the command line. In *WinPython*, the association of the extension “.py” with the Python function can be 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 [1]: %run L2_2_python_module
Your first transaction was a loss and is 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 path name.

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 `new_module.py`, and the function that we want from there `new_function`.

- `import new_module`: The function can then be accessed with `new_module.new_function()`.
- `from new_module import new_function`: In this case, the function can be called directly `new_function()`.
- `from new_module import *`: This imports all variables and functions from `new_module` into the current workspace; again, the function can be called directly with `new_function()`. However, use of this syntax is discouraged! It clutters up the current workspace, and one risks overwriting existing variables with the same name as an imported variable.

If you import a module multiple times, Python recognizes that the module is already known and skips later imports.

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

Listing 2.3: python_import.py

```
1 """ Demonstration of importing a Python module """
2
3 # Import standard packages
4 import numpy as np
5
6 # additional packages: this imports the function from above
```

⁴ <https://pyformat.info/> contains all the details of formatted output in Python.

```

7 import L2_2_python_module as py_func
8
9 # Generate test-data
10 testData = np.arange(-5, 10)
11
12 # Use a function from the imported module
13 out = py_func.income_and_expenses(testData)
14
15 # Show some results
16 print(f'You have earned {out[0]:5.2f} EUR, '+'\
17     f' and spent {-out[1]:5.2f} EUR.')

```

- **7:** The module `L2_2_python_module` (that we have just discussed above) is imported, as `py_func`.
- **13:** To access the function `income_and_expenses` from the module `py_func`, module- and function-name have to be given: `py_func.income_and_expenses(...)`. Note that `out` here contains both return-variables.

2.3 Interactive Programming—IPython/Jupyter

2.3.1 Workflow

The best way to start a program is to take a paper and pencil and explicitly write down the algorithms to be implemented! This helps to clarify the required programming steps, which parameters have to be provided explicitly, and which have to be calculated during the execution of the program. In most cases, this is also the most efficient way to start the development of a new program.

The next step is to work out the command syntax. In Python, this is best done with *IPython/Jupyter*. *IPython* (<http://ipython.org/>) 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 a lot of features that make it quick and easy to try out code.

Once the individual steps are working, one can use the *IPython* command `%history` to get the commands used. One can use either copy/paste that history, or save it directly to a file with

```
%history -f [fname]
```

Then one can switch to an integrated development environment (IDE), in my case *Wing*, to generate the final, working program.

The example in Fig. 2.7 shows the first steps for a program that generates a sine wave. Underlining the required parameters helps me to see which parameters need to be defined at the beginning of the program. And spelling out each step explicitly, e.g., the generation of a time-vector in line 4 in Fig. 2.7, clarifies which

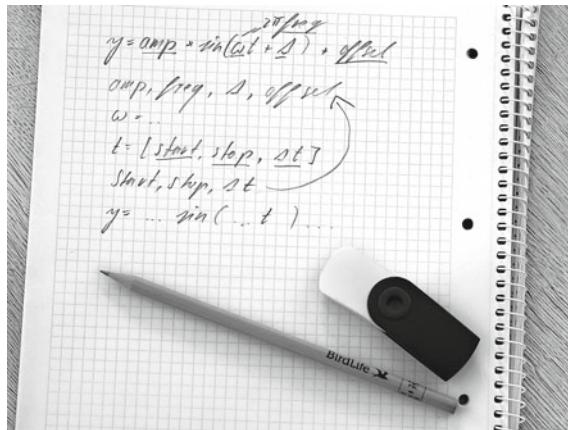


Fig. 2.7 Still, the best start to the successful development of a new program!

additional parameters arise in the program implementation. This approach speeds up the implementation of a program and is an important first step in avoiding mistakes.

2.3.2 Jupyter Interfaces

While IPython can also be run in a terminal-environment, its full power becomes available with *Jupyter*. In 2013 the *IPython Notebook*, a browser-based frontend for Python, became a very popular way to share research and results in the Python community. In 2015, the development of the frontend became its own project, called *Project Jupyter* (<https://jupyter.org/>). Today, *Jupyter* can be used not only with Python, but also with *Julia*, *R*, and more than 100 other programming languages.

The most important interfaces provided by Jupyter are

- *Qt console*.
- *Jupyter Notebook*.
- *JupyterLab*.

They can be started from a terminal with the command

```
jupyter <viewer>
```

where `viewer` is `qtconsole`, `notebook`, or `lab`.

a) Qt Console

The *Qt console* (see Fig. 2.8) is my preferred way to start coding, especially to figure out the correct command syntax. It provides immediate feedback on the command syntax, and good text completion for commands, file names, and variable names.

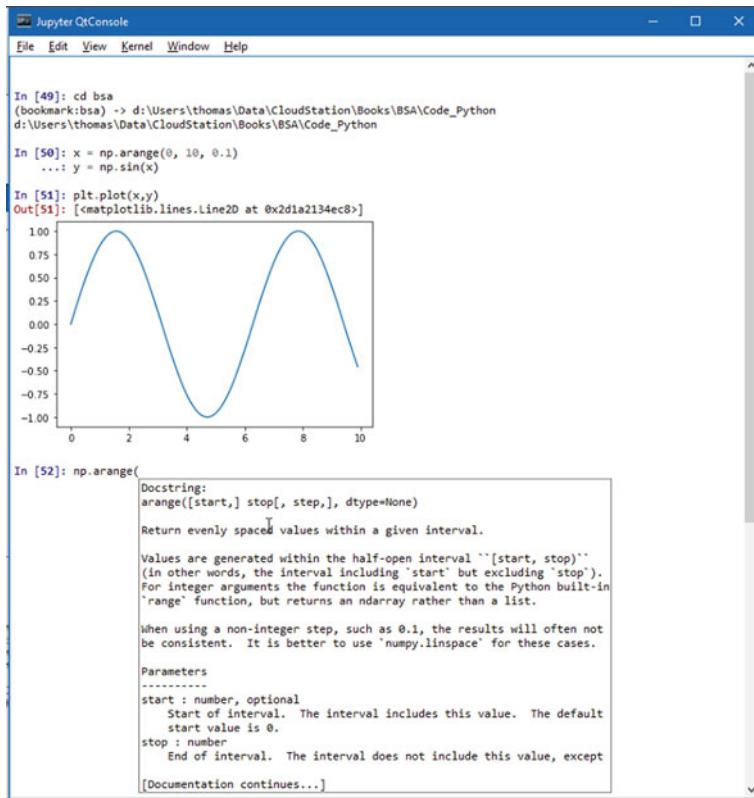


Fig. 2.8 The Qt console, displaying parameter tips for the current command, `np.arange`

b) Jupyter Notebook

The Jupyter Notebook is a browser-based interface, which is especially well suited for teaching, documentation, and collaborations. It allows you 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. 2.9). Packages such as *plotly* (<https://plot.ly/>) or *bokeh* (<https://bokeh.org/>) build on such browser-based advantages, and allow easy construction of interactive interfaces inside Jupyter Notebooks.

Code samples accompanying this book are also available as *Jupyter Notebooks*, and can be downloaded from

<https://github.com/thomas-haslwanter/statsintro-python-2e>.

c) JupyterLab

JupyterLab is the successor to the *Jupyter Notebook*. As Fig. 2.10 shows, it extends the *Notebook* with very useful capabilities such as a file browser, easy access to commands and shortcuts, and flexible image viewers. The file format stays the same as the *Notebook*, and both are saved as .ipynb-files.

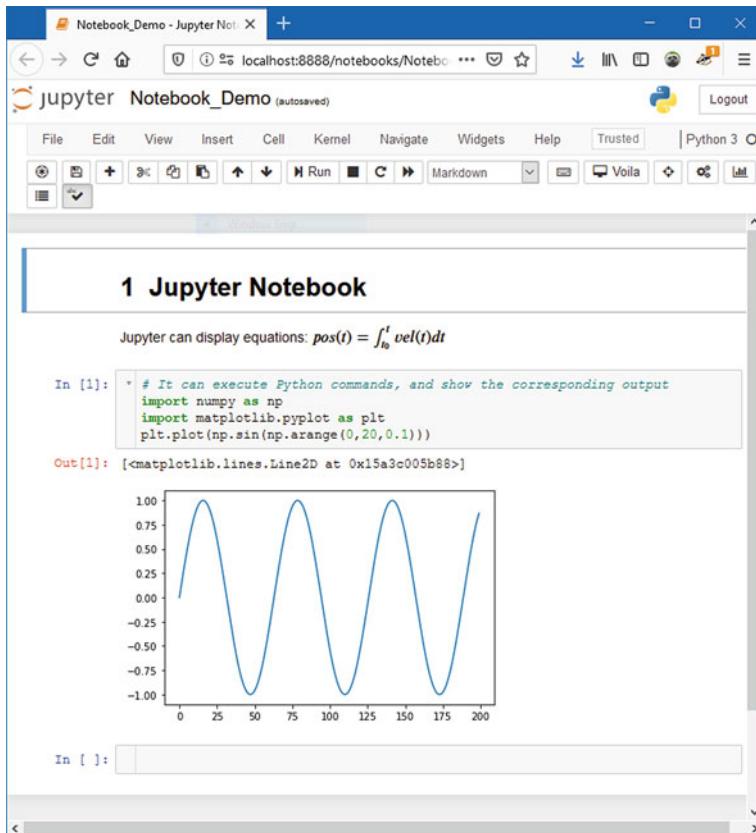


Fig. 2.9 The *Jupyter Notebook* makes it easy to share research, formulas, and results

2.3.3 Personalizing IPython/Jupyter

When working on a new problem, I always start out with the *Qt console* (see Fig. 2.8).

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 profile create.
(This creates the directory <mydir>\.ipython.)

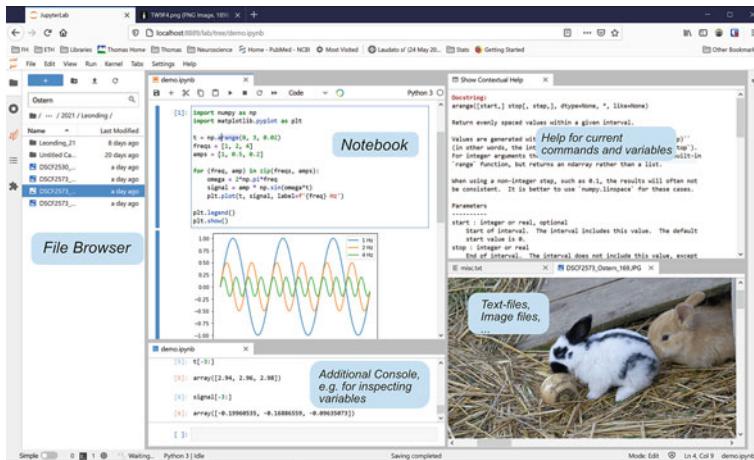


Fig. 2.10 *JupyterLab* is the successor to *Jupyter Notebook*. The blue labels indicate new features in *JupyterLab*

- Add the Variable `IPYTHONDIR` to your environment (see Sect. 2.1.2), 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 for example with the name `<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, which will import frequently used packages:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy import stats
```

- Generate a file “ipy.bat” in `<mydir>`, containing

```
jupyter qtconsole
```

- To customize the `jupyter qtconsole` type
`jupyter notebook -generate-config`.

This creates the file `jupyter_qtconsole_config.py` in your Jupyter folder. The Jupyter folder is in the subfolder `~/.jupyter` in your home directory. In this file you find multiple options to configure your Qt Console, e.g., the distance between commands, the editor used, and the header displayed at the program start.

(The same procedure can be used to customize the `jupyter notebook` and `jupyter lab`.)

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 lab
```

where `<ipynb-dir>` is the directory where all the Jupyter Notebooks are stored.

- 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 subfolder `.ipython/profile_default/startup`, place a file with, e.g., the name `00<myname>.py`, containing the lines

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
import pandas as pd
```

- 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 <ipynb-dir>
jupyter lab
```

- Make the file executable, with `chmod 755 ipynb.sh`.

You can now start “your” *IPython* by just typing `ipy`, and *JupyterLab* by typing `ipynb.sh`.

c) In Mac OS X

- 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*. 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* by default hides the folders that start with “.” (They can be shown with cmd-shift-.). So to access .ipython, open File -> Save as..... 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 should begin with 00-, so we could name it 00-<myname>..
- Open your .bash_profile (which contains the startup commands for your shell scripts), and enter the line
`alias ipy='jupyter qtconsole'.`
- To see all IPython Notebooks, do the following:
 - Go to <mydir>.
 - Create the file ipynb.sh, containing the lines

```
#!/bin/bash
cd <ipynb_dir>
jupyter lab
```

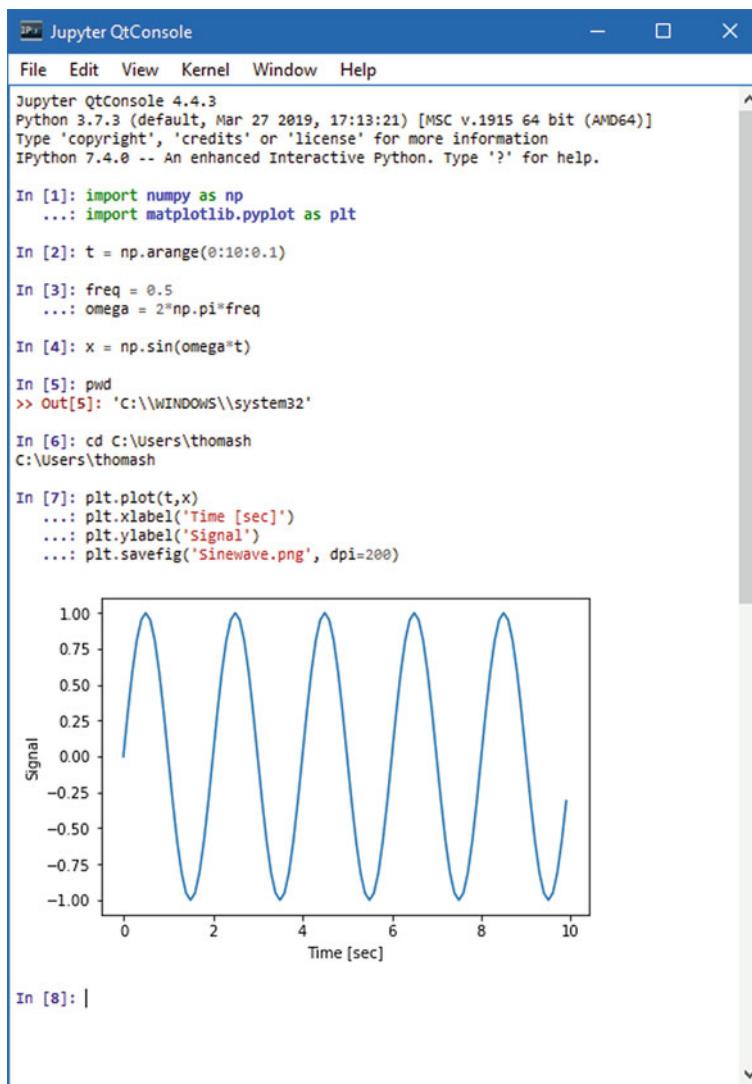
- Make the file executable, with chmod 755 ipynb.sh.

2.3.4 Sample Interactive Session

An important aspect of data analysis is interactive, visual inspection of the data. My personal preference for this, and for working out the syntax of the commands needed, is to start data analysis in the *Jupyter Qt console*.

In this example, I start my *IPython* sessions from the command line, with the command `jupyter qtconsole`. (Under *WinPython*: if you have problems starting *Jupyter* 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. 2.11:



The screenshot shows a Jupyter QtConsole window with the title "Jupyter QtConsole". The menu bar includes File, Edit, View, Kernel, Window, and Help. The main area displays an IPython session transcript and a generated plot.

```
Jupyter QtConsole 4.4.3
Python 3.7.3 (default, Mar 27 2019, 17:13:21) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.4.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import numpy as np
...: import matplotlib.pyplot as plt

In [2]: t = np.arange(0:10:0.1)

In [3]: freq = 0.5
...: omega = 2*np.pi*freq

In [4]: x = np.sin(omega*t)

In [5]: pwd
>> Out[5]: 'C:\\\\WINDOWS\\\\system32'

In [6]: cd C:\\Users\\thomash
C:\\Users\\thomash

In [7]: plt.plot(t,x)
...: plt.xlabel('Time [sec]')
...: plt.ylabel('Signal')
...: plt.savefig('Sinewave.png', dpi=200)
```

A plot is displayed below the code. The x-axis is labeled "Time [sec]" and ranges from 0 to 10. The y-axis is labeled "Signal" and ranges from -1.00 to 1.00. The plot shows a continuous blue sine wave oscillating between -1.00 and 1.00.

```
In [8]: |
```

Fig. 2.11 Sample session in the *Jupyter Qt console*

- *IPython* starts out listing the versions of *IPython* and Python that are used.
- **In [1]:** It is customary to import *numpy* as `np`, and *matplotlib.pyplot*, the *matplotlib* module containing all the plotting commands, as `plt`. Note that by hitting **CTRL+Enter**, one can execute multiline commands. (The command sequence gets executed after the next empty line.)
- **In [2]:** The command `t = np.arange(0,10,0.1)` generates a vector from 0 to 10, with a step size of 0.1. `arange` is a command in the *numpy* package.
- **In [3]:** Calculates `omega`. Note that the value of `pi` is only defined in *numpy*, and does not exist in Python!
- **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]:** The “*IPython* magic function” `pwd` stands for “print working directory”, and does just that. Tasks common with interactive computing, such as directory changes (`%cd`), bookmarks for directories (`%bookmark`), and inspection of the workspace (`%who` and `%whos`), are implemented as “*IPython* magic functions”. If no Python variable with the same name exists, the “%” sign can be left away, as here.
- **In [7]:** All the plotting commands are in the package `plt`. *IPython* generates plots by default in the *Jupyter Qt console*, as shown in Fig. 2.11. Generating graphics files is also 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 *Jupyter*, you can switch between inline graphs and output into an external Graphics window with `%matplotlib inline` and `%matplotlib qt5` (see Fig. 2.12). (Depending on your version of Python, you may have to replace `%matplotlib qt5` with `%matplotlib` or with `%matplotlib tk`.) An external graphics window allows zooming and panning, gets the cursor position (which can help to find outliers), and gets interactive input with the command `plt.ginput`. *matplotlib*’s plotting commands closely follow Matlab conventions.

2.3.5 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 number of Python conventions and syntax features.

For me, an efficient way to turn *IPython* commands into a script is to

- first obtain the command history with the command `%hist` or `%history`. (With the option `-f`, you can save the history directly with the desired filename.)
- copy the history into a good integrated development environment (IDE): my preferred IDE is *Wing* (<http://www.wingware.com/>), because it provides a very com-

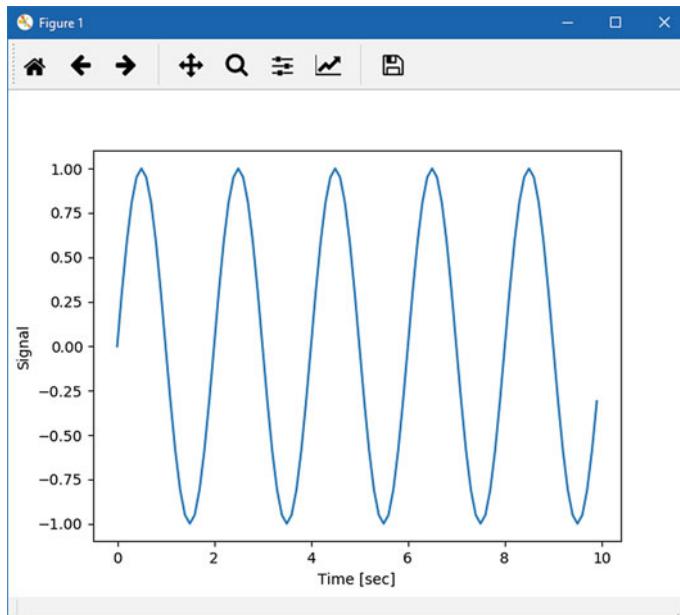


Fig. 2.12 Graphical output window, using the Qt-framework. This allows you to pan (keyboard shortcut p), zoom (o), go home (h), or toggle the grid (g). With plt.ginput(), one can also use it to get interactive input (from Listing 2.4)

fortable and powerful working environment, with integrated code versioning, testing tool, help-window, etc., and with a powerful debugger (Fig. 2.13). The latest version of *spyder*, a free, science-oriented IDE that comes installed with *anaconda* and with *WinPython*, is also really impressive (*spyder4*, <https://www.spyder-ide.org/>). Other popular and powerful IDEs are *pycharm* (<https://www.jetbrains.com/pycharm/>) and *Visual Studio Code* (<https://code.visualstudio.com/>).

- turn it into a working Python program by adding the relevant package information, substitute IPython magic commands, such as %cd, with their Python equivalent, and add more documentation.

Converting the commands from the interactive session in Fig. 2.11 into a program, we get

Listing 2.4: python_script.py

```
1 """ Short demonstration of a Python script.  
2 After a short one-line description of the content,  
3 the header can contain further details.  
4 """  
5  
6 # author: Thomas Haslwanter  
7 # date: June-2022  
8  
9 # Import standard packages
```

```

10 import numpy as np
11 import matplotlib.pyplot as plt
12
13 # Generate the time-values
14 t = np.arange(0, 10, 0.1)
15
16 # Set the frequency, and calculate the sine-value
17 freq = 0.5
18 omega = 2 * np.pi * freq
19 x = np.sin(omega * t)
20
21 # Plot the data
22 plt.plot(t,x)
23
24 # Format the plot
25 plt.xlabel('Time[sec]')
26 plt.ylabel('Values')
27
28 # Generate a figure, one directory up, and let the user know
29 # about it
30 out_file = '../Sinewave.jpg'
31 plt.savefig(out_file, dpi=200)
32 print(f'Image has been saved to {out_file}')
33
34 # Put it on the screen
35 plt.show()

```

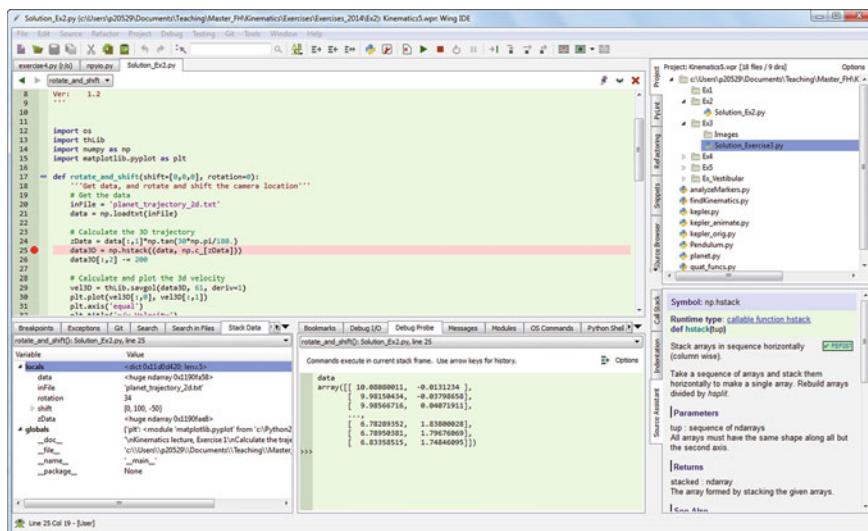


Fig. 2.13 Wing <https://www.wingware.com> is my favorite development environment, with one of the best existing debuggers for Python. **Tip:** If Python does not run right away in Wing, you may have to go to Project -> Project Properties and set the *Custom Python Executable* and/or *Python Path*

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

- The commands were put into a file with the extension .py, a so-called *Python module*.
- **1–4:** It is common style to precede a Python module with a “multiline” header block, even if the header has only a single line. Multiline comments are given between triple quotes " " " <xxx> " ". Below the first comment block describing the module, there should be the information about the author and date. (An excellent style-guide for Python can be found at <https://pep8.org/>.)
- **6:** Single-line comments use # .
- **29:** Care has to be taken with slashes in path names: both "/" and "\\" are perfectly valid path separators in Python. However, "\\\" is also used as the escape character in strings. To take "\" in a string literally, the string has to be preceded by "r" (for “raw string”), e.g., r'C:\\Users\\Peter', or it can be written as 'C:\\\\Users\\\\Peter'.
- **31:** f-strings were introduced in Python 3.6. With earlier versions, the corresponding syntax would be
`print('Image has been saved to 0'.format(out_file)).`
- **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. 2.12.

2.4 Statistics Packages for Python

2.4.1 Seaborn—Data Visualization

Seaborn (<https://seaborn.pydata.org/>) 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.

For example, the following code already produces a nice regression plot (Fig. 2.14), 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()
```

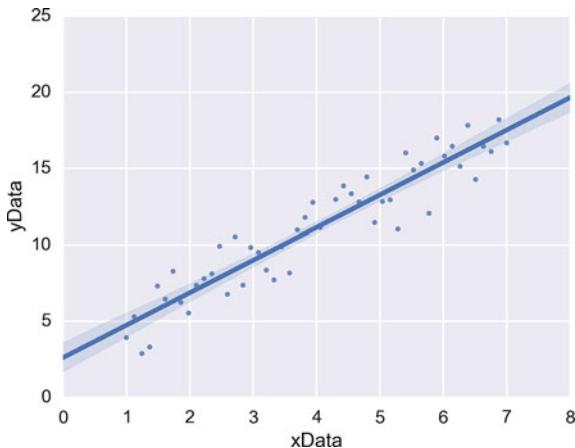


Fig. 2.14 Regression plot, from *seaborn*. The main axis shows the data, the best-fit line, and the confidence intervals for the fit

A more thorough overview of the plotting capabilities of *seaborn* is provided in the *Jupyter Notebook* on *seaborn plotting* in the github-archive of this book.⁵

2.4.2 *Pingouin*

While the sub-package `stats` from `scipy` provides low-level statistical functions, the recent package `pingouin` (<https://pingouin-stats.org/>) tries to offer simple yet exhaustive statistics functions. As an example, compare the linear regression fit to a noisy line:

```
In [1]: import numpy as np
....: import pingouin as pg
....: from scipy import stats

In [2]: np.random.seed(123)
....: x = np.arange(100)
....: y = 1.5*x + 50 + 10*np.random.randn(len(x))

In [3]: stats.linregress(x,y) # lineregress with scipy
Out[3]: LinregressResult(slope=1.5028351171729766,
intercept=50.130752434841256,
rvalue=0.9678058655187531,
pvalue=1.6044598942663455e-60,
stderr=0.039481127739603966,
intercept_stderr=2.2623409659387783)
```

⁵ https://github.com/thomas-haslwanter/statsintro-python-2e/blob/master/ipynbs/2_seaborn_plotting.ipynb.

```
In [4]: lm = pg.linear_regression(x,y) # linefit with pg
      ...: np.round(lm, 2)
Out[4]:
   names    coef      se       T   pval     r2  adj_r2    CI[2.5%]  CI[97.5%]
0 Intercept  50.13  2.26  22.16  0.0  0.94    0.94    45.64    54.62
1         x1    1.50  0.04  38.06  0.0  0.94    0.94    1.42    1.58
```

The output of pingouin presents the information in a much clearer and more useful way.

2.4.3 *Statsmodels—Tools for Statistical Modeling*

statsmodels is a Python package contributed to the community by the *statsmodels* development team (<https://www.statsmodels.org/>). It has a very active user community, and has in the last 12 years massively improved the suitability 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 is available for each estimator.

statsmodels also allows the formulation of models with the popular formula language based on the notation introduced (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 data set:

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as sm

# Generate noisy line, and save data in a pd-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() )
```

leads to the output (which will be discussed in detail in Chap. 12 on “Statistical Models”):

```

OLS Regression Results
=====
Dep. Variable:                      y      R-squared:                 0.996
Model:                             OLS      Adj. R-squared:            0.996
Method:                            Least Squares      F-statistic:             2.309e+04
Date:                Wed, 30 Jun 2021      Prob (F-statistic):       3.81e-118
Time:                    15:41:05      Log-Likelihood:           -135.77
No. Observations:                  100      AIC:                   275.5
Df Residuals:                      98      BIC:                   280.8
Df Model:                           1
Covariance Type:            nonrobust
=====
            coef    std err          t      P>|t|      [0.025      0.975]
-----
Intercept   -19.9691      0.189     -105.872      0.000     -20.343     -19.595
x           0.5001      0.003     151.943      0.000      0.494      0.507
=====
Omnibus:                     0.023      Durbin-Watson:        2.260
Prob(Omnibus):                0.988      Jarque-Bera (JB):    0.025
Skew:                         0.007      Prob(JB):              0.988
Kurtosis:                      2.924      Cond. No.               114.
=====
```

Another example would be a model that assumes that “success” is determined by “intelligence” and “diligence”, as well as the interaction of the two. With *patsy*’s formula language, such a model could be described by
`'success ~ intelligence * diligence'.`

An extensive list of result-statistics is 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 and Sample Size Calculations).
- Nonparametric Methods.
- Generalized Method of Moments.
- Empirical Likelihood.
- Graphics functions.
- A Data Sets Package.

2.5 Programming Tips

2.5.1 General Programming Tips

- Before you start programming, spell out the steps you have to do, and write them down as comments. A list of steps could look as follows:

```
# Set the parameters.  
# Select the input file.  
# Read in the data.  
# Analyze the data.  
# Show the results.  
# Save the results to an outfile.  
# Show the user the location of the outfile.
```

Not only does this help you to organize your code, but it also provides the first rudimentary documentation of the program.

- Data analysis is an interactive task. Make use of the powerful interactive programming environment offered by *IPython/Jupyter*, and first develop your analysis step by step in a *Qt console* or in *JupyterLab*.
- Once you have your data analysis—for the one block—going, grab the history with the command `history`, and turn it into a function. Think about what you want/need for the input, and what the output should be.
- And although I am repeating myself, before you implement a mathematical algorithm, write it down on paper! This makes the implementation much quicker, because you have to spell out explicitly what you want to do.
- Use the help provided by the package documentations (`numpy`, `matplotlib`, and `scipy`) and by <https://stackoverflow.com/>. (In the first step, restrict your search to these resources only: there are so many references and examples for Python on the web that it is very easy to get lost in them!)
- If possible, use some simple dummy data to test your program.
- Use clear variable names: it makes code much more readable, and easier to maintain in the long run.
- Know your editor well—you are going to use it a lot. Especially, know the keyboard shortcuts!
- Learn to use the debugger. Debuggers are immensely useful to track down execution errors in programs (see Sect. A.1). Personally, I always use the debugger from the IDE, and rarely resort to the Python built-in debugger `pdb`.
- Don’t repeat code. If you have to use a piece of code more than two times, write a function instead. The ideas of Python are nicely formulated in *The Zen of Python*, which you can see, for example, if you type in a Python console `import this`.

2.5.2 Python Tips

1. You should ALWAYS document the code that you write—even if you only hack a small program! I have been surprised how often I have had to go back and modify code that I thought I would “never need again”. And how often I then had a hard time understanding my own code if there were no comments included. A complete overview of the recommended best-practices in Python can be found under <https://pep8.org/>.
2. Stick to the standard conventions:
 - Every function should have a documentation string (in triple quotes """) 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 pingouin as pg
import seaborn as sns
```

3. To get the current directory, use `os.path.abspath(os.curdir)` or `os.path.abspath('.')`. And in Python modules, a change of directories can NOT be executed with `cd` (as in *IPython*), but instead requires the command `os.chdir(...)`.
4. Everything in Python is an object: to find out about “obj”, use `type(obj)` (to find out the data type) and `dir(obj)` (to find out the methods and properties of the object).
5. Use functions to avoid the duplication of code, and understand the `if __name__=='__main__':` construct (see p. 27).
6. If you have many of your personal functions in a directory `mydir` that is different from the current working directory, you can add that directory to your `PYTHONPATH` with the command

```
import sys
sys.path.append('mydir')
```

7. Make sure you know the basic Python syntax, especially the data structures. Try to use matrix multiplications instead of loops wherever possible: this makes the code nicer, and the programs much faster.
8. And along the same lines: note that many commands use an `axis` parameter, and can act on rows, columns, or all data:

```
In [1]: mat = [[1, 2],
             [3, 4]]
```

```
In [2]: np.max(mat)
Out[2]: 4
```

```
In [3]: np.max(mat, axis=0)
Out[3]: array([3, 4])

In [4]: np.max(mat, axis=1)
Out[4]: array([2, 4])
```

2.5.3 IPython/Jupyter Tips

1. Use IPython in the Jupyter *Qt console* or *JupyterLab*, and customize your startup as described in Sect. 2.3.3: 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. Also, check out the help tips shown with the command `%quickref`. In *JupyterLab*, you can
 - get help on the current command with Shift+Tab.
 - get contextual help with Ctrl+I (you can move the new tab to a separate console by simply click-dragging the tab-header with the mouse).
 - Ctrl+Shift+C gives a list of all commands, and the corresponding keyboard shortcuts.
3. Use TAB-completion, for file- and directory-names, variable names, and Python commands. This speeds up the coding, and helps to reduce typing mistakes.
4. To switch between inline and external graphs, use
`%matplotlib inline` and `%matplotlib` or `%matplotlib qt5`.
5. You can use `edit <fileName>` to edit files in the local directory, and
`%run <fileName>` to execute Python scripts in your current workspace.
6. The command `%bookmark` lets you quickly navigate to frequently used directories.

2.6 Exercises

1. Translating Points

Write a Python script that

- specifies two points, $P_0 = (0/0)$ and $P_1 = (2/1)$. Each point should be expressed as a Python `list([a,b])`,
- combines these two points to an `np.array`,
- shifts those data, by adding `3` to the first coordinate, and `1` to the second,
- plots a line from the original P_0 to the original P_1 , and on the same plot also plot a line between the shifted values.

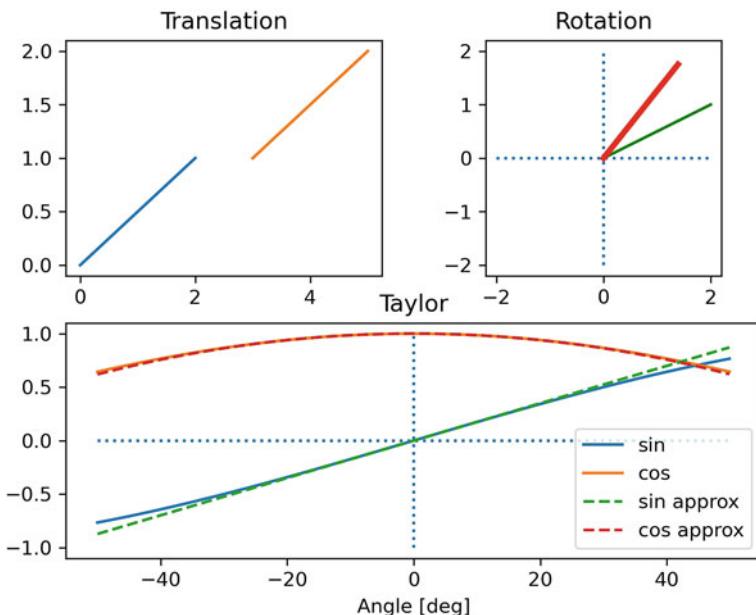


Fig. 2.15 Results of the first three exercises

The result is shown in Fig. 2.15.

More information on data display is presented in Chap. 4.

2. Rotating a Vector

Write a Python script that specifies two points, $P_0 = (0/0)$ and $P_1 = (2/1)$. Then write a Python function that

- takes a vector and an angle as input parameters,
- rotates the vector by 25 degrees by multiplying it with the rotation matrix \mathbf{R} ,
- and returns the rotated vector.

Tip A 2D rotation matrix is defined by

```
R = np.array([[np.cos(alpha), -np.sin(alpha)],
              [np.sin(alpha), np.cos(alpha)]])
```

If you want to experiment a bit with plots, you can try to

- plot a green line from P_0 to P_1 ,
- superpose this plot with a coordinate system, from -2 to $+2$,
- superpose the rotated line in red, with increased line-thickness. (You can modify the width of a line with the plot parameter “`linewidth=`”).

3. Taylor Series

- Write a function that calculates the approximation to a sine and a cosine, to second order.
- Write a script which plots the exact values, and superposes them with approximate values, in a range from -50 deg to $+50$ deg. (Command `plt.xlim`)
- Save the resulting image to a PNG-file.

Tip The second order approximations to sine and cosine are given by

$$\sin(\alpha) \approx \alpha$$

$$\cos(\alpha) \approx 1 - \frac{\alpha^2}{2}.$$

4. First Steps with pandas

- Generate a *pandas* DataFrame, with the x-column time stamps from 0 to 10 sec, at a rate of 10 Hz, the y-column data values with a sine with 1.5 Hz, and the z-column with the corresponding cosine values. Label the x-column `time`, the y-column `yvals`, and the z-column `zvals`.
- Show the head of this DataFrame.
- Extract the data in rows 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.

Tip A good, concise introduction into pandas is

https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html

Chapter 3

Data Input



This chapter shows how to read different types of 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 data analysis.

Text data input can be complicated by a number of problems, like different separators between data entries (such as spaces and/or tabs), or empty lines at the end of the file. In addition, data may have been saved in different formats, such as *MS Excel*, *HDF5* (which also includes the Matlab format), or in databases. This chapter gives an overview of where and how to start with data input.

3.1 Text

3.1.1 Visual Inspection

Reading in simple ASCII-text sounds like a trivial thing to do. A number of Python tools have been developed for data input. But regardless of which tool you use, you should always—before trying to read in the data—check the following:

- 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 tabs and/or by spaces? (Tip: you should use a text-editor which can visualize tabs, spaces, and end-of-line (EOL) characters. Do *not* use *MS Excel* to inspect text files, since the representation of numbers in Excel depends on the “Region Settings” of your computer!)

The diagram shows a table structure with various parts labeled:

- [Header]
- [Column-Header]
- [Row - Header]
- [Footer]
- Header row: Math, Music, Group
- Data rows:
 - Doris: Math (1.5), Music (3.5), Group (A)
 - Peter: Math (2.0), Music (NaN), Group (A)
 - Mary: Math (2.5), Music (empty), Group (A)
 - Paul: Math (1.5), Music (2.0), Group (A)
- Footer row: Footer

Annotations point to specific cells:

- Pointing to '3.5' in the Math column of the Doris row: "numbers and strings"
- Pointing to 'NaN' in the Music column of the Peter row: "'not-a-number'"
- Pointing to the empty cell in the Music column of the Mary row: "missing entries"
- Pointing to the empty cell in the Music column of the Paul row: "delimited with tabs and/or white-spaces"
- Pointing to the 'A' in the Group column of the Paul row: "decimal separator can be ‘.’ or ‘,’"

Fig. 3.1 ASCII-files come in many varieties. This figure shows some of the possible pitfalls when trying to read in text data

- Are there missing values, and are they indicated consistently?
- Is the data type in each variable (column) consistent?

And after the data have been read in, check the following:

- Have the data in the first line been read in correctly?
- Have the data in the last line been read in correctly?
- Is the number of columns correct?

Figure 3.1 shows some of the variable aspects that commonly occur in text-files.

3.1.2 Reading ASCII-Data

I strongly suggest that you start your data analysis by reading in and inspecting the 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 found the correct command syntax to read in the data, you can obtain the command history with `%history`, copy it into your favorite IDE, and turn it into a program as described in the previous chapter.

While the *numpy* command `np.loadtxt` allows to read in simply formatted text data, most of the time it is easier to use the *pandas* command `pd.read_csv` as it provides significantly more powerful options for data entry.

A typical workflow can contain the following steps:

- changing to the folder where the data are stored,
- listing the files in that folder,

- selecting one of these files, and reading in the corresponding data, and
- checking if the data have been read in completely, and in the correct format.

These steps can be implemented in *IPython*, for example, 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]: in_file = 'data.txt' # Don't hard-code parameters
In [6]: df = pd.read_csv(in_file)
In [7]: df.head()   # Check if first line is OK
In [8]: df.tail()   # Check the last line
```

After “In [6]”, I often have to adjust the options of `pd.read_csv` to read in the data correctly. Make sure to 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]:
   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*. Note that the *pandas* function `pd.read_csv` already recognizes the first column as integer, whereas the second and third columns

are correctly identified as float. Without the flag `header=None`, the entries of the first row are falsely interpreted as the column labels as shown in the next step:

```
In [13]: df = pd.read_csv('data.txt')    # Warning:
                                                    # Incorrect result!
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
```

b) More Complex Text-Files

The advantage of using `pandas` for data input becomes clear with more complex files. Take, for example, an input file `data2.txt` containing the following lines (including the footer):

```
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.
May, 2020

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('data2.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
```

Tip: an option of the command `pd.read_csv` that is frequently useful is `delim_whitespace`. When this parameter is set to `True`, one or more white-spaces (spaces or tabs) are taken as a single separator.

3.1.3 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, and their syntax is independent of the programming language used. They are directly supported by most programming languages. Many books have been written about them, and good, concise information on regular expressions can be found on the web, for example,

- <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.

Two examples can demonstrate 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 elements inside the brackets.

And the star (“*”) indicates *one or more* of the preceding elements.

2. Extracting columns with certain name-patterns from a *pandas* DataFrame. In the following example, all columns starting with `Vel` are extracted and combined:

```
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 Excel

There are two approaches to reading a *MS 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 = 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*.

3.3 Matlab

The best input solution for *Matlab* files depends on the complexity of the files. For .mat files which contain only strings, numbers, vectors, and matrices, the easiest solution is `scipy.io.loadmat`.

The following commands return a string, a number, a vector, and a matrix variable from a Matlab file `data.mat`, by reading them into a Python dictionary.

¹ The following section has been taken from the *pandas* documentation.

```

from scipy.io import loadmat

matlab_file = 'data.mat'
data = loadmat(matlab_file, squeeze_me=True)

# Field names are the names of the Matlab variables
text = data['my_text']
number = data['float_number']
vector = data['my_vector']
matrix = data['my_matrix']

```

If the .mat file also contains cells and structures, but no more complex data structures (e.g., arrays with more than 2 dimensions or with complex numbers, and sparse arrays), then the package `mat4py` is great. Note that `mat4py` is typically not included in the common Python distributions, and thus has to be installed by hand (`pip install mat4py`). It returns the data in simple Python data types (specifically it returns arrays as `list` and not as `np.array`).

```

import mat4py

data = mat4py.loadmat(matlab_file)
array_data = np.array( data['my_matrix'] )

cell = data['my_cell']

```

 **Code:** `ISP_matlab_data.py`² shows different ways to read in data .mat-files.

3.4 Binary Data: NPZ Format

If space and/or bandwidth is at a premium or for large data sets it may be desirable to save data in binary format, in that case more care is required, since binary representations offer many different options. Two common useful options are the .npz file format provided by `numpy`, or so-called “structured arrays”. The former one is useful if the data are read in by Python programs again. The latter one is preferable if the data should be further processed by other applications.

The example below shows how to save data to .npz format, which is a zipped archive of files named after the variables they contain. The archive is not compressed and each file in the archive contains one variable in .npz format.

```

In [1]: import numpy as np
In [3]: t = np.arange(0, 10, 0.1) # Generate np arrays
In [4]: x = np.sin(t)
In [5]: data_dict = {'time': t, 'signal': x} # dictionary
In [6]: out_file = 'binary'
# Save dict to '.npz', and then re-load '.npz' file

```

² [ISP2e>/03_DataInput/ISP_matlab_data.py](#).

```
In [7]: np.savez(out_file, **data_dict)
In [8]: new_data = np.load(out_file + '.npz')
# Use loaded data
In [9]: plt.plot(new_data['time'], new_data['signal'])
In [10]: new_dict = dict(new_data) # Convert to dict
```

3.5 Other Formats

Clipboard data from the system clipboard can be imported 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.

Zipped Archives on the WWW Python also allows reading in data directly from the web, even if they are zipped. See `ISP_read_zip.py`.

 **Code:** `ISP_read_zip.py`³ shows how to directly read in data from an *Excel* file which is stored in a zipped archive on the web.

3.6 Exercises

1. Reading in Data

Read the data from the following files into your workspace. (If the files are not yet available, they can be generated by running the script `S3_data_gen.py`.)

- data.csv : Comma-separated data file
- data_tab.txt : Tab-separated data file
- data_modified.txt : Tab-separated data file, with header
- data.xls : Excel file
- data.mat : Matlab file
- data.raw : Binary data file

³ [ISP2e>/03_DataInput/ISP_read_zip.py](#).

2. Modifying Text Files: Imaginary Numbers (hard)

The file `imaginary.txt` contains the real- and the imaginary parts of complex numbers, including a header for each column. Write a Python script that reads in those data, and adds the polar representation of each data point (radius/angle [rad]) to each line, with the name of the out-file `imaginary_out.txt`.

Tip: For the resulting out-file, separate header labels and numbers by a simple tab.

3. Mixed Inputs

- The file `.\data\swim100m.csv` contains values and strings. Read in the data, and show the first 5 and the last 5 data points.
- The MS Excel file `.\data\Table 2.8 Waist loss.xls` contains some data lower down in the file. Read in the data, and show the last 5 data points.
- **Hard:** Read in the same file, but this time from the zipped archive <https://work.thaslwanter.at/sapy/GLM.dobson.data.zip>.

4. Binary Data

The file `data.raw` has a 256-byte header, followed by triplets of data (`t`, `x`, `y`) stored in `float` representation. Read in those data, and plot `x` and `y` versus `t`.



Code: How to write data in different formats, and how to produce formatted text strings, is shown in `S3_data_gen.py`. That script also produces the input files for the exercises for this chapter.

Chapter 4

Data Display



This chapter presents the basic concepts of plotting in Python. It also provides help in turning Python plots into good-looking figures for presentations. Examples of different 2D and 3D plot types provide the first look into the capabilities of the plotting package *matplotlib*.

We will start out with the movement of a particle along a simple trajectory, showing the position and velocity of a particle that moves along a figure-eight path (see Fig. 4.1) as a function of time.

Based on the position and velocity of such a particle, the next section then explains the basic concepts for generating plots in Python. It also presents code samples for a number of helpful features, such as positioning figures on the computer screen, or querying keyboard input for figures. The last section describes how Python figures can be exported to some common, vector-based graphics programs, to facilitate the preparation of figures for different types of presentation formats.

4.1 Introductory Example

The code in Listing 4.1 produces the plot in Fig. 4.2.

Listing 4.1: simple_figure.py

```
""" Basic plotting commands, by showing position and velocity
of two curves """

# Import standard packages
import numpy as np
import matplotlib.pyplot as plt
from datetime import date
```

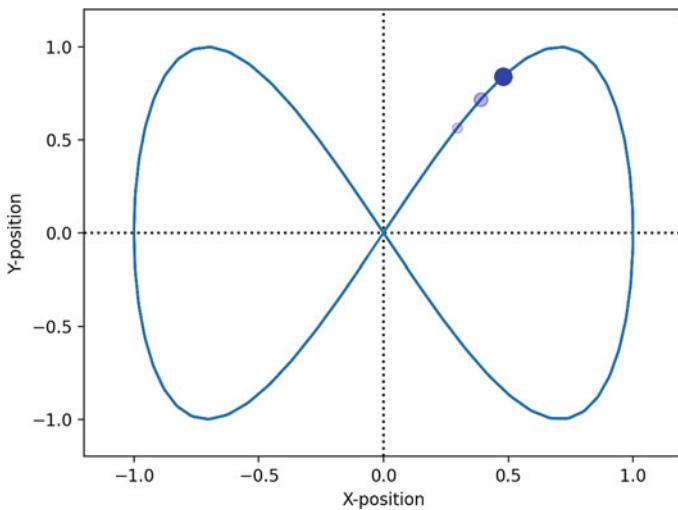


Fig. 4.1 x-y trajectory of a particle moving along a figure-eight

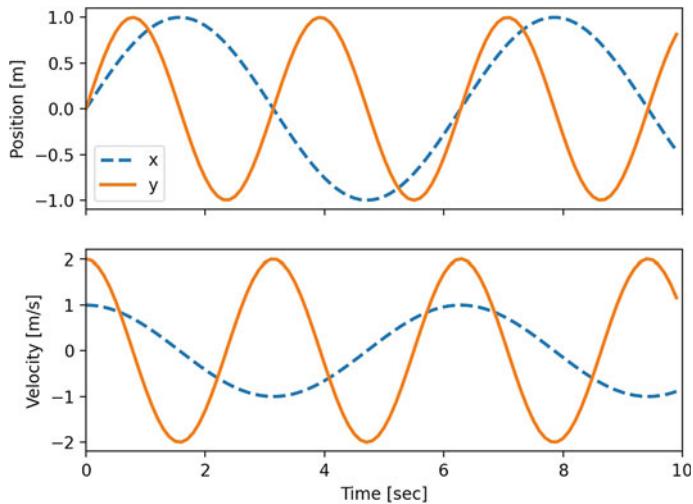


Fig. 4.2 Simple demo figure (from Listing 4.1)

```
# Generate the data
t = np.arange(0,10,0.1)

x = np.sin(t)          # position
y = np.sin(2*t)        # velocity

vx = np.cos(t)
vy = 2*np.cos(2*t)
```

```
# Put position in top and velocity in bottom plot
# Ensure that zooming into one adjusts both plots
fig, axs = plt.subplots(nrows=2, ncols=1,
                       sharex=True)

# Here a plot-parameter is modified
# 'linewidth=' and 'lw=' are equivalent
axs[0].plot(t, np.column_stack([x,y]), linewidth=2)
axs[1].plot(t, np.column_stack([vx,vy]), lw=2)

# Add the axis labels
axs[0].set_ylabel('Position [m]')
axs[1].set_xlabel('Time [sec]')
axs[1].set_ylabel('Velocity [m/s]')

# Set the x-limit (Note that since the x-axes are
# shared, we only have to do this once!
axs[0].set_xlim([0, 10])

# Also put the date on the figure
fig.text(0.8, 0.02, date.isoformat(date.today()))

# Properties of figure elements can also be changed
# after they have been drawn:
for ax in axs:
    lines = ax.get_lines()
    lines[0].set_linestyle('dashed')

# Add a legend to the first axis
axs[0].legend(['x', 'y'])

# Save the figure
out_file = 'simple_figure.jpg'
plt.savefig(out_file, dpi=200)

# Always inform the user if any file has been added
# or modified on the computer
print(f'Image saved to {out_file}')

plt.show()
```

The following basic rules can help to optimize figures:

- Illustrations should contain as few elements as possible, but as many as necessary (Fig. 4.2).
- Always label the axes, and include axes units in the labels!!
- Minimize the amount of reading the user has to do. For example,
 - If two axes are above each other with the same x-scale, only use x-tick-labels for the lower axis.
 - If two axes have the same line style and labels, only use one legend.

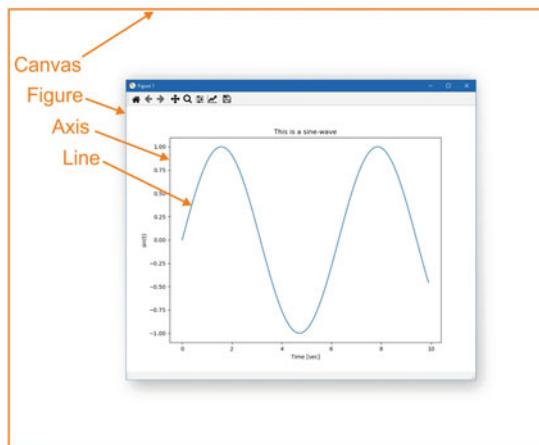


Fig. 4.3 Basic elements of a figure in *matplotlib*. The canvas can be a figure window, but it can also be a PDF file, or a section in a browser. It only needs to be addressed rarely, for example, to position the figure window on the computer screen

- Consider putting a date on your figures. This may help you later on to interpret them—especially if you modify some parameters in the data analysis.
- Separate the generation of the figure elements from the formatting (see next section). This helps to clarify the code.
- When saving a figure to a file, always tell the user where a file has been generated/modified, and the name of that file!

4.2 Plotting in Python

The first step in data analysis should always be a visual inspection of the raw data.

The dominant task of the human cortex is to extract visual information from the activity patterns on the retina. Our visual system is therefore exceedingly good at detecting patterns in visualized data sets. As a result, one can almost always *see* what is happening before it can be demonstrated through quantitative analysis of the data. Visual data displays are also helpful in finding extreme data values, which are often caused by mistakes in the execution of the paradigm or mistakes in the data acquisition.

In practice, the display of data can be tricky, as there are so many options: graphical output can be displayed as a picture on an HTML-page or in an interactive graphics window; plots can demand the attention of the user (so-called “blocking figures”), or can automatically close after a few seconds, etc. This section will therefore focus on general aspects of plotting data; the next section will then present different types of plots, e.g., histograms, error bars, and 3D-plots.

The Python core does not include any tools to generate plots. This functionality is added by other packages. By far, the most common package for plotting is *matplotlib*. If you have installed Python with a scientific distribution like *WinPython* or *Anaconda*, it will already be included. *Matplotlib* is intended to mimic the style of Matlab. As such, users can either generate plots in a functional style (“Matlab style”), or in the traditional, object-oriented Python style (see below).

Matplotlib (<https://matplotlib.org/>) contains different modules and features:

matplotlib.pyplot This is the module that is commonly used to generate plots. It provides the interface to the plotting library in *matplotlib*, and is by convention imported in Python functions and modules with `import matplotlib.pyplot as plt`.

pyplot handles lots of little details, such as creating figures and axes for the plot, so that the user can concentrate on data analysis.

matplotlib.mlab Contains a number of functions that are commonly used in Matlab, such as `find` and `griddata`.

backends *matplotlib* can produce output in many different formats, which are referred to as “backends”:

- In a *Jupyter Notebook* or in a *Jupyter QtConsole*, the command `%matplotlib inline` directs output into the current browser window.
- In the same environment, `%matplotlib qt51` directs the output into a separate graphics window (Fig. 2.12). This allows panning and zooming the plot, and interactive selection of points on the plot by the user with the command `plt.ginput`.
- With `plt.savefig`, output can be easily directed to external files, e.g., in PDF, PNG, or JPG format.

pylab is a convenience module that imports `matplotlib.pyplot` (for plotting) and `numpy` (for mathematics and working with arrays) in bulk in a single name space. Although many examples use *pylab*, it is no longer recommended.

The easiest way to find an implementation of one of the many image types that *matplotlib* offers is to browse the *matplotlib* gallery (<https://matplotlib.org/stable/gallery/index.html>), and copy the corresponding Python code into your program.

Other interesting plotting packages and resources are

- Also, *pandas* (which builds on *matplotlib*) offers many convenient ways to visualize DataFrames, because the axis- and line-labels are already defined by the column names.
https://pandas.pydata.org/pandas-docs/stable/user_guide/visualization.html
- *bokeh* is a Python interactive visualization library that targets modern web browsers for presentation. *bokeh* allows the creation of interactive plots, dashboards, and data applications (see Fig. 4.4) (<https://docs.bokeh.org/>).

¹ Depending on your build of Python, this command may also be `%matplotlib tk`.

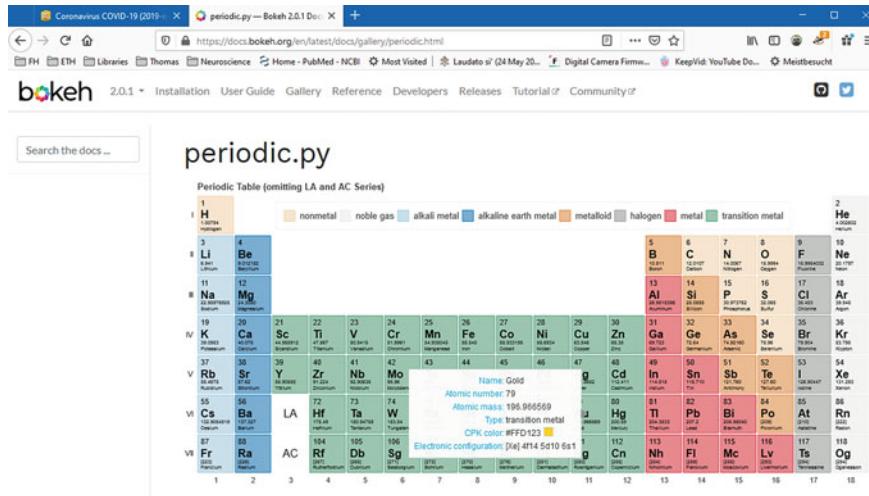


Fig. 4.4 Bokeh <https://bokeh.org> makes it possible to generate interactive graphics. Here, information about the element currently under the cursor is displayed

- *Dash* from <https://plotly.com> is used for the generation of web-based analytics apps.

4.2.1 Functional and Object-Oriented Approaches

Python plots can be generated in a functional, Matlab-like style, or in an object-oriented, more pythonic way. These styles are all perfectly valid, and each has its pros and cons. The only caveat is to avoid mixing the coding styles in your own code.

First, consider the frequently used functional “`pyplot`” style:

```
# Import the required packages, with their
# conventional names
import matplotlib.pyplot as plt
import numpy as np

# Generate the data
x = np.arange(0, 10, 0.2)
y = np.sin(x)

# Generate the plot
plt.plot(x, y)

# Display the plot on the screen
plt.show()
```

Note that the creation of the required figure and axis is done automatically by `pyplot`.

Second, a more pythonic, object-oriented style may be clearer when working with multiple figures and axes. Compared to the example above, only the section entitled “*# Generate the plot*” changes:

```
# Generate the plot
fig = plt.figure()          # Generate the figure
ax = fig.add_subplot(111)    # Add an axis to that figure
ax.plot(x,y)                # Add a plot to that axis
```

So, why all the extra typing as one moves away from the pure Matlab style? For very simple things like this example, the only advantage is academic: the wordier styles are more explicit, clearer as to where things come from and what is going on. For more complicated applications, this explicitness and clarity become increasingly valuable, and the richer and more complete object-oriented interface will likely make the program easier to write and maintain. For example, the following lines of code produce a figure with two plots above each other, and clearly indicate which plot goes into which axis:

```
# Import the required packages
import matplotlib.pyplot as plt
import numpy as np

# Generate the data
x = np.arange(0, 10, 0.2)
y = np.sin(x)
z = np.cos(x)

# Generate the figure and the axes simultaneously
fig, axs = plt.subplots(nrows=2, ncols=1)

# On the first axis, plot the sine and y-label
axs[0].plot(x,y)
axs[0].set_ylabel('Sine')

# On the second axis, plot the cosine
axs[1].plot(x,z)
axs[1].set_ylabel('Cosine')

# Display the resulting plot
plt.show()
```

 **Code:** `ISP_getting_started.py`² gives a short demonstration of Python for scientific data analysis.

² <[ISP2e> /04_DataDisplay/gettingStarted/ISP_gettingStarted.py](#).

4.2.2 Interactive Plots

Matplotlib provides different ways to interact with the user. The examples in `ISP_interactive_plots.py` may help to quickly create user interfaces for interactive visual data inspection. They show how to

- position figures on the screen,
- pause between two plots, and proceed automatically after a few seconds,
- proceed on a click or keyboard hit,
- evaluate keyboard entries, and
- interactively show information about individual data points, helping to find and evaluate outliers.



Code: `ISP_interactive_plots.py`³ shows the corresponding source code.

4.3 Saving a Figure

Matplotlib offers multiple options for the export of figures. In general, one distinguishes between pixel-based options (such as PNG or TIFF), vector-based options (such as SVG), or compressed formats (such as JPEG). Three of those formats are particularly useful:

JPEG (“Joint Photographic Experts Group”) is a compressed format which is recommendable for figures that are going to be printed or imported in other documents. When exporting to JPEG, note that it is recommendable to specify a relatively high-quality rate in order to avoid compression artifacts.

SVG (“Scalable Vector Graphic”) is useful when figures are to be modified in external applications. SVG is a vector graphic format, and allows you to modify different aspects of the data (LineStyle, LineWidth, etc.) in an external application as demonstrated in the next section.

PNG (“Portable Network Graphics”) is a compressed raster graphic format. It is commonly used on the WWW.

```
out_file = 'my_figure.jpg'

# specify a resolution of 200 dots-per-inch
plt.savefig(out_file, dpi=200)

# If you want/need a higher or lower quality, use
# pil_kwarg = {'quality': 99}
# plt.savefig(out_file, dpi=200, pil_kwarg=
```

³ <[ISP2e> /04_DataDisplay/interactivePlots/ISP_interactive_plots.py](#).

```
# pil_kwarg  
print(f'The figure has been saved to {out_file}')
```

Tip: One should always let the user know when the program generates a new file!

4.4 Preparing Figures for Presentation

4.4.1 General Considerations

All elements in Python, and therefore all elements of a *graphical user interface (GUI)*, are objects (see Fig. 4.3). As a result characteristics of figure elements can be determined or modified at three stages:

1. at the creation of the original figure elements,
2. after the creation by modifying the properties of the element. (For example, a line has a `linewidth`, a `color`, etc.), and
3. in an external graphics program.

I strongly recommend doing only the basic figure adjustments in Python. Then save the figure in SVG-format so that you can modify the individual elements and all the fine details, such as line-width and line-color, and text-size, in an external vector graphics program (see below). Properties such as text-size may change depending on how the figure is going to be used, and it is typically much easier to modify these properties in a graphics program instead of having to go back and also redo the computation of the figure. (This would require that the data analysis program is still working, that the input data are still available, that they are also in the correct location, ...)

Popular vector graphics programs

Adobe Illustrator I used to be an Adobe fan. But since Adobe has eliminated the option to buy software and only offers software subscriptions, I have turned my back on it.

Affinity Designer The new kid on the block, affordable, and getting very good reviews.

CorelDraw Commercial, cheaper than Illustrator, but more powerful than Inkscape (my personal favorite).

Inkscape An open-source, free vector drawing program.

While the specific steps may depend slightly on the program used, the overall procedure should be almost the same:

1. The SVG-file created in Python has to be imported into the vector graphics program.
2. All elements have to be un-grouped. (Note that there may be grouped elements within other groups!)
3. Now the individual elements can be modified.
4. For clarity, it is often useful to use *layers* for the different elements of the figure (lines, labels, and annotations).

4.4.2 Modifying SVG Figures

a) CorelDraw

In the following, the workflow for *CorelDraw* is described for modifying the figure obtained from `sine.svg`, a simple sine wave generated with *matplotlib*:

- File | Import → select `sine.svg` (Fig. 4.5, left).
- Specify the area where you want to insert the figure, either by clicking, or by click-and-drag.
- Object | Group | Ungroup All Objects.
- Select and remove the background. (This may have to be done twice, as there may be two background layers.) Note that the background is often white, and only becomes visible in the *wireframe* view of the illustration (Fig. 4.5, center).
- Select the line.
- Change line-color (by right-clicking the desired color) and fill color (left-clicking) of your elements. (“No color” can be selected by clicking the color \square .)
- Adjust the line-width and the line-style.

For adjusting text elements and labels, I typically proceed by adding additional “Layers”, and moving all the text (axis labels, line labels, etc.) into that layer. This way I can “lock” the figure and adjust the labeling without having to worry about the curves (Fig. 4.6).

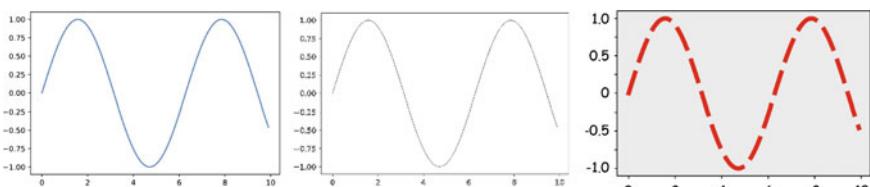


Fig. 4.5 **Left** Original SVG Figure. **Center** Wireframe view of the figure imported into CorelDraw. Note the background frame around the figure, which is not visible in the standard view!. **Right** Figure, with the line-style, background, and the tick-labeling modified, and exported to a JPEG-file

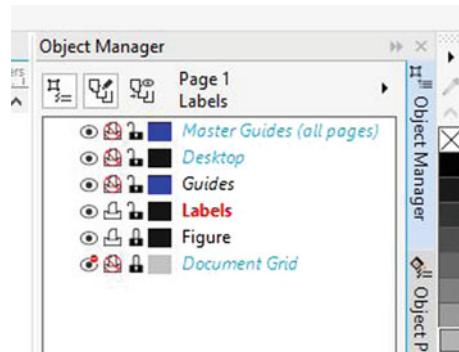


Fig. 4.6 Screenshot of the *Object Manager* in *CorelDraw*. The black arrow in the top-right corner lets you generate a new layer

- Open the *Object Manager* (*Windows | Dockers | Object Manager*).
- Click the black arrow/triangle in the upper-right corner, or the “New Layer” symbol in the lower-left corner of the *Object Manager*, and add a New Layer.
- Call the original layer “Figure”, and the new layer “Labels”.
- Select all the Text elements, and move/drag them from “Figure” to “Labels”.
- Lock “Figure”.
- Select “Labels”, and make all the desired text adjustments.

b) Affinity Designer

The workflow for *Affinity Designer* is similar:

- *File | Place* → select *sine.svg*.
- Specify the area where you want to insert the figure.
- Double-click the figure.
- *Layer | Ungroup All*.
- Select and remove the background. (This may have to be done twice, as there may be two background layers.) Note that the background is often white, and only becomes visible in the “wireframe” view of the illustration (Fig. 4.5, center).
- Select the line with the *Node* tool.
- Use the shortcut key X to toggle between the *Fill* color and the *Line* color. Change the fill to *empty* (by left-clicking the color), and the color to the color you want.
- Adjust the line-width and the line-style.

And for handling all the text in its own layer:

- In the *Layers* panel, select *Add Layer*.
- Call the original layer “Figure”, and the new layer “Labels”.
- Select all the Text elements with the *Move* tool, group them with *CTRL+G*, and label the group in the *Layers*-panel *Text*.
- Move/drag this group from “Figure” to “Labels”.

- Lock “Figure”.
- Select “Labels”, and make all the desired text adjustments.

c) Inkscape

Also in Inkscape, the procedure is similar to the other programs. It is free and open-source and gets the job done. But it is definitely less intuitive and less convenient to work with than commercial programs.

- File | Import → select sine.svg.
- Select the figure with the Select and Transport-tool, right-click and Ungroup it.
- Select the chosen line, and type Ctrl+Shift+F to adjust the Fill and Stroke-properties.
- Type Ctrl+Shift+L to get to the layers, and add a new Layer for the Text, and place it above Layer 1.
- To move the existing labels from Layer 1 to the Text-layer, Group all the labels together, select them, and type Shift+PageUp.

4.5 Display of Statistical Data Sets

Below, a short list of the most important *matplotlib* graphics commands for the display of statistical data is presented. Figure 4.17 shows how those different representations of data are related to each other.

4.5.1 Plots of Data with One Variable

The following examples all have the same format. Only the “Plot-command” line changes.

```
# Import standard packages
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import scipy.stats as stats
import seaborn as sns

# Generate the data
x = np.random.randn(500)

# Plot-command start -----
plt.plot(x, '.')
# Plot-command end -----

# Show plot
plt.show()
```

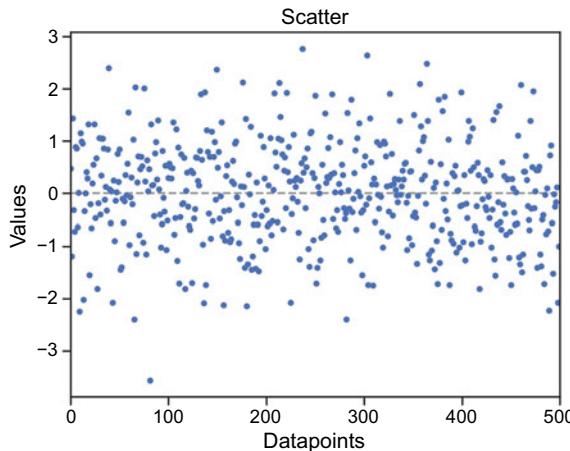


Fig. 4.7 Scatter plot (from the code-quantlet (“CQ”) show_plots.py)

a) Scatter Plots

This is the simplest way to represent “univariate” data, i.e., data with a single variable: just plot each individual data point (Fig. 4.7). The corresponding plot-command is either

```
plt.plot(x, '.')
```

or

```
plt.scatter(np.arange(len(x)), x)
```

Things to consider when generating scatter plots:

- If the data are not part of a sequence (e.g., data as a function of time), do *not* connect data points with a line.
- If there are few data points, it might be aesthetically more pleasing to use '`o`' or '`*`' instead of '`.`' as plot-symbol, leading to larger dot sizes.

Note: In cases where we only have a few discrete values on the x-axis (e.g., *Group1*, *Group2*, *Group3*), it may be helpful to spread overlapping data points slightly (also referred to as “*adding jitter*”) to show each data point. An example can be found at <http://stanford.edu/~mwaskom/software/seaborn/generated/seaborn.stripplot.html>.

b) Histograms

Histograms provide a good first overview of the distribution of data (Fig. 4.8). The box-width is arbitrary, and the smoothness of the histogram depends on the chosen box-width. The histogram can be represented as a *frequency histogram*, by simply counting the number of samples in each box. By using the option `density=True` of the command `plt.histogram` the histogram can be “normalized”, which corresponds to dividing the frequency counts by the total number of samples and the

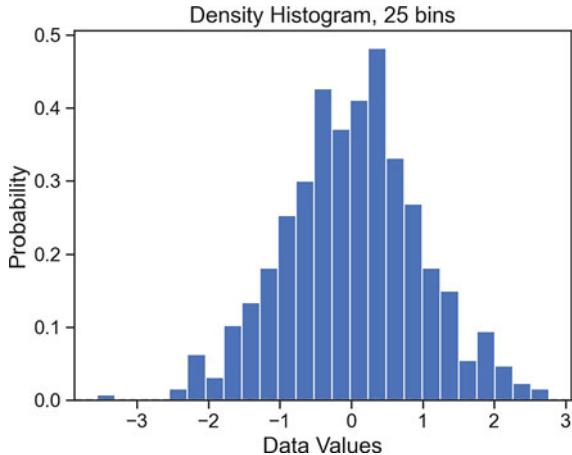


Fig. 4.8 A *density histogram* indicates the probability to find a single data sample in the corresponding bin with a bar (from CQ ISP_showPlots.py)

bin-width. In that mode, the area of each box corresponds to the probability of finding a data value in the corresponding data range, and the overall area of the histogram is exactly 1.

```
plt.hist(x, bins=25, density=True)
```

c) Kernel-Density-Estimation (KDE) Plots

Histograms have the disadvantage that they are discontinuous, and that their shape critically depends on the chosen bin-width. In order to obtain smooth *probability densities*, i.e., curves providing for each point a relative likelihood that the value of the random variable would be close to that point, the technique of *Kernel Density Estimation* (KDE) can be used. Thereby, a normal distribution is typically used for the kernel. The width of this kernel function determines the amount of smoothing. To see how this works, we compare the construction of histogram and kernel density estimators, using the following six data points:

```
x = [-2.1, -1.3, -0.4, 1.9, 5.1, 6.2].
```

For the histogram, first the horizontal axis is divided into sub-intervals or bins which cover the range of the data. In Fig. 4.9, left, we have 6 bins each of width 2. Whenever a data point falls inside this interval, we place a box of height $1/12 = 1/(6 * 2)$. If more than one data point falls inside the same bin, we stack the boxes on top of each other.

For the kernel density estimate, we place a normal kernel with variance 2.25 (indicated by the red dashed lines in Fig. 4.9, right) on each of the data points x_i . The kernels are summed to make the kernel density estimate (solid blue curve). The smoothness of the kernel density estimate is evident. Compared to the discreteness

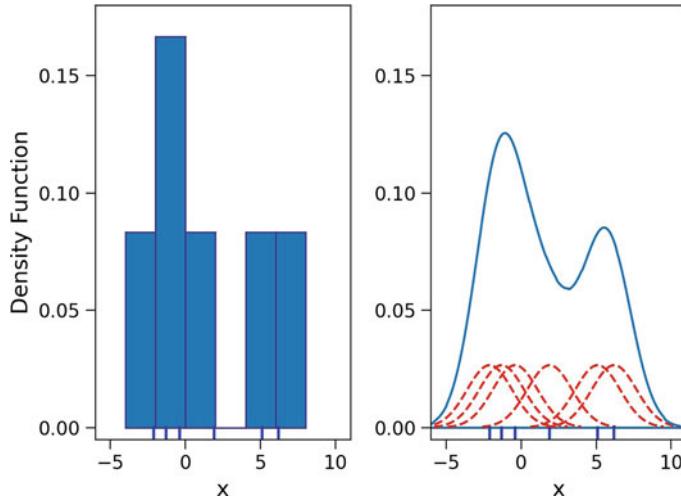


Fig. 4.9 Comparison of the histogram (left) and kernel density estimate (right) constructed using the same data. The six individual kernels are the red dashed curves, and the kernel density estimate the solid blue curve. The data points are the “rug plot” on the horizontal axis

of the histogram, the kernel density estimates converge faster to the true underlying density for continuous random variables.

```
sns.kdeplot(x)
```

The bandwidth of the kernel is the parameter which determines how much we smooth out the contribution from each event. To illustrate its effect, we take a simulated random sample from the standard normal distribution, plotted as the blue spikes in the *rug plot* on the horizontal axis in Fig. 4.10, left. (A *rug plot* is a plot where every data entry is visualized by a small vertical tick.) The right plot shows the true density in blue (A normal density with mean 0 and variance 1). In comparison, the gray curve is undersmoothed since it contains too many spurious data artifacts arising from using a bandwidth $h = 0.1$ which is too small. The green dashed curve is oversmoothed since using the bandwidth $h = 1$ obscures much of the underlying structure. The red curve with a bandwidth of $h = 0.42$ is considered to be optimally smoothed since its density estimate is close to the true density.

It can be shown that under certain conditions, the optimal choice for h is

$$h = \left(\frac{4\hat{\sigma}^5}{3n} \right)^{\frac{1}{5}} \approx 1.06\hat{\sigma} n^{-1/5}, \quad (4.1)$$

where $\hat{\sigma}$ is the standard deviation of the samples (“Silverman’s rule of thumb”).

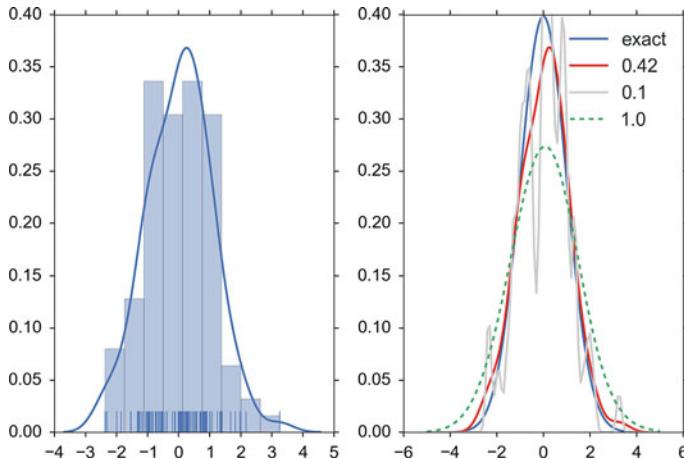


Fig. 4.10 **Left:** Rug plot, histogram, and kernel density estimate (KDE) of a random sample of 100 points from a standard normal distribution. **Right:** True density distribution (blue), and KDE with different bandwidths. Gray dashed: KDE with $h = 0.1$; red: KDE with $h = 0.42$; green dashed: KDE with $h = 1.0$

d) Cumulative Frequencies

A *cumulative frequency* gives the number of data points that lie below a given value. Dividing this number by the number of total points provides the percentage of data points below that value, and is commonly referred to as *cumulative distribution function (CDF)*, Fig. 4.11). This curve is very useful for statistical analysis, for example, when we want to know the data range containing 95% of all the values. Cumulative frequencies are also useful for comparing the distribution of values in two or more different groups of individuals.

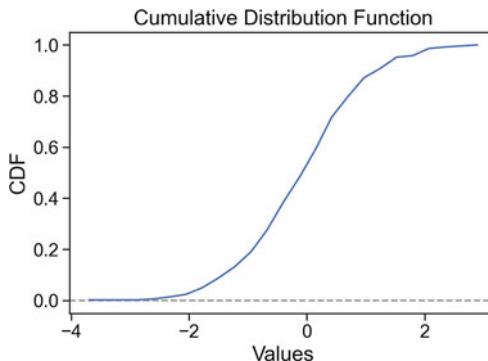


Fig. 4.11 Empirical cumulative distribution function for a normal distribution

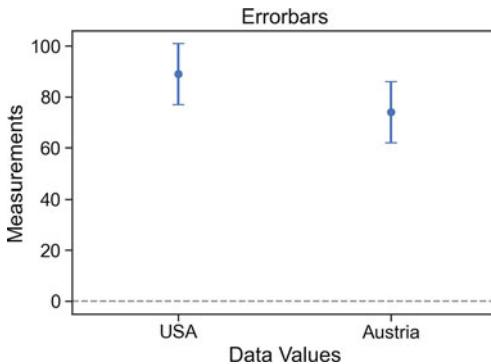


Fig. 4.12 Displaying mean and variability for different groups (from CQ ISP_showPlots.py)

The cumulative frequency is provided by the command `cumfreq` from the package `scipy.stats`. Since this command returns the *cumulative frequency*, the *lower limit*, and the *binsize*, the corresponding CDF can be plotted with

```
n_bins = 25      # number of bins for the figure

res = stats.cumfreq(x, numbins=n_bins)
lower_lim = res.lowerlimit
upper_lim = res.lowerlimit + n_bins*res.binsize
values = np.linspace(lower_lim, upper_lim, n_bins)
cdf = res.cumcount / len(x)

plt.plot(values, cdf)
plt.xlabel('Values')
plt.ylabel('CDF')
plt.title('Cumulative Distribution Function')
```

e) Error Bars

Error bars are a common way to show mean value and variability when comparing measurement values (Fig. 4.12). Note that it always has to be stated explicitly if the error bars correspond to the *standard deviation* or to the *standard error* of the data. *Standard errors* (see Sect. 6.1.2) make it easy to discern statistical differences between groups: when error bars for the standard errors for two groups overlap, one can be sure the difference between the two means is not statistically significant ($p > 0.05$). However, the opposite is not always true!

The following commands also show how to replace the tick labels on the x-axis with strings:

```
weight = {'USA':89, 'Austria':74}
sd_male = 12
plt.errorbar([1,2], weight.values(),
            yerr=sd_male * np.r_[1,1],
            capsizes=5, lw=0,
            elinewidth=2, marker='o')
```

```
plt.xlim([0.5, 2.5])
plt.xticks([1,2], weight.keys())
plt.ylabel('Weight [kg]')
plt.title('Adult male, mean +/- sd')
```

f) Boxplots

Boxplots are frequently used in scientific publications to indicate values in two or more groups (Fig. 4.13). The bottom and top of the box indicate the *lower quartile* (i.e., the value larger than 25% of the data) and *upper quartile* (i.e., the value larger than 75% of the data), respectively, and the height of the box is therefore also called the *interquartile range (IQR)*. The line inside the box shows the median (the value larger than 50% of the data). In other words, the box contains 50% of the data samples. The whiskers typically correspond to the most extreme values within $1.5 * \text{IQR}$ beyond the box. Samples with more extreme values are per definition outliers. Note that outliers often make important contributions to the statistical analysis, and should be checked individually for correctness; by default, they have to be included in the data analysis!

```
plt.boxplot(x, sym='*')
```

With *pandas* DataFrames and the *seaborn* command `sns.boxplot`, it is simple to use boxplots for the comparison of multiple groups (Fig. 4.14).

g) Grouped Bar Charts

For some applications, the plotting abilities of *pandas* can facilitate the generation of useful graphs, e.g., for grouped bar plots (Fig. 4.15). Here, I use colormaps from

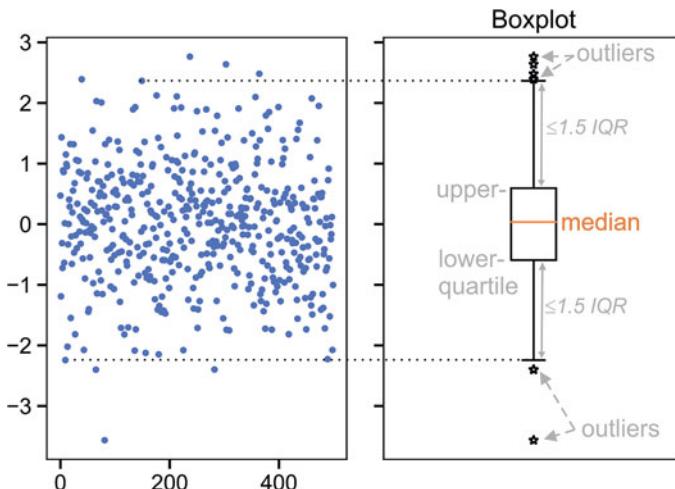


Fig. 4.13 Boxplot. Data that lie beyond the error bars indicate “outliers”. (From CQ ISP_showPlots.py; additional labels were added manually.)

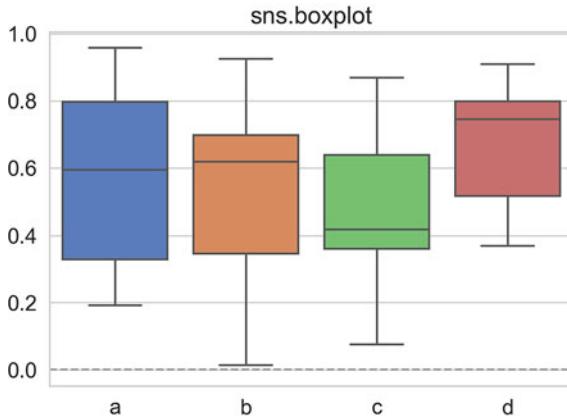


Fig. 4.14 Grouped boxplot, for a DataFrame with the columns ['a', 'b', 'c', 'd'] (from CQ ISP_showPlots.py)

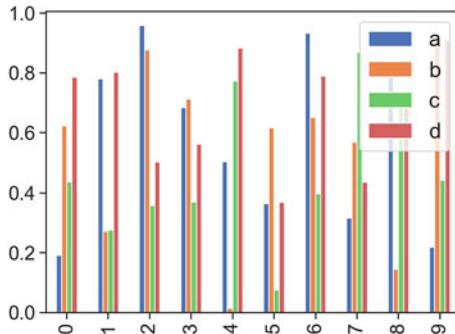


Fig. 4.15 Grouped barplot, produced with *pandas* (from CQ ISP_showPlots.py)

the statistics visualization package *seaborn*, which is commonly imported as `sns`. Among many statistical visualization features, *seaborn* also offers a number of practical color maps (https://seaborn.pydata.org/tutorial/color_palettes.html).

```
df = pd.DataFrame(np.random.rand(10, 4),
                  columns=['a', 'b', 'c', 'd'])
df.plot(kind='bar', grid=False,
        color=sns.color_palette('muted')) # easy on the eye
```

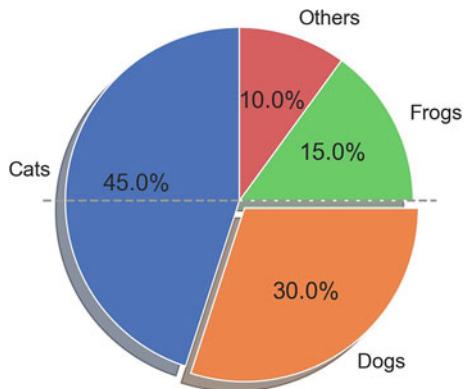
h) Pie Charts

Pie charts can be generated with a number of different options (Fig. 4.16).

```
import seaborn as sns
import matplotlib.pyplot as plt

txtLabels = 'Cats', 'Dogs', 'Frogs', 'Others'
```

Fig. 4.16 “Sometimes, it is raining cats and dogs” (from CQ ISP_showPlots.py)



```

fractions = [45, 30, 15, 10]
offsets =(0, 0.05, 0, 0)

plt.pie(fractions, explode=offsets, labels=txtLabels, autopct
        ='%1.1f%%', shadow=True, startangle=90, colors=sns.
        color_palette('muted'))
plt.axis('equal')

```

i) Programs: Data Display



Code: ISP_show_plots.py⁴ contains the Python code to generate the plots in this section.

j) Summary of Important Univariate Plots

Figure 4.17 summarizes the relationships between some important representations of univariate data. Details explaining the equations on the right-hand side will be described in the next chapters.

4.5.2 Plots of Data with Two or More Variables

There is always a trade-off between simplicity and information density. Care should be taken to keep plots that put more information into a single graph understandable to the reader.

a) Scatter Plots

Scatter plots of 2D data (“bivariate data”) can add additional information through the symbol size (Fig. 4.18). This type of plot is still very straightforward to understand.

⁴ <ISP2e> /code_quantlets/04_DataDisplay/ISP_showPlots.py.

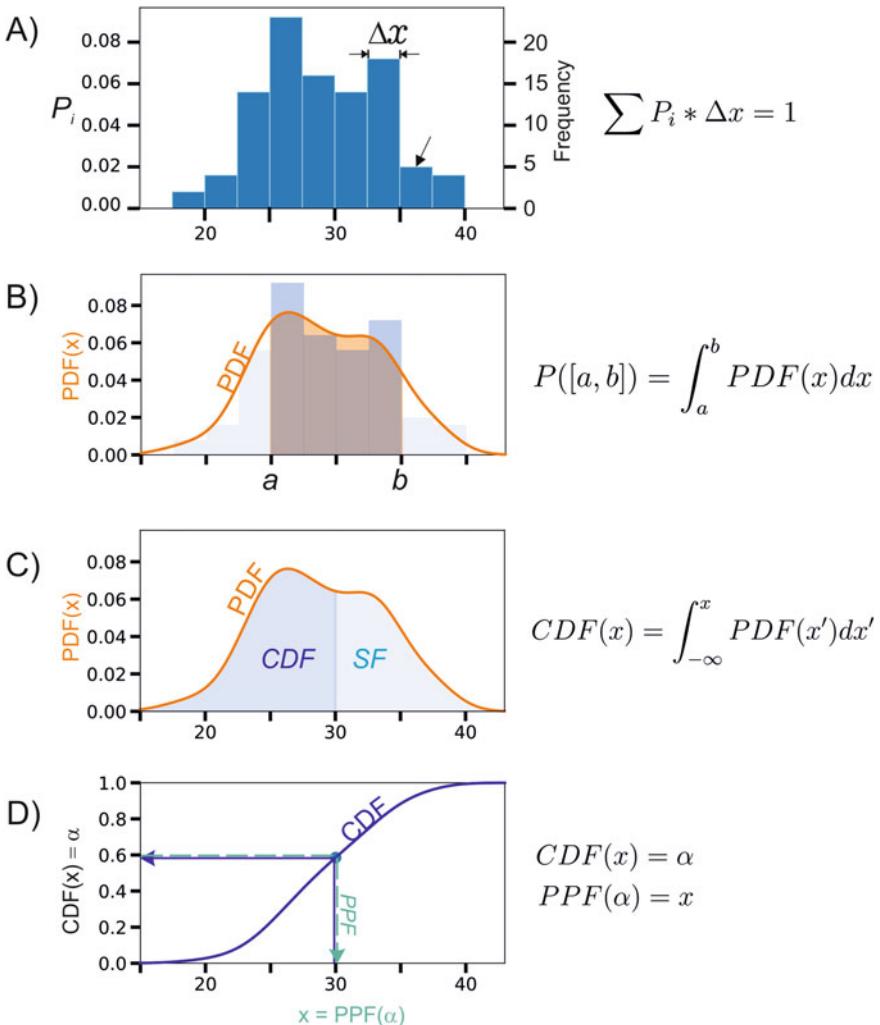
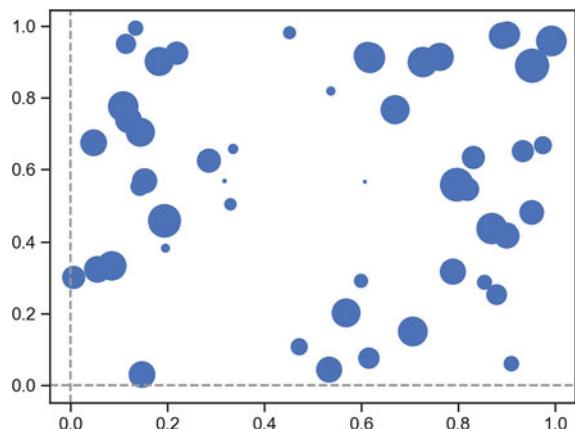


Fig. 4.17 Summary of relationships between important statistical plots. **A** For a *probability histogram* (left y-axis), the area of each bar corresponds to the probability of finding a sample in the corresponding value range. Here, the bars have a width of $\Delta x = 2.5$, so the probability of finding a value between 35 and 37.5 (second bar from the right, black arrow, $P = 0.02$) is $0.02 * 2.5 = 0.05 \equiv 5\%$. Here, $N=100$, so 5 subjects were found in that range (*Frequency-histogram*, right y-axis). **B** A *Kernel density estimation (KDE)* provides a smoothed version of the probability histogram. The corresponding curve is a *Probability Density Function (PDF)*. The probability to find a value between a and b is given by the area under the PDF in that range, indicated here in orange. Since any measurement must have *some* value, a PDF must have the property $\int_{-\infty}^{\infty} PDF(x)dx = 1$. **C** The *Cumulative Distribution Function (CDF)* of x gives the probability of obtaining a sample with a value less than $x = 30$, and corresponds to the area under the PDF up to that value. The *Survival Function (SF)* is simply the complementary probability: $SF(x) = 1 - CDF(x)$. **D** The *Percentile Point Function (PPF)* is the inverse of the CDF ($PPF = CDF^{-1}$). In the example here, the turquoise arrow indicates that in order to obtain 60% of the area under the PDF, all samples with a value up to 30 have to be included

Fig. 4.18 Scatter plot with scaled data points (from CQ ISP_showPlots.py)



```
""" Demonstration of 3D plots """

# imports specific to the plots in this example
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d.axes3d import get_test_data

# Twice as wide as it is tall.
fig = plt.figure(figsize=plt.figaspect(0.5))

#---- First subplot
# Note that the declaration "projection='3d'" is required for 3d plots!
ax = fig.add_subplot(1, 2, 1, projection='3d')
```

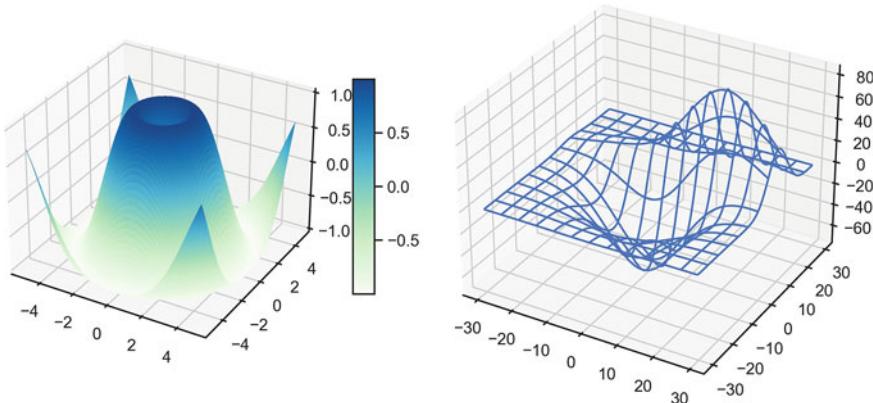


Fig. 4.19 Two types of 3D graphs. **Left** surface plot. **Right** wireframe plot (from CQ ISP_showPlots.py)

```

# Generate the grid
X = np.arange(-5, 5, 0.1)
Y = np.arange(-5, 5, 0.1)
X, Y = np.meshgrid(X, Y)

# Generate the surface data
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

# Plot the surface
surf = ax.plot_surface(X, Y, Z, rstride=1,
cstride=1, cmap=cm.GnBu, linewidth=0,
antialiased=False)
ax.set_zlim3d(-1.01, 1.01)

fig.colorbar(surf, shrink=0.5, aspect=10)

#---- Second subplot
ax = fig.add_subplot(1, 2, 2, projection='3d')
X, Y, Z = get_test_data(0.05)
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)

outfile = '3dGraph.jpg'
plt.savefig(outfile, dpi=200)
print(f'Image saved to {outfile}')
plt.show()

```

4.6 Exercises

1. Plotting Data

From the command line, create two cycles of a noisy sine wave with the following properties:

`amplitude = 1, frequency = 0.3 Hz, sample_rate = 100 Hz.`
Add Gaussian random noise with a standard deviation of 0.5 to these data.

- Plot the data, label the x- and the y-axis, and add a title to the plot.
- When this works, take your command history, clean it up, and create a Python function that
 - takes the number of cycles and the frequency as input.
 - sets the remaining parameters as above.
 - shows the resulting graph.
- Set a breakpoint in the function somewhere after the amplitude has been defined, and use a debugger to inspect the workspace variables at that point. Modify the amplitude to 2 and continue the function.

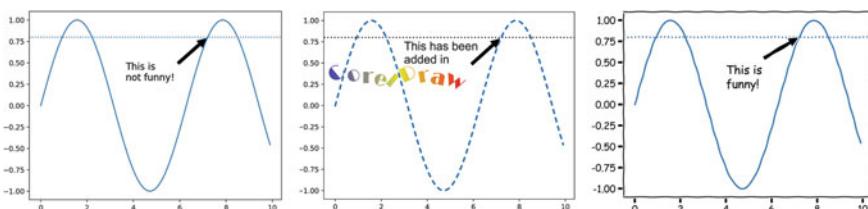
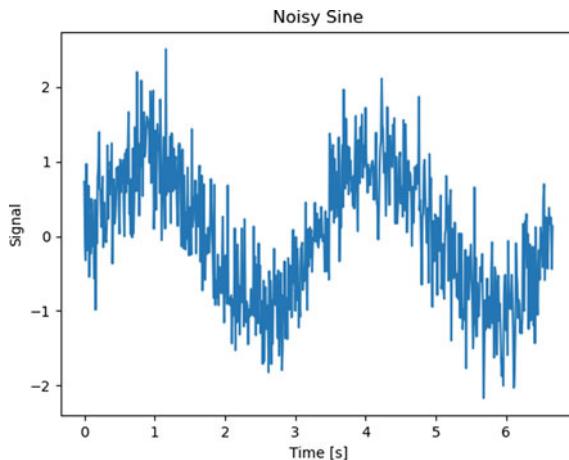


Fig. 4.20 **Left** Original `matplotlib` figure, with annotation. **Center** SVG figure, modified in a vector graphic program. **Right** Same as left figure, but with `plt.xkcd()`

2. Modifying Figures

Start out with:

- plotting a sine wave, and
- adding a horizontal line at $y=0.8$.
- Then find the *matplotlib* command to annotate a point on the plot, and annotate the intersection of the horizontal line with the sinusoid (Fig. 4.20, left).

This gives you the original *matplotlib* figure (see left panel).

Then:

- Save the figure in SVG-format. Use a vector graphics program, modify the line attributes of the plot, and modify the text of the annotation (Fig. 4.20, center).
- Make the figure look like a hand-drawn sketch, as indicated in Fig. 4.20, right panel. Check the *matplotlib* documentation for the command `plt.xkcd()`, which provides that functionality.

Part II

Distributions and Hypothesis Tests

This part of the book moves the focus from Python to statistics.

The first chapter in this part serves to define the statistical basics, like the concepts of *populations* and *samples*, data types, and *probability distributions*. It also includes a short overview of *study design*. The design of statistical studies is seriously underestimated by most beginning researchers: faulty study design produces garbage data, and the best analysis cannot remedy those problems (“*Garbage in–garbage out*”). However, if the study design is good, but the analysis faulty, the situation can be fixed with a new analysis, which typically takes much less time than an entirely new study.

The next chapter shows how to characterize the position and the variability of a distribution, and then uses the normal distribution to describe the most important Python methods common to all distribution functions. After that, the most important discrete and continuous distributions are presented.

The third chapter in this part first describes a typical workflow in the analysis of statistical data. Then the concept of *hypothesis tests* is explained, as well as the different types of errors, and common concepts like *sensitivity* and *specificity*.

The remaining chapters explain the most important hypothesis tests, for continuous variables and for categorical variables. A separate chapter is dedicated to survival analysis (which also encompasses the statistical characterization of material failures and machine breakdowns), as this question requires a somewhat different approach than the other hypothesis tests presented here. Each of these chapters also includes working Python sample code (including the required data) for each of the tests presented. This should make it easy to implement the tests for different data sets.

Chapter 5

Basic Statistical Concepts



This chapter briefly introduces the main concepts underlying the statistical analysis of data. It defines discrete and continuous probability distributions, and then gives an overview of various types of study designs.

5.1 Populations and Samples

In the statistical analysis of data, we typically use data from a few selected *samples* to draw conclusions about the *population* from which these samples were taken. Correct *study design* should ensure that the sample data are representative of the population from which the samples were taken.

Since the parameters characterizing the population are unknown, we have to use samples to obtain estimates of those parameters (see Fig. 5.1).

Population includes all of the elements from a set of data.

Sample consists of one or more observations from the population.

More than one sample can be taken from the same population.

When estimating a *parameter* of a *population*, e.g., the expected value of the weight of male Europeans, we typically cannot measure all subjects. We have to limit ourselves to investigating a (hopefully representative) *sample* taken from this group. Based on the *sample statistic*, i.e., the corresponding value calculated from the sample data, we use *statistical inference* to draw conclusions about the corresponding parameter in the population.

Parameter Characteristic of a distribution describing a population, such as the mean or standard deviation of a normal distribution. Often notated using Greek letters.

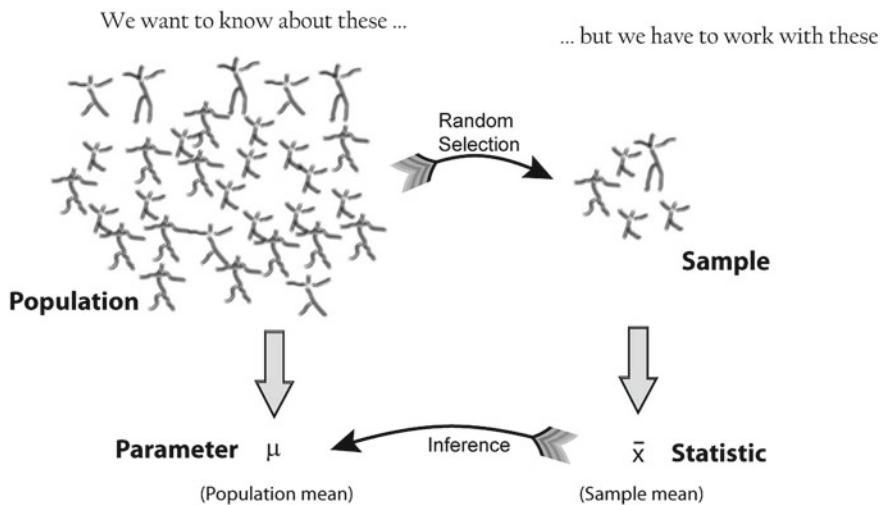


Fig. 5.1 With statistical inference, information from samples is used to estimate parameters from populations

Statistic A numerical value that represents a property of a random sample. Examples of statistics are

- the mean value of the sample data.
- the range of the sample data.
- deviation of the data from the sample mean.

(Empirical) sampling distribution The probability distribution of a given statistic based on a random sample.

Statistical inference Enables you to make an educated guess about a population parameter based on a statistic computed from a representative sample from that population.

Examples of parameters and statistics are given in Table 5.1. Population parameters are often indicated using Greek letters, while sample statistics typically use standard letters.

Table 5.1 Comparison of sample statistics and population parameters

	Sample statistic	Population parameter
Mean	\bar{x}	μ
Standard deviation	sd	σ

5.2 Data Types

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.

5.2.1 Categorical

a) Boolean

Boolean data are data which can only have two possible values. For example,

- 0/1
- yes/no
- smoker/non-smoker
- 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*, sometimes also called *ranked data*, are ordered and have a logical sequence, e.g., *very few/few/some/many/very many*.

5.2.2 Numerical

a) Numerical Continuous

Whenever possible, it is best to record the data in their continuous format. Of course it is feasible to keep only a limited number of decimal places. For example, it does not make sense to record the body size with more than 1 mm 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 ...*

5.2.3 Data with One, Two, or More Variables

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 versus age. *Multivariate* data have three or more variables, e.g., the position of a particle in space.

5.3 Probability Distributions

5.3.1 Definitions

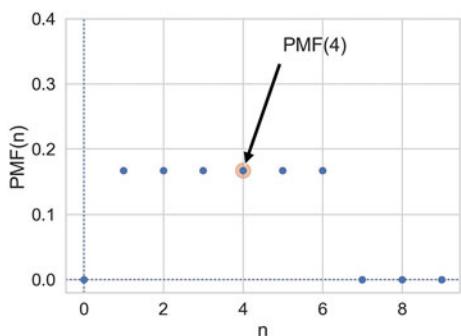
a) Random Variables and Variates

A *random variable* X is a quantity which depends on random events. For example, it can be the value of a rolled dice, which can take on any number from 1 to 6. A *random variate* is a particular outcome or “realization” of a random variable. For example, when you roll a dice three times and get the values [1, 3, 4], then those values would be three random variates.

b) Probability Distribution

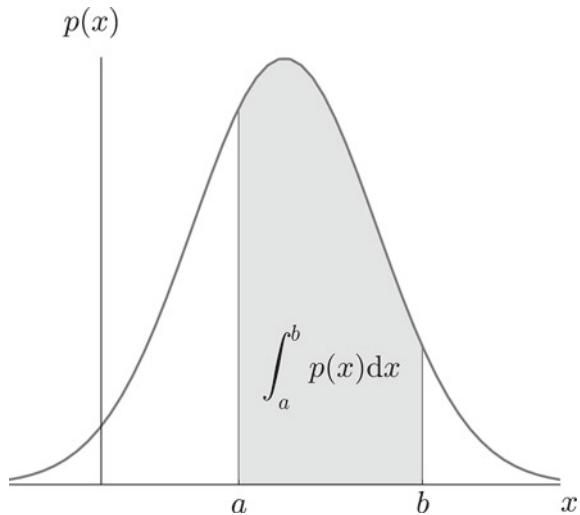
The mathematical tools to describe the randomness of data in populations and samples are *probability distributions*. For discrete distributions one uses the so-called *Probability Mass Functions (PMF)* (see Fig. 5.2); and for continuous distributions the *Probability Density Functions (PDF)* (see Fig. 5.3)¹.

Fig. 5.2 When throwing a dice, each of the numbers from 1 to 6 has a $1/6$ probability of coming up. For all other numbers, the probability is zero. These values define the *Probability mass function (PMF)* for rolling dice. The labeled point indicates that the probability to throw a 4 is $1/6$, i.e., about 17%



¹ In *Mathematica* also the distribution functions for discrete distributions are called PDF.

Fig. 5.3 Let $p(x)$ be the Probability Density Function (PDF) of a random variable X . The integral over $p(x)$ between a and b represents the probability of finding the value of X in that range



5.3.2 Discrete Distributions

A simple example of a discrete probability distribution is the game of throwing dice: for each of the numbers $i = 1, \dots, 6$, the probability that at the throw of a die the side showing the number i faces upward, P_i , is (Fig. 5.2)

$$P_i = \frac{1}{6}, \quad i = 1 \dots 6. \quad (5.1)$$

The set of all these probabilities $\{P_i\}$ makes up the *probability distribution* for rolling dice.

Note that the smallest possible value for P_i is 0. And since one of the faces of the die has to turn up at every throw of the dice, we have

$$\sum_{i=1}^6 P_i = 1. \quad (5.2)$$

Generalizing this, we can say that a discrete probability distribution has $\{P_i\}$ with the following properties:

- $0 \leq P_i \leq 1 \forall i \in \mathbb{Z}$ (i.e. for all integer numbers).
- $\sum_i P_i = 1$.

For a given discrete distribution, the P_i are called the *Probability Mass Function (PMF)* of that distribution.

5.3.3 Continuous Distributions

Many measurements have an outcome that is not restricted to discrete integer values. For example, the weight of a person can be any positive number. In this case, the curve describing the *probability distribution*, i.e. the probability for each value, is a (piecewise) continuous function $p(x)$, the *Probability Density Function (PDF)*.

The PDF, or *density* of a continuous random variable, is a function that describes the relative probability of a random variable X to take on a value close to x .

The PDF $p(x)$ has the following properties (Fig. 5.3)²:

- $p(x) \geq 0 \forall x \in \mathbb{R}$.
- The probability that a value between a and b occurs is
$$p(a < x < b) = \int_a^b p(x) dx.$$
- $\int_{-\infty}^{\infty} p(x) dx = 1$.

5.3.4 Expected Value and Variance

5.3.4.1 Expected Value

The PDF allows us to calculate the *expected value* $E[X]$ or *first moment* of a continuous distribution of X :

$$E[X] = \int_{-\infty}^{\infty} x \cdot p(x) dx. \quad (5.3)$$

The expected value is typically indicated with μ . Since parameters characterizing a random variable are sometimes pre-fixed with the attribute “population”, this value is also referred to as the *population mean*.

For discrete distributions, the integral over x is replaced by the sum over all possible values, multiplied with their respective probabilities:

$$E[X] = \sum_i x_i P_i, \quad (5.4)$$

where x_i represents all possible values that the random variable can have. The expected value only depends on the probability distribution.

Figure 5.1 shows a sketch how a sample statistic relates to the corresponding population parameter. The *sample mean* of our sample, \bar{x} , is the observed mean value of the sample. If the experiment has been designed correctly, the expected uncertainty in the sample mean should decrease as more and more data are included in the sample.

² In this book only one-dimensional probability distributions are considered.

5.3.4.2 Variance

The variability of a random variable X is characterized by its *variance*, sometimes also referred to as the *second standardized moment*:

$$\begin{aligned} \text{Var}(X) &= E[(X - E[X])^2] \\ &= E[X^2] - (E[X])^2 \end{aligned} \tag{5.5}$$

This *population variance* also depends only on the distribution of X , and is commonly denoted with σ^2 , where σ is the corresponding *population standard deviation*.

When we calculate the *sample variance*, we typically do not know the true value of $E[X] = \mu$. The best thing we can do is to replace μ with the average value of our data set, \bar{x} , thereby minimizing the mean square error. One can show that the expected sample variance \tilde{s}^2 , also called *biased sample variance*, is slightly but systematically lower than the population variance:

$$E[\tilde{s}^2] = \frac{n-1}{n} \sigma^2, \tag{5.6}$$

where n is the sample size. Taking the factor of $\frac{n-1}{n}$ into consideration, the best estimate of the population variance is the (*unbiased*) *sample variance*, denoted s^2 :

$$s^2 = \frac{n}{n-1} \tilde{s}^2 = \frac{\sum_i (x_i - \bar{x})^2}{n-1} \tag{5.7}$$

This (unbiased) sample variance, often short called *sample variance*, is what you commonly use when working with data.

5.3.4.3 Standard Deviation

The *standard deviation* is the square root of the variance, and has the advantage that it has the same dimension as X . The *sample standard deviation* is the square root of the sample variance:

$$s = \sqrt{\frac{\sum_i (x_i - \bar{x})^2}{n-1}} \tag{5.8}$$

As indicated in Table 5.1, in statistics it is common to denote the population standard deviation with σ , and the sample standard deviation with s . To improve clarity, in code segments in this book the sample standard deviation will be indicated with `sd` or `std`.

5.4 Degrees of Freedom

The concept of Degrees of Freedom (DOF), which in mechanics appears to be crystal clear, may be harder to grasp for statistical applications.

In mechanics, a particle which moves in a plane has “2 DOF”: at each point in time, two parameters (the x/y-coordinates) define the location of the particle. If the particle moves about in space, it has “3 DOF”: the x/y/z-coordinates.

In statistics, a group of n values has n DOF. If we subtract the sample mean from each value, then, the remaining data only have $n-1$ DOF. (This is clearest for $n = 2$: if we know the mean value and the value of sample_1 , then we can calculate the value of sample_2 by $\text{val}_2 = 2 * \text{mean} - \text{val}_1$.)

The case becomes more complex when we have many groups. For example, in Sect. 8.3.1, there is an example with 22 patients, divided into 3 groups. In the *analysis of variance (ANOVA)*, the DOFs in this example are divided as follows:

- 1 DOF for the total mean value.
- 2 DOF for the mean value of each of the three groups. Remember: if we know the mean values of 2 groups *and* the total mean, we can calculate the mean value of the third group.
- 19 DOF ($= 22 - 1 - 2$) are left for the residual deviations from the group means.

5.5 Study Design

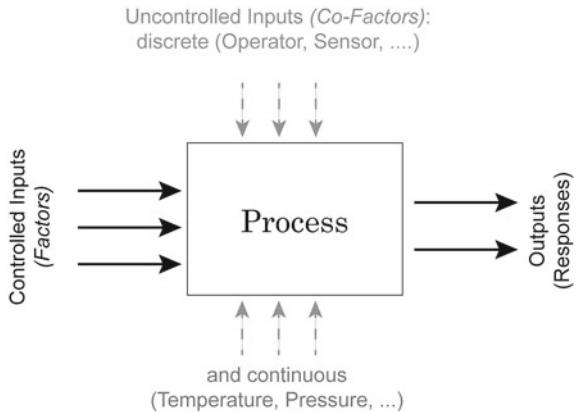
The importance of a good study design has been demonstrated recently by an investigation showing the effect of the introduction of the *clinicaltrials.gov* registry (Kaplan and Irvin 2015): A 1997 US law mandated the creation of the registry, requiring researchers from 2000 onwards to record their trial methods and outcome measures *before* collecting data. Kaplan et al. looked at studies evaluating drugs or dietary supplements for the treatment or prevention of cardiovascular disease. They found that before the introduction of *clinicaltrials.gov*, 57% of the studies showed a positive outcome, while after the introduction, this number was reduced dramatically to only 8%. In other words, without rigorous study design, there is a significant bias toward getting the result that you hope for.

5.5.1 Terminology

In the context of study design, various terminology can be found (Fig. 5.4):

- The controlled inputs are often called *factors* or *treatments*.
- The uncontrolled inputs are called *cofactors*, *nuisance factors*, or *confoundings*.

The term *covariate* refers to a variable that is possibly predictive of the outcome being studied, and can be a factor or a cofactor.

Fig. 5.4 Process schematic

When we try to model a process with two inputs and one output, we can formulate a mathematical model, for example, as

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_{12} X_1 X_2 + \epsilon. \quad (5.9)$$

The terms with the single X (β_1, β_2) are called *main effects*, and the terms with multiple X (β_{12}) *interaction terms*. And since the β parameters enter the equation only linearly, this is referred to as a *general linear model*. The ϵ are called *residuals*, and are expected to be distributed approximately normally around zero if the model describes the data correctly.

The *primary outcome measure* is the outcome that an investigator considers to be the most important one. The primary outcome measure should be defined *before* the beginning of the data collection, to prevent the investigator from cherry-picking significant results and presenting these as the main findings of the study. The primary outcome measure forms the basis for the calculation of the required sample size (Andrade 2015).

5.5.2 Overview

The first step in the design of a study is the explicit clarification of the goal of the study. Do we want to

1. Compare two or more groups, or one group to a fixed value?
2. Screen the observed responses to identify factors/effects that are important?
3. Maximize or minimize a response (variability, distance to target, robustness)?
4. Develop a regression model to quantify the dependence of a response variable on the process input?

The first question leads to a *hypothesis test*. The second one is a *screening investigation*, where one must be watchful of artifacts if the factors in the model are not

completely independent. The third task is an *optimization problem*. And the last one brings us into the realm of *statistical modeling*.

Once we have determined *what* we want to do, we have to decide *how* we want to do this. Either *controlled experiments* or *observations* can be used to obtain the necessary data. In a controlled experiment, we typically try to vary only a single parameter, and investigate the effects of that parameter on the output.

5.5.3 Types of Studies

a) Observational or Experimental

In an *observational study*, the researcher only collects information, but does not interact with the study population. In contrast, in an *experimental study*, the researcher deliberately influences events (e.g., treats the patient with a new type of medication) and investigates the effects of these interventions.

Observational studies are sometimes also referred to as *uncontrolled studies*, and *experimental studies* as *controlled studies*.

b) Prospective or Retrospective

In a *prospective study*, the data are collected from the beginning of the study. In contrast, a *retrospective study* takes data acquired from previous events, e.g., routine tests done at a hospital.

c) Longitudinal or Cross-Sectional

In *longitudinal* investigations, the researcher collects information over a period of time, maybe multiple times from each patient. In contrast, in *cross-sectional* studies individuals are observed only once. For example, most surveys are cross-sectional, but experiments are usually longitudinal.

d) Case Control and Cohort Studies

In a *case control study*, first the patients are treated, and then they are selected for inclusion in the study, based on certain criteria (e.g., whether they responded to a certain medication). In contrast, in a *cohort study*, subjects of interest are selected first, and then these subjects are studied over time, e.g., for their response to a treatment.

e) Randomized Controlled Trial

The golden standard for experimental scientific clinical trials, and the basis for the approval of new medications, is the *randomized controlled trial*. Here, bias is avoided by splitting the subjects to be tested into an *intervention group* and a *control group*. Group allocation is *random*.

In a designed experiment, there may be several conditions, called *factors*, that are controlled by the experimenter. By having the groups differ in only one aspect, the factor *treatment*, one should be able to detect the effect of the treatment on the patients.

Through randomization, confoundings should be balanced across the groups.

f) Crossover Studies

An alternative to randomization is the *crossover* design of studies. A crossover study is a longitudinal study in which subjects receive a sequence of different treatments. Every subject receives every treatment. (The subject “crosses over” from one treatment to the next.) To avoid causal effects, the sequence of the treatment allocation should be randomized.

For example, in an investigation that tests the effect of standing and sitting on the concentration of subjects, each subject performs both the execution of tasks while standing and the execution of tasks while sitting. The sequence of standing/sitting is randomized, to cancel out any sequence effects.

5.5.4 *Design of Experiments*

Block whatever you can; and randomize the rest!

I have mentioned above that we have *factors* (which we can control) and *nuisance factors*, which influence the results, but which we cannot control and/or manipulate. Assume, for example, that we have an experiment where the results depend on the person who performs the experiment (e.g., the nurse who tests the subject), and on the time of the day. In that case, we can block the factor *nurse*, by having all tests performed by the same nurse. But it won't be possible to test all subjects at the same time. So, we try to average out time effects, by *randomly* mixing the timing of the subjects. If, in contrast, we measure our patients in the morning and our healthy subjects in the afternoon, we will invariably bring some *bias* into our data.

a) Sample Selection

When selecting the subjects, one should pay attention to the following:

1. The samples should be representative of the group to be studied.
2. In comparative studies, groups must be similar with respect to known sources of variation (e.g., age, ...).
3. **Important:** Make sure that your selection of samples (or subjects) sufficiently covers all of the parameters that you need! For example, if age is a nuisance factor, make sure you have enough young, middle-aged, and elderly subjects.

Ad 1: For example, randomly selected subjects from patients at a hospital automatically bias the sample toward subjects with health problems.

Ad 3: For example, tests of the efficacy of a new rehabilitation therapy for stroke patients should *not* only include patients who have had a stroke: make sure that there are equal numbers of patients with mild, medium, and severe symptoms. Otherwise, one may end up with data which primarily include patients with few or no after-effects of the stroke. (This is one of the easiest mistakes to make, and cost me many months of work!)

Many surveys and studies fall short of these criteria (see the section on *Bias* below). The field of “matching by propensity scores” attempts to correct these problems (Rosenbaum and Rubin 1983).

b) Sample Size

Many studies also fail because the sample size is too small to observe an effect of the desired magnitude. In determining the sample size, one has to know

- What is the variance of the parameter under investigation?
- What is the magnitude of the expected effect, relative to the standard deviation of the parameter?

This is known as *power analysis*. It is especially important in behavioral research, where research plans are not approved without careful sample size calculations. (See also Sect. 7.2.5.)

c) Bias

To explain the effects of selection bias on a statistical analysis, consider the 1936 presidential elections in the USA. The Republican A. Landon challenged the incumbent president, F. D. Roosevelt. *Literary Digest*, at the time one of the most respected magazines, asked 10 million Americans who they would vote for. 2.4 million responded, and *Literary Digest* predicted Landon would win 57% of the vote compared with 41% for Roosevelt. However, the actual election results were 62% for Roosevelt and 38% for Landon. In other words, despite the huge sample size, the predictions were a whopping 19% off!

What went wrong?

First, the sample was poorly chosen, and not representative of the American voter: the mailing lists for the survey were taken from telephone directories, club membership lists, and lists of magazine subscribers. Thus, they were strongly biased toward the American middle- and upper classes. And second, only about one-fourth of the people asked responded. And people who respond to surveys are different from people who don't, the so-called *non-response bias*. This example shows that a large sample size alone does not guarantee a representative response. One has to watch out for selection bias and non-response bias.

In general, when selecting the subjects, one tries to make them representative of the group to be studied; and one tries to conduct the experiments in a way representative of investigations by other researchers. However, it is very easy to get biased data.

Bias can have a number of sources:

- The selection of subjects.
- The structure of the experiment.
- The measurement device.
- The analysis of the data.

Care should be taken to avoid bias in the data as much as possible.

d) Randomization

This may be one of the most important aspects of experimental planning. Randomization is used to avoid bias as much as possible, and there are different ways to randomize an experiment. For randomization, *random number generators*, which are available in most computer languages, can be used. To minimize the chance of bias, the randomly allocated numbers should be presented to the experimenter as late as possible.

Depending on the experiment, there are various ways to randomize the group assignment:

Simple Randomization

This procedure is robust against selection and accidental bias. The disadvantage is that the resulting group sizes can differ significantly.

For many types of data analysis, it is important to have the same sample number in each group. To achieve this, other options are possible:

Block Randomization

This is used to keep the number of subjects in the different groups closely balanced at all times. For example, with two types of treatment, A and B, and a block-size of four, one can allocate the two treatments to the blocks of four subjects in the following sequences:

1. AABB
2. ABAB
3. ABBA
4. BBAA
5. BABA
6. BAAB

Based on this, one can use a random number generator to generate random integers between 1 and 6, and use the corresponding blocks to allocate the respective treatments. This will keep the number of subjects in each group always almost equal.

Minimization

A closely related, but not completely random, way to allocate a treatment is *minimization*. Here, one takes whichever treatment has the smallest number of subjects, and allocates this treatment with a probability greater than 0.5 to the next patient.

Assume, for example, that you are conducting a randomized controlled trial of a new medication, with a “placebo-group” and a “real medication group”. Halfway through the trials, you realize that your placebo-group already contains 60 subjects, while your medication-group only has 40. You can now solve this imbalance, by giving each remaining subject with 60% probability (instead of the previously used 50%) the medication instead of the placebo.

Stratified Randomization

Sometimes, one may want to include a wider variety of subjects, with different characteristics. For example, one may choose to have younger as well as older subjects.

In this case, one should try to keep the number of subjects within each *stratum* balanced. In order to do this, separate lists of random numbers should be kept for each group of subjects.

e) Blinding

Consciously or not, the experimenter can significantly influence the outcome of an experiment. For example, a young researcher with a new “brilliant” idea for a new treatment will be biased in the execution of the experiment, as well in the analysis of the data, to see the hypothesis confirmed. To avoid such subjective influence, ideally the experimenter as well as the subject should be blinded to the therapy. This is referred to as *double blinding*. When also the person who does the analysis does not know which group the subject has been allocated to, we speak about *triple blinding*.

f) Factorial Design

When each combination of factors is tested, we speak of *full factorial design* of the experiment.

In planning the analysis, one must distinguish between *within subject comparisons*, and *between subjects* comparisons. The former, *within subject comparisons*, allows detecting smaller differences with the same number of subjects than *between subject comparisons*.

5.5.5 Recommendations for Researchers

The very recommendable article by Button et al. on low-powered studies (i.e., studies with a low likelihood of finding an actually existing effect) lists the following recommendations for scientific studies (Button et al. 2013).

Perform an a Priori Power Calculation

Use the existing literature to estimate the size of the effect you are looking for and design your study accordingly. If time or financial constraints mean your study is underpowered, make this clear and acknowledge this limitation (or limitations) in the interpretation of your results. (This point will be elaborated in more detail in Sect. 7.2.5 on sample size calculation.)

Disclose Methods and Findings Transparently

If the intended analyses produce null findings and you move on to explore your data in other ways, say so. Null findings locked in file drawers bias the literature, whereas exploratory analyses are only useful and valid if you acknowledge the caveats and limitations.

Pre-register Your Study Protocol and Analysis Plan

Pre-registration clarifies whether analyses are confirmatory or exploratory, encourages well-powered studies and reduces opportunities for non-transparent data mining

and selective reporting. Various mechanisms for this exist (for example, the Open Science Framework, or for life science studies <https://clinicaltrials.gov/>).

Make Study Materials and Data Available

Making research materials available will improve the quality of studies aimed at replicating and extending research findings. Making raw data available will enhance opportunities for data aggregation and meta-analysis, and allow external checking of analyses and results.

Work Collaboratively to Increase Power and Replicate Findings

Combining data increases the total sample size (and therefore power) while minimizing the labor and resource impact on any one contributor. Large-scale collaborative consortia in fields such as human genetic epidemiology have transformed the reliability of findings in these fields.

5.5.6 Personal Advice

1. Be realistic about your task.
2. Plan in sufficient control/calibration experiments.
3. Take notes.
4. Store your data in a well-structured way.

Preliminary Investigations and Murphy's Law

Most investigations require more than one round of experiments and analyses. Theoretically, you state your hypothesis first, then do the experiments, and finally accept or reject the hypothesis. Done.

Most of my real investigations have been less straightforward, and typically took two rounds of experiments. Typically, I start out with an idea. After making sure that nobody else has found the solution yet, I sit down, do the first rounds of measurements, and write the analysis programs required to analyze the data. Through this, I find most of the things that can go wrong (they typically do, as stated by *Murphy's Law*: "Anything that can go wrong will go wrong."), and what I should have done differently in the first place. If the experiments are successful, that first round of investigation provides me with a "proof of principle" that my question is tractable; in addition, I also obtain data on the variability of typical responses. This allows me to obtain a reasonable estimate of the number of subjects/samples needed in order to accept or reject my hypothesis. By this time, I also know whether my experimental setup is sufficient or whether a different or better setup is required. The second round of investigations is in most cases the real thing, and (if I am lucky) provides me with enough data to publish my findings.

Calibration Runs

Measurements of data can be influenced by numerous artifacts. To control these artifacts as much as possible, one should always start and end experimental recordings

with something known. For example, during movement recordings, I try to start out by recording a stationary point, and then move it 10 cm forward, left, and up. Having a recording with exact knowledge of what is happening not only helps to detect drift in the sensors and problems in the experimental setup. These recordings also help to verify the accuracy of the analysis programs.

Documentation

Make sure that you document all the factors that may influence your results, and everything that happens during the experiment:

- The date and time of the experiment.
- Information about the experimenters and the subjects.
- The exact paradigm that you have decided on.
- Anything noteworthy that happens during the experiment.

Be as brief as possible, but take down everything noteworthy that happens during the experiment. Be especially clear about the names of the recorded data-files, as they will be the first thing you need when you analyze the data later. Often, you won't need all the details from your notes. But when you have outliers or unusual data points, these notes can be invaluable for the data analysis.

Data Storage

Try to have clear, intuitive, and practical naming conventions. For example, when you perform experiments with patients and with normals on different days, you could name these recordings “[p/n][yyyy/mm/dd]_[x].dat”, e.g., *n20210329_a*. With this convention, you have a natural grouping of your data, and the data are automatically sorted logically by their date.

Always store the raw data immediately, preferably in a separate directory. I prefer to make this directory read-only, so that I don't inadvertently delete valuable raw data. You can in most cases easily redo an analysis run. But often, you will not be able to repeat an experiment.

5.5.7 Good Study Design: Clinical Investigation Plan

Many hints for a good study design can be taken from the requirements for a *Clinical Investigation Plan (CIP)*. (All but points 9, 17, and 18 below are generally relevant for statistical studies). To design a medical study properly, a clinical investigation plan is not only advisable, but it is even required by ISO 14155-1:2003, for *Clinical investigations of medical devices for human subjects*. This norm specifies many aspects of clinical studies. It enforces the preparation of a *CIP*, specifying

1. Type of study (e.g., double-blind, and with or without control group).
2. Discussion of the control group and the allocation procedure.
3. Description of the paradigm.

4. Description and justification of primary endpoint of the study.
5. Description and justification of chosen measurement variable.
6. Measurement devices and their calibration.
7. Inclusion criteria for subjects.
8. Exclusion criteria for subjects.
9. Point of inclusion (“When is a subject part of the study?”).
10. Description of the measurement procedure.
11. Criteria and procedures for subjects who drop out.
12. Chosen sample number and level of significance, and their justification.
13. Procedure for documentation of negative effects or side-effects.
14. List of factors that can influence the measurement results or their interpretation.
15. Procedure for documentation, also for missing data.
16. Statistical analysis procedure.
17. The designation of a *monitor* for the investigation.
18. The designation of a *clinical investigator*.
19. Specifications for data handling.

Chapter 6

Distributions of One Variable



Distributions of one variable are called *univariate distributions*. (In this book, only univariate distributions will be discussed.) They can be divided into *discrete distributions*, where the observations can only take on integer values (e.g. the number of children); and *continuous distributions*, where the observation variables are continuous values (e.g. the weight of a person).

The beginning of this chapter shows how to describe and work with statistical distributions. Then the most important discrete and continuous distributions are presented.

6.1 Characterizing a Distribution

6.1.1 Distribution Center

When we have a data sample, we can characterize the center of the distribution with different parameters. Thereby the data can be evaluated in two ways:

1. By their value.
2. By their rank (i.e., their list number when they are ordered according to magnitude).

a) Mean

By default, when we talk about the mean value we refer to the arithmetic mean \bar{x} :

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n}. \quad (6.1)$$

Not surprisingly, the mean of an array x can be found with the command `np.mean`.

Real-life data often include missing values, which are in many cases replaced by `nan`'s (`nan` stands for “Not-A-Number”). For statistics of arrays which include `nan`'s, `numpy` has a number of functions starting with `nan...`. These remove the `nan`-values from a sample before executing any other computation.

```
In [1]: import numpy as np

In [2]: x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [3]: np.mean(x)
Out[3]: 4.5

In [4]: xWithNan = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, np.nan]

In [5]: np.mean(xWithNan)
Out[5]: nan

In [6]: np.nanmean(xWithNan)
Out[6]: 4.5
```

Note: `np.mean`, like many other `numpy` commands, can use an `axis` parameter to act on rows, columns, or on all data:

```
mat = [[1, 2],
       [3, 4]]
np.max(mat)           # >> 4
np.max(mat, axis=0)   # >> array([3, 4])
np.max(mat, axis=1)   # >> array([2, 4])
```

b) Median

The *median* is the value that separates the higher half from the lower half of a data sample, and therefore corresponds for an odd number of samples to the 50th percentile of the sample (see below). In contrast to the mean, the median is not affected by outlying data points. The median can be found with

```
In [7]: np.median(x)
Out[7]: 4.5
```

Note that when a distribution is symmetrical, as is the case here, the mean and the median value coincide.

c) Mode

The *mode* or *modal value* of a sample is the most frequently occurring value in that sample.

The easiest way to find the mode value is the corresponding function in `scipy.stats`, which provides value and frequency of the mode value.

```
In [8]: from scipy import stats

In [9]: data = [1, 3, 4, 4, 7]

In [10]: stats.mode(data)
Out[10]: ModeResult(mode=array([4]), count=array([2]))
```

d) Geometric Mean

In some situations, the *geometric mean* can be useful. It can be calculated via the arithmetic mean of the log of the values:

$$\text{mean}_{\text{geometric}} = \left(\prod_{i=1}^n x_i \right)^{1/n} = \exp \left(\frac{\sum_i \ln(x_i)}{n} \right). \quad (6.2)$$

Again, the corresponding function is located in `scipy.stats`:

```
In [11]: x = np.arange(1,101)
```

```
In [12]: stats.gmean(x)
Out[12]: 37.9927
```

Note that the input numbers for the geometric mean have to be positive.

6.1.2 Quantifying Variability

a) Range

The *range* is simply the difference between the highest and the lowest data value, and can be found with

```
range = np.ptp(x)
```

where *ptp* stands for “peak-to-peak”. When calculating the range of a sample, the only thing that should be watched are erroneous data points. These are often outliers, i.e., data points with a value much higher or lower than the rest of the data. Often, such points are caused by errors in the selection of the sample or in the measurement procedure.

There are a number of tests to check for outliers. One of them is to check for data which lie more than 1.5*inter-quartile-range (IQR) above or below the first/third quartile (“quartiles” are defined in the next section).

b) Percentiles

The simplest way to understand *centiles*, also called *percentiles*, is to first define the *Cumulative Distribution Function (CDF)*:

$$CDF(x) = \int_{-\infty}^x PDF(x')dx'. \quad (6.3)$$

The CDF also exist for discrete distributions, where

$$CDF(n) = \sum_{i=-\infty}^n PMF(i). \quad (6.4)$$

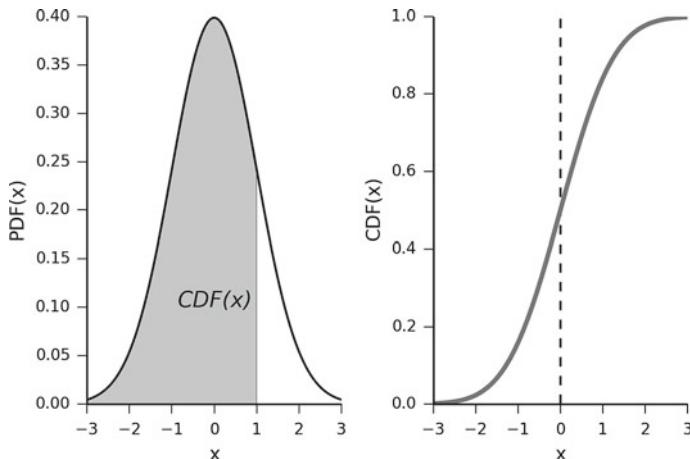


Fig. 6.1 Probability density function (left) and Cumulative distribution function (right) of a normal distribution

(For a visual summary of the relationships between PDF, CDF, and percentiles, see Fig. 4.17.) The CDF is the integral of the PDF from minus infinity up to the given value (see Fig. 6.1), and thus specifies the percentage of the data that lie below this value. Knowing the CDF simplifies the calculation of the probability of finding a value of X between a and b (Fig. 5.3): The probability to find a value between a and b is given by the integral over the PDF in that range, and can be found by the difference of the corresponding CDF-values:

$$\mathbf{P}[a < X < b] = \int_a^b P\text{DF}(x)dx = CDF(b) - CDF(a). \quad (6.5)$$

For discrete distributions the integral has to be replaced by a sum, where care has to be taken whether the endpoints should be included or not.

Coming back to *percentiles* of continuous distributions: those are just the inverse of the CDF, and give the value below which a given percentage of the data values occur (see Fig. 4.17d, ‘PPF’). (For discrete distributions, percentiles are not uniquely defined.) One will frequently encounter specific centiles:

- To find the two-tailed range which includes 95% of the data, one has to find the 2.5th and the 97.5th percentile of the sample distribution (Fig 6.3a).
- The 50th percentile is the *median*.
- Also important are the upper and lower *quartiles*, i.e., the 25th and the 75th percentile. The difference between them is called the *inter-quartile range (IQR)*.

Median, upper, and lower quartiles are used for the data display in box plots (Fig. 4.13).

In Python, the percentiles of distributions can be calculated with the *percentile point function (PPF)*. For example, the lower quartile of the standard normal distribution `stats.norm()` is given by

```
lower_quartile = stats.norm().ppf(25/100)    # -0.674
```

6.1.2.1 Standard Deviation and Variance

When calculating variance and standard deviation in Python, *numpy* uses the parameter ($n-dof$) in the divisor to distinguish between population- and sample-parameters. `dof` stands for “delta degrees of freedom”.

Attention: In *numpy* the default value for `dof` in the calculation of variance and standard deviation is set to `dof=0`, while in *pandas* the default is `dof=1`!

```
In [1]: data = np.arange(7,14)

In [2]: np.std(data, ddof=0) # population SD
Out[2]: 2.0

In [3]: np.std(data) # default = population SD!!
Out[3]: 2.0

In [4]: np.std(data, ddof=1) # sample SD, typically used
Out[4]: 2.16025

In [5]: df = pd.DataFrame(data)
...: std = df.std() # returns a pandas 'Series'
...: std.values # the corresponding numpy array
Out[5]: array([2.1602469])
```

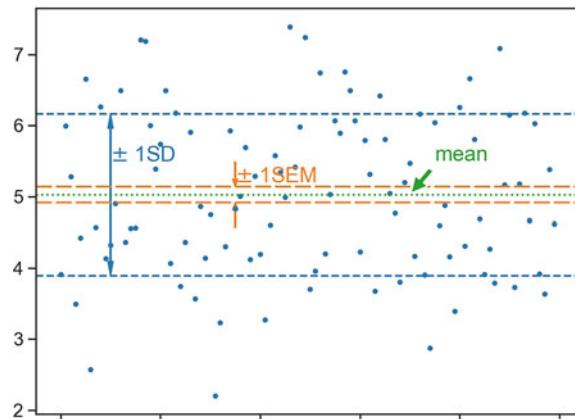
c) Standard Error

The *standard error* is the estimate of the standard deviation of a coefficient. For example, in Fig. 6.2, we have 100 data points from a normal distribution about 5. The more data points we have to estimate the mean value, the better our estimate of the mean becomes.

The *sample standard error of the mean* (SE or SEM) is

$$SEM = \frac{s}{\sqrt{n}} = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n-1}} \cdot \frac{1}{\sqrt{n}}. \quad (6.6)$$

Fig. 6.2 100 random data points, from a normal distribution of about 5. The sample mean (dotted line) is very close to the real mean. The standard deviation of the mean (long dashed line), or standard error of the mean (SEM), is 10 times smaller than the standard deviation of the samples (short dashed line)



So with 100 data points the standard deviation of our estimate, i.e., the standard error of the mean, is 10 times smaller than the sample standard deviation.

The standard error of model parameters is also typically provided with the results of linear regression fits (see p. 230).

d) Confidence Intervals

A *confidence interval (CI)* is a range estimate, and is always computed at a designated *confidence level*. The most common confidence level is 95%.

For data, the 95%-CI is the value range that contains 95% of the data.

And for unknown parameters, the 95%-CI reports the range that contains the true value for the parameter with a probability of 95%. In the statistical analysis of data, it is common to state the confidence interval of an estimated parameter.

These CIs can be *one-sided* or *two-sided* (see Fig. 6.3). An example may help to explain the difference. Assume you have produced a batch of nails, with an intended length of 10 cm. You measure the length of 100 nails, and get an empirical distribution function for them. If you ask the question “Do the produced nails match the intended length of 10 cm?” or equivalently “Is the length of the nails different from 10 cm?” you have to look if the reference value (here 10 cm) lies inside the two-tailed 95%-CI of the nails. But if you ask the question “Are the nails longer than 10 cm?”, you have to check if the lower 95%-CI is greater than 10 cm. And for the question “Are the nails shorter than 10 cm?”, you have to check if the upper 95%-CI is smaller than 10 cm.

If the sampling distribution is symmetrical and unimodal (i.e., decaying smoothly on both sides of the maximum), it will often be possible to approximate the confidence interval by

$$ci = \text{mean} \pm std * N_{PPF}\left(\frac{\alpha}{2}\right) \quad (6.7)$$

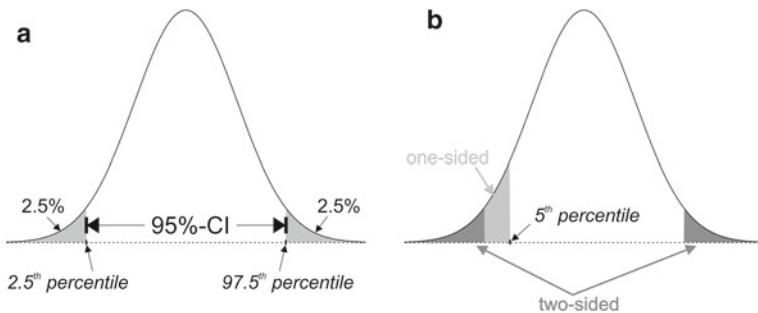


Fig. 6.3 **a** By default, confidence intervals (CIs) are “two-sided” or “two-tailed”. **b** However, in some applications “one-sided” CIs are required. Here the 95%-lower CI

where std is the standard deviation, and N_{PPF} the *percentile point function (PPF)* for the standard normal distribution (see Fig. 6.5). For the 95% two-sided confidence intervals, for example, you have to calculate the `stats.norm.ppf(0.025)` and `stats.norm.ppf(0.975)` of the standard normal distribution to get the lower and upper limit of the confidence interval. A *Python* implementation for a normal distribution is for example given on p. 122.

Notes:

- To calculate the confidence interval for the mean value, the standard deviation has to be replaced by the standard error, and the normal distribution typically with the t-distribution (Eq. 6.14).
- If the distribution is skewed, Eq. 6.7 is *not* appropriate and does not provide the correct confidence intervals, and the exact percentiles from the corresponding distribution must be used.

6.1.3 Parameters Describing the Form of a Distribution

In `scipy.stats`, continuous distribution functions are characterized by their *location* and their *scale*. To give two examples: for the normal distribution, `(location, scale)` are given by `(mean, sd)` of the distribution; and for the uniform distribution, they are given by the `(start, end)` of the range where the distribution is different from zero.

a) Location

A *location parameter* x_0 determines the location or shift of a distribution:

$$p_{x_0}(x) = p(x - x_0).$$

Examples of location parameters include the mean, the median, and the mode.

b) Scale

The *scale parameter* s describes the width of a probability distribution. If the scale parameter s is large, then the distribution will be more spread out; if s is small, then it will be more concentrated. If the probability density exists for all values of s , then the density (as a function of the scale parameter only) satisfies for continuous distributions

$$PDF_s(x) = PDF(x/s)/s$$

where PDF_s is the standardized version of the PDF .

c) Shape Parameters

It is customary to refer to all parameters beyond location and scale as *shape parameters*. Thankfully, almost all of the distributions frequently used in statistics have only one or two parameters.

Skewness Distributions are “skewed” if they depart from symmetry (Fig. 6.4, left). For example, for measurements that can only take on positive values, we can infer that the data have a skewed distribution if the standard deviation is more than half the mean. Such an asymmetry is referred to as *positive skewness*. The opposite, negative skewness, is rare.

Kurtosis Kurtosis is a measure of the “peakedness” of the probability distribution (Fig. 6.4, right). Since the normal distribution has a kurtosis of 3, the *excess kurtosis* = *kurtosis*-3 is 0 for the normal distribution. Distributions with negative or positive excess kurtosis are called *platykurtic distributions* or *leptokurtic distributions*, respectively.

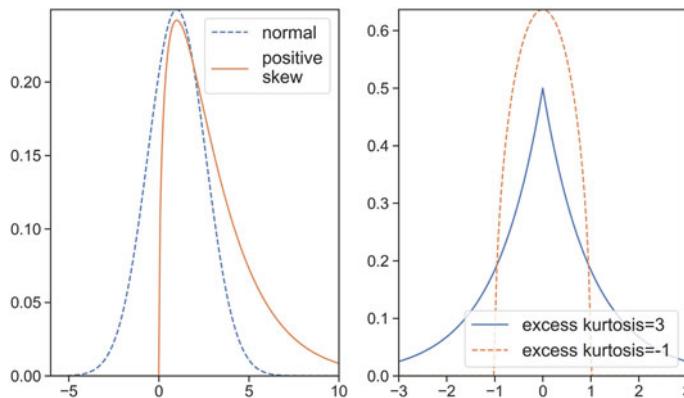


Fig. 6.4 **Left:** Normal distribution, and distribution with positive skewness. **Right:** The (leptokurtic) Laplace distribution has an excess kurtosis of 3, and the (platykurtic) Wigner semicircle distribution an excess kurtosis of -1

6.1.4 Important Methods of Probability Density Functions

Figure 6.5 shows a number of functions that are equivalent to the PDF, but each represents a different aspect of the probability distribution. I will give examples which demonstrate each aspect for a normal distribution describing the size of male subjects. The abbreviations in brackets give the name of the corresponding methods in Python.

- *Probability density function (PDF)*: Note that to obtain the probability for the variable appearing in a certain interval, you have to integrate the PDF over that range.

Example: What is the chance that a man is between 160 and 165 cm tall?

- *Cumulative distribution function (CDF)*: gives the probability of obtaining a value smaller than the given value.

Example: What is the chance that a man is less than 165 cm tall?

- *Survival Function (SF)* = 1-CDF: gives the probability of obtaining a value larger than the given value. It can also be interpreted as the proportion of data “surviving” above a certain value.

Example: What is the chance that a man is larger than 165 cm?

- *Percentile Point Function (PPF)*: the inverse of the CDF. The PPF answers the question “Given a certain probability, what is the corresponding input value for the CDF?”

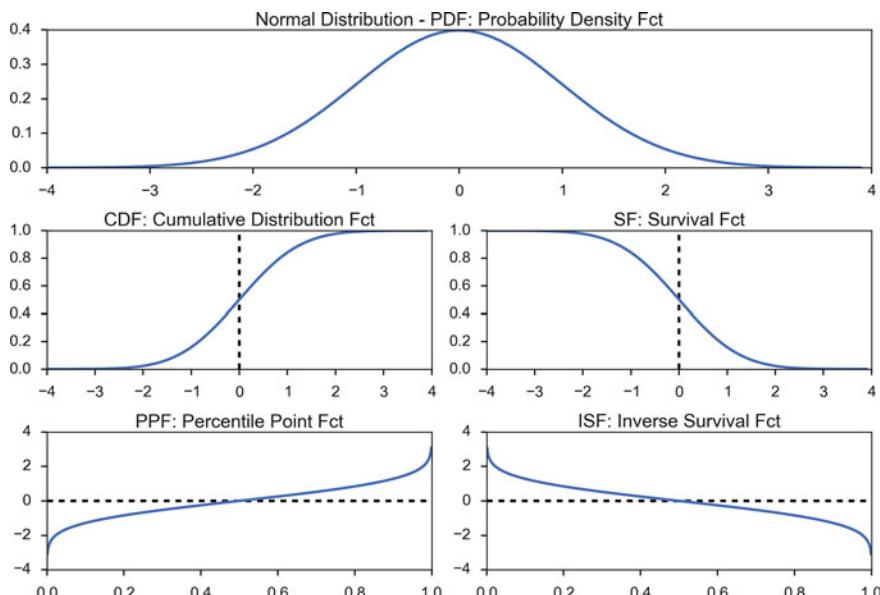


Fig. 6.5 Utility functions for continuous distributions, here for the normal distribution

Example: Given that I am looking for a man who is smaller than 95% of all other men, what size does the subject have to be?

- *Inverse Survival Function (ISF)*: the name says it all.

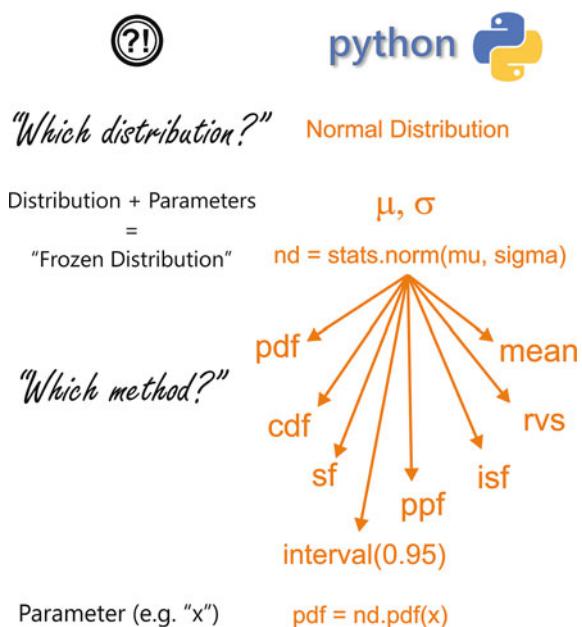
Example: Given that I am looking for a man who is larger than 95% of all other men, what size does the subject have to be?

Another frequently used method is *RVS (Random Variate Samples)*: those are random variates, i.e. random numbers, from the give distribution.

Note: In *Python*, the most elegant way of working with distribution functions is a two-step procedure (Fig. 6.6):

- In the first step, you create the distribution using all its required parameters (e.g. `nd = stats.norm(mu, sigma)`). Note that this is a distribution (in *Python* parlance a “frozen distribution”), not a function yet!
- In the second step, you decide which function you want to use from this distribution, and calculate the function value for the desired x-input (e.g. `y = nd.cdf(x)`).

Fig. 6.6 The concept of “frozen distributions”, i.e., distributions with all required parameters fixed, greatly facilitates the work with distribution functions in Python



```

import numpy as np
from scipy import stats

my_dist = stats.norm(5,3) # Create frozen distribution

x = np.linspace(-5, 15, 101)
y = my_dist.cdf(x) # Calculate corresponding CDF

```

6.2 Discrete Distributions

Discrete distributions are defined by their *Probability Mass Function (PMF)*. Two discrete distributions are frequently encountered: the *binomial distribution* (Fig. 6.7) and the *Poisson distribution* (Fig. 6.8).

The big difference between those two distributions: applications of the binomial distribution have an inherent upper limit (e.g. when you throw dice five times, each side can come up a maximum of five times); in contrast, the Poisson distribution does not have an inherent upper limit. (E.g., the answer to the question, “How many people do you know”, does not have a clear upper limit.)

A distribution similar to the binomial distribution is the *hypergeometric distribution*: but while the binomial distribution is based on independent samples, often sampling with replacement, the samples for hypergeometric distribution are *not* independent, often sampling without replacement.

Fig. 6.7 Binomial Distribution. Note that legal values exist only for integer x . The dotted lines in between only facilitate the grouping of the values to individual distribution parameters

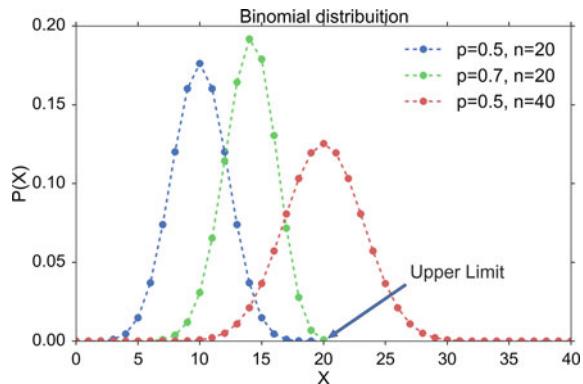
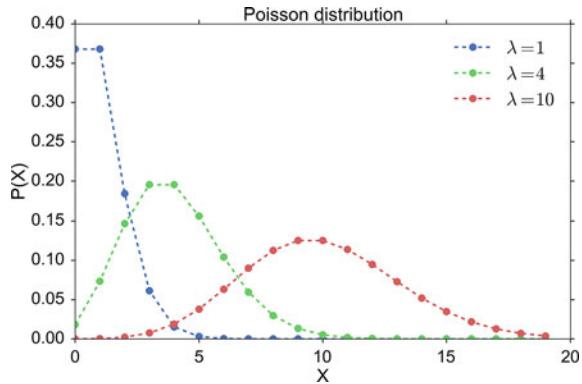


Fig. 6.8 Poisson

Distribution. Again note that legal values exist only for integer x . The dotted lines in between only facilitate the grouping of the values to individual distribution parameters



6.2.1 Bernoulli Distribution

The simplest case of a univariate distribution, and also the basis of the binomial distribution, is the *Bernoulli distribution* which has only two states, 0 and 1. It takes the value 1 with probability p , and 0 with $q = 1 - p$. So the one parameter p completely determines the distribution. An example is the simple coin flipping test. If we flip a coin (and the coin is not rigged), the chance that “heads” comes up is $p_{heads} = 0.5$.

We implement the distribution describing the coin flipping test with the commands

```
In [1]: from scipy import stats
In [2]: p = 0.5
In [3]: bernoulli_dist = stats.bernoulli(p)
```

In Python this is called a “frozen distribution function”, and it allows us to calculate everything we want for this distribution. For example, the probability if head comes up 0 or 1 times is given by the *probability mass function (PMF)*

```
In [4]: p_tails = bernoulli_dist.pmf(0)
In [5]: p_heads = bernoulli_dist.pmf(1)
```

And we can simulate 10 Bernoulli trials with

```
In [6]: trials = bernoulli_dist.rvs(10)

In [7]: trials
Out[7]: array([0, 0, 0, 1, 0, 0, 0, 1, 1, 0])
```

In line 6, `rvs` stands for *random variates*.

6.2.2 Binomial Distribution

If we flip a coin multiple times and ask “How often did heads come up?” we have the *binomial distribution*. In general, the binomial distribution is associated with the question “Out of a given (fixed) number of trials, how many will succeed?”. Some example questions that are modeled with a binomial distribution:

- Out of ten tosses, how many times will a coin land heads up?
- From the children born in a given hospital on a given day, how many of them will be girls?
- How many students in a given classroom will have green eyes?
- How many mosquitoes, out of a swarm, will die when sprayed with insecticide?

We conduct n repeated experiments, where the probability of success is given by the parameter p , and add up the number of successes. This number of successes is represented by the random variable X . The value of X is then between 0 and n .

When a random variable X has a binomial distribution with parameters p and n we write it as $X \sim B(n, p)$ and the probability mass function at $X = k$ is given by the equation:

$$P[X = k] = \begin{cases} \binom{n}{k} p^k (1-p)^{n-k} & 0 \leq k \leq n \\ 0 & \text{otherwise} \end{cases} \quad 0 \leq p \leq 1, \quad n \in \mathbb{N} \quad (6.8)$$

where $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

In Python, the procedure is the same as above for the Bernoulli distribution, with one additional parameter, the number of coin tosses. First we generate the frozen distribution function, for example, for four coin tosses:

```
In [1]: from scipy import stats
In [2]: import numpy as np

In [3]: (num, p) = (4, 0.5)
In [4]: binom_dist = stats.binom(num, p)
```

and then we can calculate for example the probabilities how often heads come up during those four tosses, given by the PMF for the values zero to four:

```
In [5]: binom_dist.pmf(np.arange(5))
Out[5]: array([0.0625, 0.25, 0.375, 0.25, 0.0625])
```

For example, the chance that heads never comes up is about 6%, and the chance that it comes up exactly once is 25%, etc.

Also note that the sum of all probabilities has to add up exactly to one:

$$p_0 + p_1 + \dots + p_{n-1} = \sum_{i=0}^{n-1} p_i = 1. \quad (6.9)$$

Example: Binomial Test

Suppose we have a board game that depends on the roll of a die and attaches special importance to rolling a 6. In a particular game, the die is rolled 235 times, and 6 comes up 51 times. If the die is fair, we would expect 6 to come up $235/6 = 39.17$ times. Is the proportion of 6's significantly higher than would be expected by chance, on the null hypothesis of a fair die?

To find an answer to this question using the *binomial test*, we consult the binomial distribution with $n = 235$ and $p = 1/6$, to determine from the PMF the probability of finding exactly 51 sixes in a sample of 235 if the true probability of rolling a 6 on each trial is 1/6. We then find the probability of finding exactly 52, exactly 53, and so on up to 235, and add all these probabilities together. In this way, we calculate the probability of obtaining *the observed result (51 sixes) or a more extreme result (> 51 sixes) assuming that the die is fair*. In this example, the result is 0.0265, which indicates that observing 51 sixes is unlikely (not significant at the 5% level) to come from a die that is not loaded to give many sixes (one-tailed test).

Clearly a die could roll too few sixes as easily as too many and we would be just as suspicious, so we should use the two-tailed test which splits the 5% probability across the two tails. This test is implemented in `scipy.stats` as `binomtest` (see also the explanation of one- and two-tailed t-tests, p. 161):

```
from scipy import stats
p = stats.binomtest(51, n=235, p=1/6)    # >> 4.4%
```



Code: `ISP_binomial.py`¹ Example of a one-and two-sided binomial test for the example described above.

6.2.3 Poisson Distribution

Any French speaker will notice that “Poisson” means “fish”, but really there’s nothing fishy about this distribution. It’s actually pretty straightforward. The name comes from the mathematician Siméon-Denis Poisson (1781–1840).

The Poisson distribution is very similar to the binomial distribution. We are examining the number of times an event happens. The difference is subtle. Whereas the binomial distribution looks at how many times we register a success over a fixed total number of trials, the Poisson distribution measures how many times a discrete event occurs, over a period of continuous space or time. There is no “total” value n , and the Poisson distribution is defined by a single parameter, the “expected value”.

The following questions can be answered with the Poisson distribution:

- How many pennies will I encounter on my walk home?
- How many children will be delivered at the hospital today?

¹ [<ISP2e>/06_Distributions/binomialTest/ISP_binomialTest.py](#).

Table 6.1 Properties of discrete distributions. (See also Sect. 9.1)

	Mean	Variance
Binomial	$n \cdot p$	$n \cdot p \cdot (1 - p)$
Poisson	λ	λ

- How many products will I sell after airing a new television commercial?
- How many mosquito bites did you get today after having sprayed with insecticide?
- How many defects will there be per 100 m of rope sold?

What's a little different about this distribution is that the random variable X which counts the number of events can take on any non-negative integer value. In other words, I could walk home and find no pennies on the street. I could also find one penny. It's also possible (although unlikely, short of an armored money transporter exploding nearby) that I would find 10 or 100 or 10,000 pennies.

Instead of having a parameter p that represents a component probability as in the binomial distribution, this time we have the parameter “lambda” or λ which represents the “average” or “expected” number of events to happen within our experiment. Equations for the mean and the variance of the binomial distribution and the Poisson distribution are given in Table 6.1. And the probability mass function of the Poisson distribution is given by

$$P[X == k] = \frac{e^{-\lambda} \lambda^k}{k!}. \quad (6.10)$$



Code: `ISP_distDiscrete.py`² shows different discrete distribution functions.

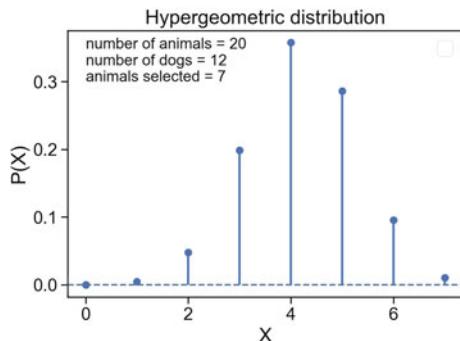
6.2.4 Hypergeometric Distribution

The name of the *hypergeometric distribution* is much more intimidating than the distribution itself. In fact it is similar to the binomial distribution, which deals with independent events, but is for *dependent* events. A simple example may help to illustrate the difference.

Take a set of two balls, a white one and a black one. When you select the first ball from this population, the chance of getting a white one is 50%. If you replace the ball, and re-select another one, the chance of getting a white one again is again 50%. But if you have selected e.g. a white ball and do *not* replace the ball, there is only the black ball left—and your chance of selecting a white one again is 0%. So there is obviously a difference between selection with replacement (-> binomial) and without replacement (-> hypergeometric).

² [ISP2e>/06_Distributions/distDiscrete/ISP_distDiscrete.py](#).

Fig. 6.9 Application of the hypergeometric distribution function. Here X is the number of dogs in the selection



Another good application example comes from the *scipy* documentation. Suppose we have a collection of 20 animals, of which 7 are dogs. Then if we want to know the probability of finding a given number of dogs if we choose at random 12 of the 20 animals, we can initialize a frozen distribution and plot the probability mass function (Fig. 6.9):

In cases where the sample size does not exceed 5% of the total population, there is little difference between the hypergeometric distribution and the binomial distribution.

6.3 Normal Distribution

The “Normal distribution” or “Gaussian distribution” is by far the most important of all the distribution functions. This is due to the fact that the mean values of *all* distribution functions approximate a normal distribution for large enough sample numbers (see Sect. 6.3.2). Mathematically, the normal distribution is characterized by a mean value μ and a standard deviation σ :

$$p_{\mu,\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \quad (6.11)$$

where $-\infty < x < \infty$, and $p_{\mu,\sigma}$ is the *Probability Density Function (PDF)* of the normal distribution. In contrast to the PMF (probability mass function) of discrete distributions, which is defined only for discrete integers, the PDF is defined for continuous values. The *standard normal distribution* is a normal distribution with a mean of zero and a standard deviation of one, and is sometimes referred to as *z-distribution*.

For smaller sample numbers, the sample distribution can show quite a bit of variability. For example, look at 25 distributions generated by sampling 100 numbers from a normal distribution (Fig. 6.11).

The normal distribution with parameters μ and σ is denoted as $N(\mu, \sigma)$. If the random variates (rvs) of X are normally distributed with expectation μ and standard deviation σ , one writes: $X \in N(\mu, \sigma)$. Figure 6.12 and Table 6.2 show the confidence intervals for one, two, and three standard deviations.

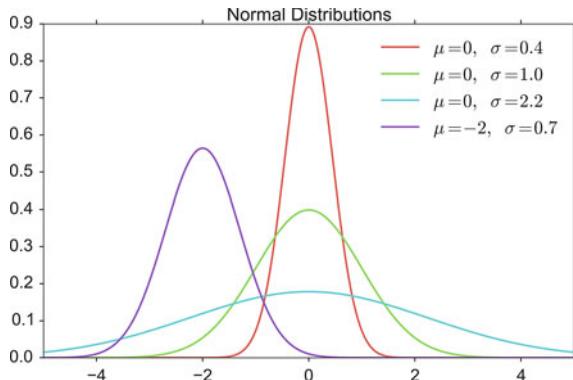


Fig. 6.10 Normal Distributions, with different parameters for μ and σ

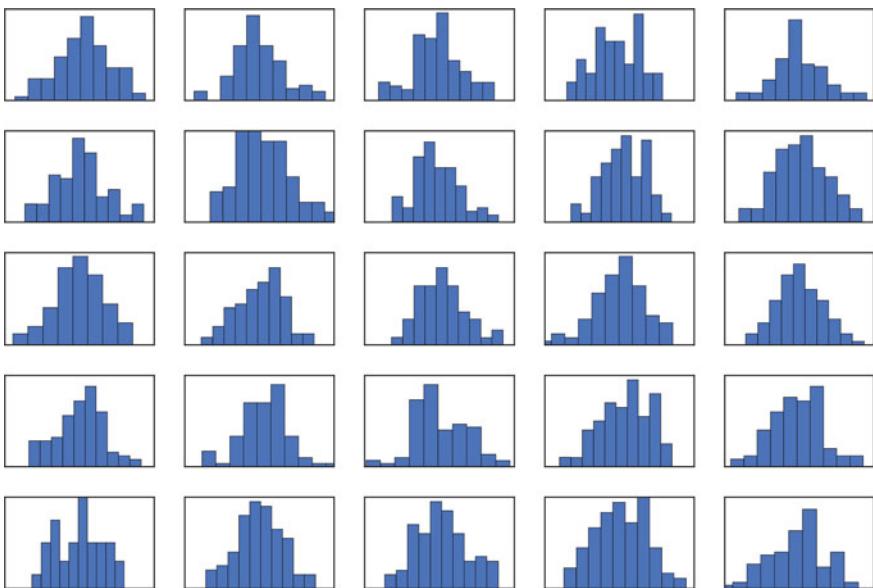


Fig. 6.11 25 randomly generated samples of 100 points from a standard normal distribution



Code: `ISP_distNormal.py`³ shows simple manipulations of normal distribution functions.

The following code example shows of how to calculate the two-tailed interval of the PDF containing 95% of the data, for the purple curve in Fig. 6.10.

³ [ISP2e>/06_Distributions/distNormal/ISP_distNormal.py](#).

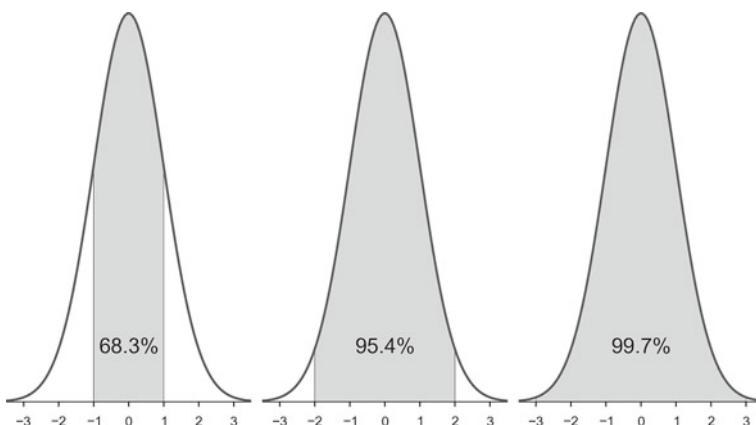


Fig. 6.12 Area under ± 1 , 2 , and 3 standard deviations of a normal distribution

Table 6.2 Tails of a normal distribution, with the distance from the mean expressed in standard deviations (sds)

Range	Probability of being	
	within range (%)	outside range (%)
mean ± 1 sds	68.3	31.7
mean ± 2 sds	95.4	4.6
mean ± 3 sds	99.7	0.27

```
In [1]: import numpy as np
In [2]: from scipy import stats

In [3]: mu = -2
In [4]: sigma = 0.7
In [5]: my_dist = stats.norm(mu, sigma)
In [6]: alpha = 0.05

In [7]: my_dist.interval(1-alpha)
Out[8]: array([-3.3720, -0.6280])
```

Sum of Normal Distributions

An important property of normal distributions is that the sum (or difference) of two normal distributions is also normally distributed, i.e., if

$$X \in N(\mu_X, \sigma_X^2)$$

$$Y \in N(\mu_Y, \sigma_Y^2)$$

$$Z = X \pm Y,$$

then

$$Z \in N(\mu_X \pm \mu_Y, \sigma_X^2 + \sigma_Y^2). \quad (6.12)$$

Or in words:

For normal distributions, the variance of the sum is the sum of the variances.

6.3.1 Examples of Normal Distributions

- If the average man is 175 cm tall with a standard deviation of 6 cm, what is the probability that a man selected at random will be 183 cm tall?
- If cans are assumed to have a standard deviation of 4 grams, what does the average weight needs to be in order to ensure that 99% of all cans have a weight of at least 250 grams?
- If the average man is 175 cm tall with a standard deviation of 6 cm, and the average woman is 168 cm tall with a standard deviation of 3 cm, what is the probability that a randomly selected man will be shorter than a randomly selected woman?

6.3.2 Central Limit Theorem

The central limit theorem states that the mean of a sufficiently large number of identically distributed random variates will be approximately normally distributed. Or in other words, the sampling distribution of the mean tends toward normality, regardless of the distribution. Figure 6.13 shows that averaging over 10 uniformly distributed data already produces a smooth, almost Gaussian distribution.



Code: `ISP_centralLimitTheorem.py`⁴ demonstrates that already averaging over 10 uniformly distributed data points produces an almost Gaussian distribution.

⁴ <[ISP2e>/06_Distributions/centralLimitTheorem/ISP_centralLimitTheorem.py](#).

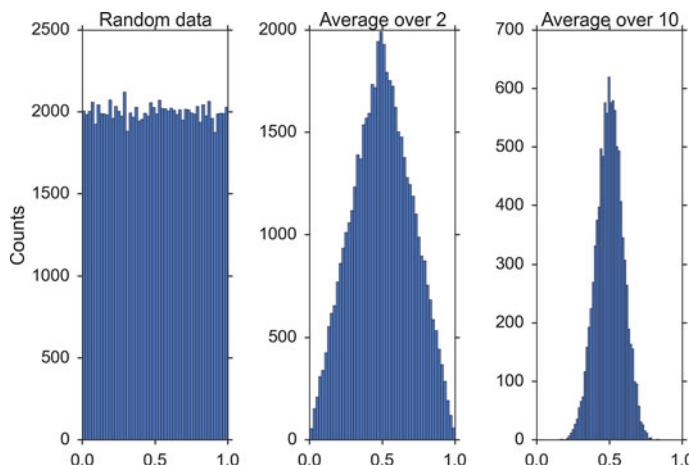


Fig. 6.13 Demonstration of the *Central Limit Theorem* for a uniform distribution: **Left:** Histogram of uniformly distributed random data between 0 and 1. **Center:** Histogram of average over two data points. **Right:** Histogram of average over 10 data points

6.3.3 Distributions and Hypothesis Tests

To illustrate the connection between distribution functions and hypothesis tests, let me go step-by-step through the analysis of the following problem:

The average weight of a newborn child in the US is 3.5 kg, with a standard deviation of 0.76 kg. If we want to check all children that are significantly different from the typical baby, what should we do with a child that is born with a weight of 2.6 kg?

We can re-phrase that problem in the form of a *hypothesis test*: our hypothesis is that *the baby comes from the population of healthy babies*. Can we keep the hypothesis, or does the weight of the baby suggest that we should reject that hypothesis?

To answer that question, we can proceed as follows:

- Find the (frozen) distribution that characterizes healthy babies $\rightarrow \mu = 3.5, \sigma = 0.76$.
- Calculate the CDF at the value of interest $\rightarrow CDF(2.6 \text{ kg}) = 0.118$. In other words, the probability that a healthy baby is at least 0.9 kg lighter than the average baby is 11.8%.
- Since we have a normal distribution, the probability that a healthy baby is at least 0.9 kg heavier than the average baby is also 11.8%.
- Interpret the result \rightarrow *If the baby is healthy, the chance that its weight deviates by at least 0.9 kg from the mean is $2 \cdot 11.8\% = 23.6\%$ - This is not significant, so we do not have sufficient evidence to reject our hypothesis, and our baby is regarded as healthy.* (see Fig. 6.14).

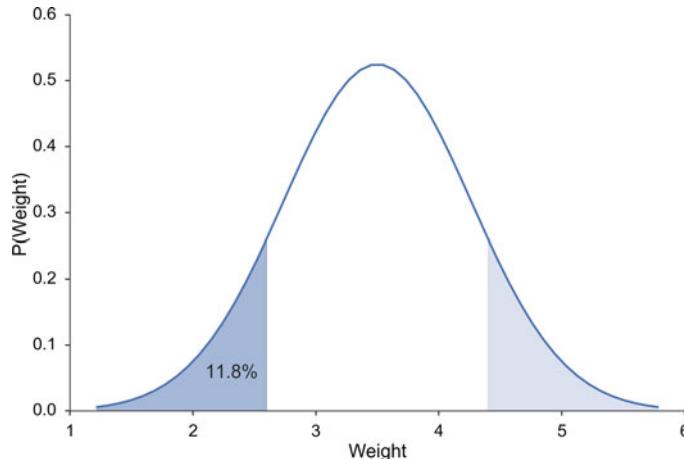


Fig. 6.14 The chance that a healthy baby weighs 2.6 kg or less is 11.8% (darker area). The chance that the difference from the mean is as extreme or more extreme than for 2.6 kg is twice that much, as the lighter area must also be considered

```
In [1]: from scipy import stats
```

```
In [2]: nd = stats.norm(3.5, 0.76)
```

```
In [3]: nd.cdf(2.6)
```

```
Out[3]: 0.11816
```

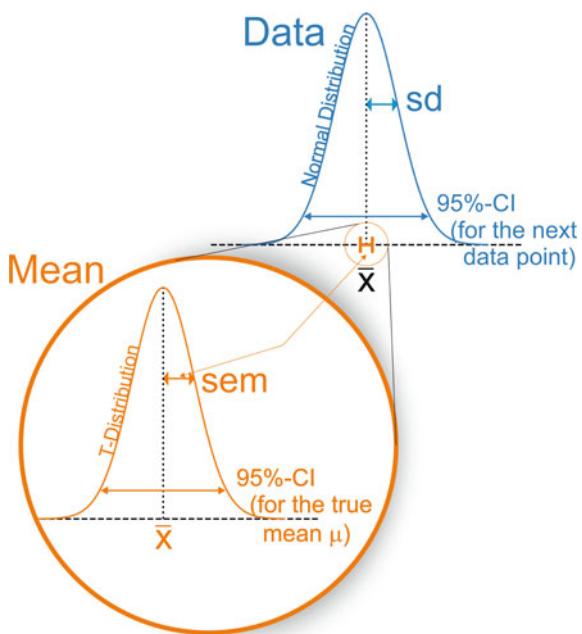
Note: The starting hypothesis is often referred to as *null hypothesis*. In our example it would mean that we assume that there is *null* difference between the distribution the baby comes from and the population of healthy babies.

6.4 Continuous Distributions Derived from the Normal Distribution

Some frequently encountered continuous distributions are closely related to the normal distribution:

- **t-distribution**—the sample distribution of mean values for samples from a normally distributed population (Fig. 6.15). Typically used for small sample numbers, when the true mean and standard deviation are not known. (If the σ is known, the distribution of the mean also follows a normal distribution.)
- **χ^2 -square distribution**—for describing variability of normally distributed data.
- **F-distribution**—for comparing variabilities of two sets of normally distributed data.

Fig. 6.15 While the normal distribution describes the variability of the data, the t-distribution describes the variability of the mean value.
Note: In the (rare) case that the *true* standard deviation σ is *exactly* known, then also the mean values are normally distributed!



In the following, we will discuss these continuous distribution functions.



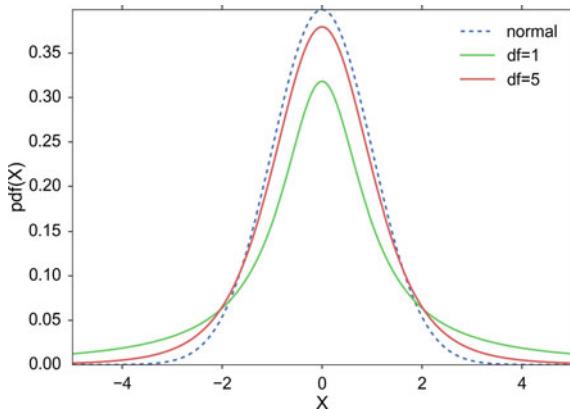
Code: `ISP_distContinuous.py`⁵ shows different continuous distribution functions.

6.4.1 T-Distribution

In 1908 W.S. Gosset, who worked for the Guinness brewery in Dublin, was interested in the problems of small samples, for example, the chemical properties of barley where sample sizes might be as low as 3. Since in these measurements the true variance of the mean was unknown, it must be approximated by the sample standard error of the mean. And the ratio between the sample mean and the standard error had a distribution that was unknown till Gosset, under the pseudonym “Student”, solved that problem. The corresponding distribution is the *t-distribution*, and converges for larger values towards the normal distribution (Fig. 6.16). Due to Gosset’s pseudonym, “Student”, it is now also known as *Student’s t-distribution*.

Since in most cases the population mean and its variance are unknown, one typically works with the t-distribution when analyzing sample data.

⁵ [<ISP2e>/06_Distributions/distContinuous/ISP_distContinuous.py](#)

Fig. 6.16 t-distribution

If \bar{x} is the sample mean, and s the sample standard deviation, the resulting statistic is

$$t = \frac{\bar{x} - \mu}{s/\sqrt{n}} = \frac{\bar{x} - \mu}{SE}. \quad (6.13)$$

Since in Eq. 6.13 the mean value is already subtracted, the t-distribution for the mean over n data has $n - 1$ degrees of freedom (DOF). A very frequent application of the t-distribution is in the calculation of confidence intervals for the mean, i.e., the interval that contains the true mean at a given confidence level α :

$$ci = mean \pm se * t_{df,\alpha}. \quad (6.14)$$

The following example shows how to calculate the t-values for the 95%-CI, for $n = 20$. The lower end of the 95% CI is the value that is larger than 2.5% of the distribution, and the upper end of the 95%-CI is the value that is larger than 97.5% of the distribution. These values can be obtained either with the *percentile point function (PPF)*, or with the *inverse survival function (ISF)*. For comparison, I also calculate the corresponding value from the normal distribution:

```
In [1]: import numpy as np
In [2]: from scipy import stats
In [3]: n = 20
In [4]: df = n-1
In [5]: alpha = 0.05

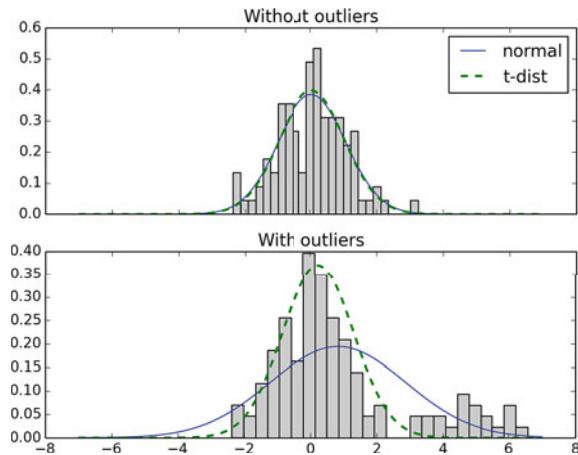
In [6]: stats.t(df).isf(alpha/2)
Out[6]: 2.093

In [7]: stats.norm.isf(alpha/2)
Out[7]: 1.960
```

In Python, the 95%-CI for the mean can be obtained with a one-line of code:

```
In [8]: df = len(data)-1
In [9]: ci = stats.t.interval(1-alpha, df,
                           loc=np.mean(data), scale=stats.sem(data))
```

Fig. 6.17 The t-distribution is much more robust against outliers than the normal distribution. Top: Best-fit normal and t-distribution, for a sample from a normal population. Bottom: Same fits, with 20 “outliers”, normally distributed data about 5, added



Since the t-distribution has longer tails than the normal distribution, it is much less affected by extreme cases (see Fig. 6.17).

6.4.2 Chi-Square Distribution

The chi-square distribution (also written as “chi2-distribution” or “ χ^2 -distribution”) is related to the normal distribution in a simple way: if a random variable X has a normal distribution ($X \in N(0, 1)$), then X^2 has a chi-square distribution, with one degree of freedom ($X^2 \in \chi_1^2$). The sum squares of n independent and standard normal random variables has a chi-square distribution with n degrees of freedom (Fig. 6.18):

$$\sum_{i=1}^n X_i^2 \in \chi_n^2. \quad (6.15)$$

Application Example

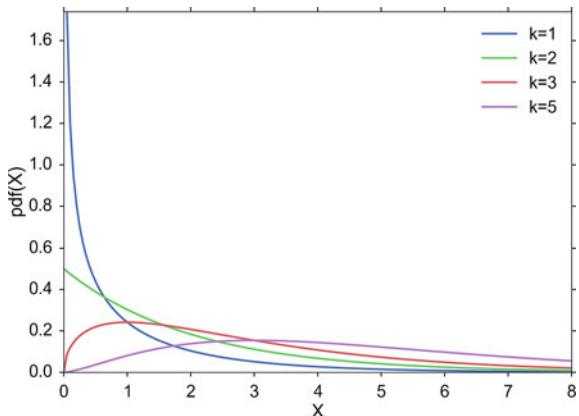
A pill producer is ordered to deliver pills with a standard deviation of $\sigma = 0.05$. From the next batch of pills $n = 13$ random samples have a weight of 3.04, 2.94, 3.01, 3.00, 2.94, 2.91, 3.02, 3.04, 3.09, 2.95, 2.99, 3.10, 3.02 g.

Question: Is the standard deviation larger than allowed?

Answer: Since the chi-square distribution describes the distribution of the summed squares of random variates from a *standard normal distribution*, we have to normalize our data before we calculate the corresponding CDF-value:

$$SF_{\chi_{(n-1)}^2} = 1 - CDF_{\chi_{(n-1)}^2} \left(\sum \left(\frac{x - \bar{x}}{\sigma} \right)^2 \right) = 0.1929. \quad (6.16)$$

Fig. 6.18 Chi-square Distribution



Interpretation: if the batch of pills is from a distribution with a standard deviation of $\sigma = 0.05$, the probability of obtaining a chi-square value as large or larger than the one observed is about 19%, so it is not atypical. In other words, the batch matches the expected standard deviation.

Note: The number of the DOF is $n - 1$, because we are only interested in the shape of the distribution, and the mean value of the n data is subtracted from all data points.

```
In [1]: import numpy as np
In [2]: from scipy import stats
In [3]: data = [3.04, 2.94, 3.01, 3.00, 2.94, 2.91, 3.02,
            3.04, 3.09, 2.95, 2.99, 3.10, 3.02]
In [4]: sigma = 0.05
In [5]: chi2Dist = stats.chi2(len(data)-1)
In [6]: statistic = sum(((data-np.mean(data))/sigma)**2)

In [7]: chi2Dist.sf(statistic)
Out[7]: 0.19293
```

6.4.3 F-Distribution

This distribution is named after Sir Ronald Fisher, who developed the F distribution for use in determining critical values in ANOVAs (“ANalysis Of VAriance”, see Sect. 8.3.1).

If we want to investigate whether two groups have the same variance, we have to calculate the ratio of the sample variances:

$$F = \frac{var_x}{var_y} \quad (6.17)$$

where var_x and var_y are the variance of the first and second sample, respectively. The distribution of this statistic is the *F-distribution*.

Application Examples

The most common application of the F-distribution is the comparison of three or more groups. In that case, the variation *between* the groups is compared to the variation *within* the groups. Appropriately, this is called *analysis of variance (ANOVA)*, and is discussed in detail in Sect. 8.3.1.

For applications in ANOVAs, the cutoff values for an F-distribution are generally found using three variables:

- ANOVA numerator degrees of freedom
- ANOVA denominator degrees of freedom
- significance level.

An ANOVA compares the size of the variance between two different samples. This is done by dividing the larger variance by the smaller variance (Eq. 6.17). The formula for the resulting F statistic is (Fig. 6.19):

$$F(r_1, r_2) = \frac{\chi_{r_1}^2 / r_1}{\chi_{r_2}^2 / r_2} \quad (6.18)$$

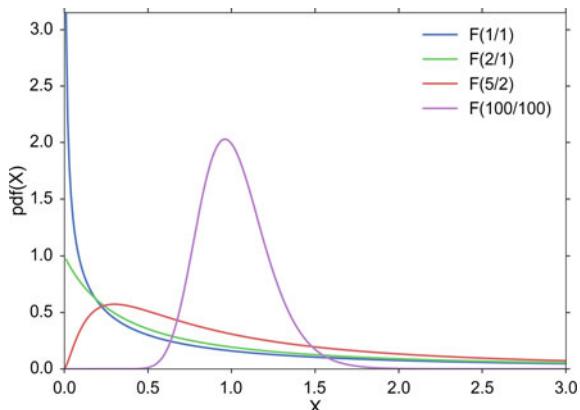
where $\chi_{r_1}^2$ and $\chi_{r_2}^2$ are the chi-square statistics of sample one and two, respectively, and r_1 and r_2 are their degrees of freedom, $r_i = n_i - 1$.

A second common application for the F-distribution is the comparison of measurement (or production) variances. Take, for example, a company that produces hip implants, and changes from an old system to a new system. For the old and the new process, we obtain samples with the following femur head diameters (in mm):

Old Method: [29.7, 29.4, 30.1, 28.6, 28.8, 30.2, 28.7, 29.]

New Method: [30.7, 30.3, 30.3, 30.3, 30.7, 29.9, 29.9, 29.9, 30.3, 30.3, 29.7, 30.3]

Fig. 6.19 F-distribution



To compare if the *precision* of the new system is as good as the old one, Eq. 6.17 is used to compare the two variances. A value of $F = 1$ would indicate that both precisions are equivalent; if the new method were more precise, F would be smaller than 1; and if the new method were less precise, F would be larger than 1. If the F -value is within the 95%-confidence interval, it would mean that the two methods are *not significantly different*.

For the example here the F statistic is $F = 0.244$, and has $n - 1$ and $m - 1$ degrees of freedom, where n and m are the number of recordings with each method. The code sample below shows that the F statistic is in the tail of the distribution ($p = 0.019$), so we reject the hypothesis that the two methods have the same precision.

Note that there is an important difference between accuracy and precision (Fig. 6.20):

For production, the *accuracy* is the difference between the intended and the produced parts, and the *precision* is the variability between the produced parts. And for measurements, the *accuracy* gives the deviation between the real and the measured value, while the *precision* is determined by the variance of the measurements. While the accuracy can often be changed easily by adjusting the system controls, the precision depends on the variability in the production or measurement process and thus is much harder to control. In the area of quality control, quality is inversely proportional to precision (Montgomery 2019).

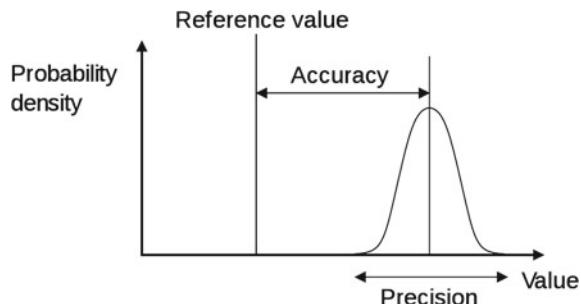
```
import numpy as np
from scipy import stats

old = [29.7, 29.4, 30.1, 28.6, 28.8, 30.2, 28.7, 29.]
new = [30.7, 30.3, 30.3, 30.3, 30.7, 29.9, 29.9,
       29.9, 30.3, 30.3, 29.7, 30.3]

f_val = np.var(new, ddof=1)/np.var(old, ddof=1) # -> F=0.244
fd = stats.f(len(new)-1, len(old)-1)
p = fd.cdf(f_val)      # -> p=0.019

if (0.025 < p < 0.975):
    print('No significant difference.')
else:
    print('There is a significant difference ' +\
          'between the two distributions.')
```

Fig. 6.20 Accuracy and precision of a measurement are two different characteristics!



6.5 Other Continuous Distributions

Some common distributions which are not directly related to the normal distribution are described briefly in the following:

- **Lognormal distribution**—A normal distribution, plotted on an exponential scale. This transformation is often used to convert a strongly skewed distribution into a normal one.
- **Weibull distribution**—Mainly used for reliability or survival data.
- **Exponential distribution**—Exponential curves.
- **Uniform distribution**—When everything is equally likely.

6.5.1 Lognormal Distribution

Normal distributions are the easiest ones to work with. In some circumstances a set of data with a positively skewed distribution can be transformed into a symmetric, normal distribution by taking logarithms. Taking logs of data with a skewed distribution will often give a distribution that is near to normal (see Fig. 6.21).

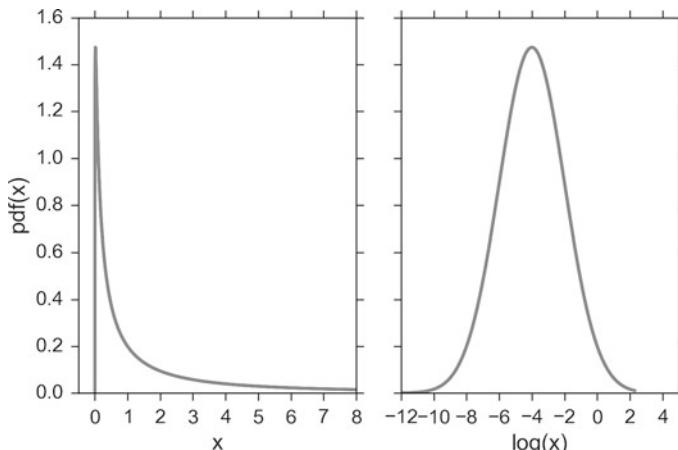


Fig. 6.21 LogNormal distribution, plotted against a linear abscissa (left) and against a logarithmic abscissa (right)

6.5.2 Weibull Distribution

The Weibull distribution is the most commonly used distribution for modeling reliability data or “survival” data (Chap. 10). It has two parameters, which allow it to handle increasing, decreasing or constant failure-rates (see Fig. 6.22). Its PDF is defined as

$$p(x) = \begin{cases} \frac{k}{\lambda} \left(\frac{x}{\lambda}\right)^{k-1} e^{-(x/\lambda)^k} & x \geq 0, \\ 0 & x < 0, \end{cases} \quad (6.19)$$

where $k > 0$ is the shape parameter and $\lambda > 0$ is the scale parameter of the distribution. (It is one of the rare cases where we use a shape parameter different from skewness and kurtosis.) Its complementary cumulative distribution function is a stretched exponential function.

If the quantity x is a “time-to-failure”, the Weibull distribution gives a distribution for which the failure rate is proportional to a power of time. The shape parameter, k , is that power plus one, and so this parameter can be interpreted directly as follows:

- A value of $k < 1$ indicates that the failure rate decreases over time. This happens if there is significant “infant mortality”, or defective items failing early and the failure rate decreasing over time as the defective items are weeded out of the population.
- A value of $k = 1$ indicates that the failure rate is constant over time. This might suggest random external events are causing mortality, or failure.
- A value of $k > 1$ indicates that the failure rate increases with time. This happens if there is an “aging” process, or parts that are more likely to fail as time goes on. An example would be products with a built-in weakness that fail soon after the warranty expires.

In the field of materials science, the shape parameter k of a distribution of strengths is known as the *Weibull modulus*.

Fig. 6.22 Weibull distribution. The scale parameters for all curves is $\lambda = 1$

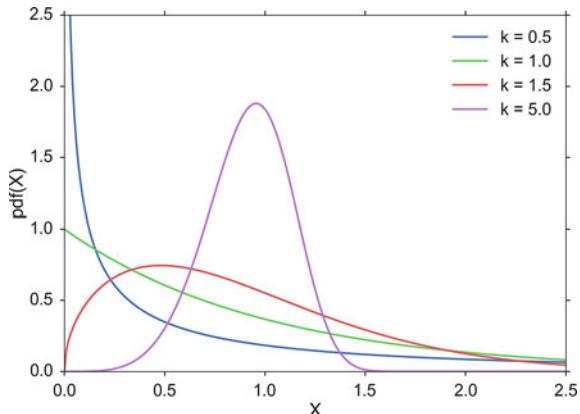
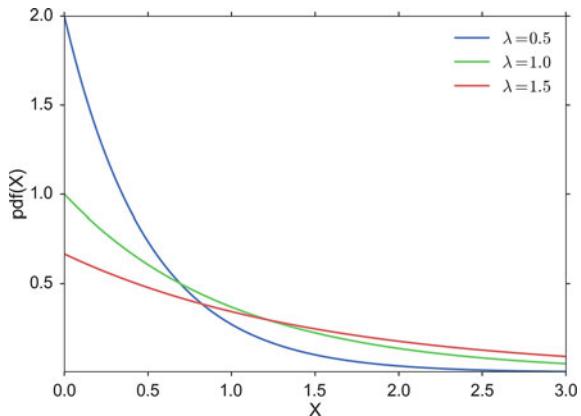


Fig. 6.23 Exponential distribution



6.5.3 Exponential Distribution

For a stochastic variable X with an exponential distribution, the PDF is:

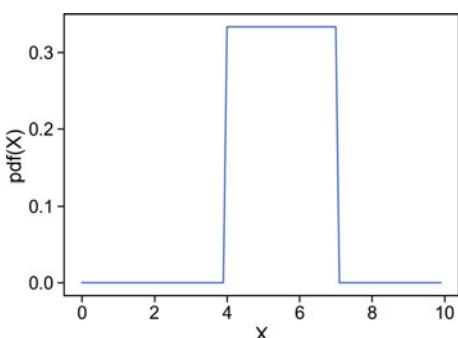
$$p(x) = \begin{cases} \lambda e^{-\lambda x}, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases} \quad (6.20)$$

The exponential PDF is shown in Fig. 6.23.

6.5.4 Uniform Distribution

This is a simple one: an even probability for all data values (Fig. 6.24). Not very common for real data.

Fig. 6.24 Uniform distribution, here with $(\text{loc, scale}) = (4, 3)$. The height of the uniform distribution is $y=1/\text{scale}$



6.6 Confidence Intervals of Selected Statistical Parameters

One important application of statistics is the area of quality control (Montgomery 2019). Depending on the goal of the “statistical process control (SPC)”, the observed values for different parameters are compared to their respective confidence intervals.

One of these parameters is the mean value of normally distributed data. As discussed in Sect. 6.4.1, the distribution of the mean value is typically characterized by the t-distribution, and its scale is determined by the standard error of the mean (Eq. 6.14). Other frequently used parameters are listed in Table 6.3.

Since the equations underlying these confidence intervals don't (immediately) convey any deeper insights, they have been listed in the Appendix (Sect. C). For practical applications the function `ISP_confidence_interval.py` should provide all the values that used to be printed in long tables at the end of classical statistics books.

For each of these parameters the `two-sided`[`default`], `upper`, `lower` confidence interval can be determined, and the default significance level is set to $\alpha = 0.05$.

The following listing gives examples for the calculation of some of these intervals:

```
import ISP_confidence_intervals as ci

# Set data and parameters
data = [89, 104.1, 92.3, 106.2, 96.3, 107.8, 102.5]
(n_obs, n_tot) = (1, 20) # for Binomial
n_expected = 12 # for Poisson

# Calculate CIs
ci_mean = ci.mean(data, ci_type='lower')
ci_s = ci.sigma(data, ci_type='upper')
ci_bin = ci.binomial(n_obs, n_tot, ci_type='two-sided')
ci_poisson = ci.poisson(n_expected, alpha=0.05)

print(f'CI-limit(s) for Poisson mean: {ci_poisson}' )
```

Since frequently confusion arises over which type of confidence interval should be calculated, the following rules-of-thumb should help in that decision, where the significance level α is typically set to $0.05 = 5\%$:

- “Are the data *different* from a certain value?”
→ `two-sided` confidence interval, which corresponds to

$$CI = \text{frozen_dist.ppf}([\frac{\alpha}{2}, 1 - \frac{\alpha}{2}]).$$

Table 6.3 The confidence intervals of the parameters marked in orange are frequently used in the area of statistical quality control

Distribution	Parameters
Normal	μ, σ
Binomial	n, p
Poisson	λ

- “Are the data data *larger than* an allowed limit?”
→ *lower* confidence limit

$$CI_{lower} = \text{frozen_dist.ppf}(\alpha)$$

$$CI = [CI_{lower}, \infty].$$

- “Are the data data *smaller than* an allowed limit?”
→ *upper* confidence limit

$$CI_{upper} = \text{frozen_dist.ppf}(1 - \alpha)$$

$$CI = [-\infty, CI_{upper}].$$



Code: `ISP_confidence_intervals.py`⁶ Confidence intervals for the normal distribution (mean, standard deviation), the binomial distribution (p), and the Poisson distribution (λ).

6.7 Exercises

1. Sample Standard Deviation

Create an numpy-array, containing the data $1, 2, 3, \dots, 10$. Calculate mean and sample(!)-standard deviation.

2. Normal Distribution

- Generate and plot the Probability Density Function (PDF) of a normal distribution, with a mean of 5 and a standard deviation of 3.
- Generate 1000 random data from this distribution.
- Calculate the standard error of the mean of these data.
- Plot the histogram of these data.
- From the PDF, calculate the interval containing 95% of these data.
- Your doctor tells you that he can use hip implants for surgery even if they are 1 mm bigger or smaller than the specified size. And your financial officer tells you that you can discard 1 out of 1000 hip implants, and still make a profit.

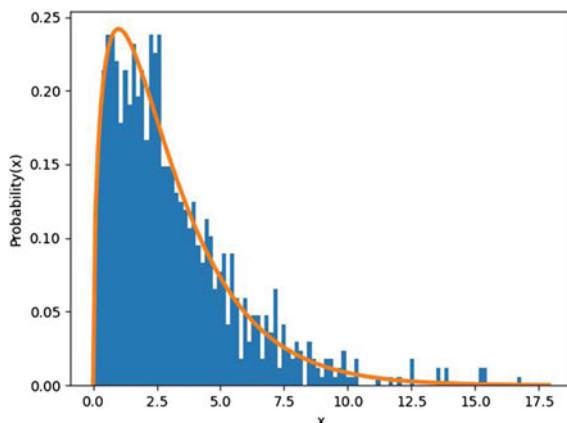
What is the required standard deviation for the producer of the hip implants, to simultaneously satisfy both requirements?

3. Continuous Distributions

- **T-Distribution:** Measuring the weight of your colleagues, you have obtained the following weights: 52, 70, 65, 85, 62, 83, 59 kg. Calculate the corresponding mean, and the 99% confidence interval for the mean. Note: with n values you have $n-1$ DOF for the t-distribution.

⁶ [ISP2e>/06_Distributions/confidenceIntervals/ISP_confidence_interval.py](https://ISP2e.readthedocs.io/en/latest/06_Distributions/confidenceIntervals/ISP_confidence_interval.py).

Fig. 6.25 Chi2 distribution with 3 degrees of freedom



- **Chi-square Distribution:** Create 3 normally distributed data sets (mean = 0, sd = 1), with 1000 samples each. Then square them, sum them (so that you have 1000 data points), and create a histogram with 100 bins. This should be similar to the corresponding curve for the Chi-square distribution, with 3 DOF (i.e., it should come down at the left, see Fig. 6.25).
- **F-Distribution:** In a production line for hip implants, an old machine is replaced with a newer model. Before the replacement, a random selection of hip-implants has heads with a diameter of [32.0, 32.5, 31.5, 32.1, 31.8] mm. Another selection taken the installation of the new model produces head sizes of [33.2, 33.3, 33.8, 33.5, 34.0] mm.

Is the precision of the two machines equal?

Note: calculate the corresponding F-value, and check if the CDF for the corresponding F-distribution is < 0.025 .

- **Uniform Distribution:** Define a uniform distribution, with a range [0, 1], and generate 1000 random variates of that distribution. Plot these data as a scatter plot. What is the 95%-CI of this distribution? What the 99.9%-CI? Try to solve this without a calculator!

4. Discrete Distributions

- **Binomial Distribution** “According to research, pure blue eyes in Europe approach greatest frequency in Finland, Sweden and Norway (at 72%), followed by Estonia, Denmark (69%); Latvia, Ireland (66%); Scotland (63%); Lithuania (61%); The Netherlands (58%); Belarus, England (55%); Germany (53%); Poland, Wales (50%); Russia, The Czech Republic (48%); Slovakia (46%); Belgium (43%); Austria, Switzerland, Ukraine (37%); France, Slovenia (34%); Hungary (28%); Croatia (26%); Bosnia and Herzegovina (24%); Romania (20%); Italy (18%); Serbia, Bulgaria (17%); Spain (15%); Georgia, Portugal (13%); Albania (11%); Turkey and Greece (10%). Further analysis shows that the average occurrence of blue eyes in Europe is 34%, with 50% in Northern Europe and 18% in Southern Europe.”

If we have 15 Austrian students in the classroom, what is the chance of finding 3, 6, or 10 students with blue eyes?

- **Poisson Distribution** In 2012 there were 62 fatal accidents on streets in Austria. Assuming that those are evenly distributed, we have on average $62 / (365/7) = 1.19$ fatal accidents per week. How big is the chance that in a given week there are no, 2, or 5 accidents?

Chapter 7

Hypothesis Tests



This chapter describes a typical workflow in the analysis of statistical data. Special attention is paid to visual and quantitative tests of normality for the data. Then the concept of a *hypothesis test* is explained, as well as the different types of errors, and the interpretation of *p-values* is discussed. Finally, the common test concepts of *sensitivity* and *specificity* are introduced and explained.

7.1 Typical Analysis Procedure

In “the old days” (before computers with almost unlimited computational power were available), the statistical analysis of data was typically restricted to hypothesis tests: you formulate a hypothesis, collect your data, and then accept or reject the hypothesis. The resulting hypothesis tests form the basic framework for by far most analyses in medicine and life sciences, and the most important hypotheses tests will be described in the following chapters.

The advent of powerful computers has changed the game. Nowadays, the analysis of statistical data is (or at least should be) a highly interactive process: you look at the data, and generate models which may explain your data. Then you determine the best-fit parameters for these models, and check these models, typically by looking at the residuals. If you are not happy with the results, you modify the model to improve the correspondence between models and data; when you are happy, you calculate the confidence intervals for your model parameters, and form your interpretation based on these values. An introduction into this type of statistical analysis based on statistical modeling is provided in Chap. 12.

In either case, one should start off with the following steps:

- Visually inspect the data
- Find extreme samples, and check them carefully

- Determine the data type of the values
- If the data are continuous, check whether or not they are normally distributed.
- Select and apply the appropriate test, or start with the model-based analysis of the data.

7.1.1 Data Screening and Outliers

The first step in data analysis is the visual inspection of the data. Our visual system is enormously powerful, and if the data are properly displayed, trends that characterize the data can be clearly visible. In addition to checking if the first and the last data values have been read in correctly, it is recommendable to check for missing data and outliers.

There is no unique definition for outliers. However, for normally distributed samples they are often defined as data that lie either more than $1.5 \cdot \text{IQR}$ (inter-quartile range, see Fig. 4.13) beyond the upper and lower quartile, or more than 2 standard deviations (see Fig. 6.12), from the sample mean. Outliers often fall in one of two groups: they are either caused by mistakes in the recording, in which case they should be excluded; or they constitute very important and valuable data points, in which case they have to be included in the data analysis. To decide which of the two is the case, you have to check the underlying raw data (for saturation or invalid data values) and the protocols from your experiments (for mistakes that may have occurred during the recording). If an underlying problem is detected, then—and only then—one may eliminate the outliers from the analysis. In every other case, the data have to be kept!

7.1.2 Normality Check

Statistical hypothesis tests can be grouped into *parametric tests* and *non-parametric tests*. Parametric tests assume that the data can be well described by a distribution that is defined by one or more parameters, in most cases by a normal distribution. For the given data set, the best-fit parameters for this distribution are then determined, together with their confidence intervals, and interpreted.

However, this approach only works if the given data set is in fact well approximated by the chosen distribution. If not, the results of the parametric test can be completely wrong. In that case non-parametric tests have to be used which are less sensitive, but therefore do not depend on the data following a specific distribution.

a) Probability-Plots

In statistics different tools are available for the visual assessments of distributions. Different graphical methods exist for comparing two probability distributions by plotting their quantiles, or closely related parameters, against each other:

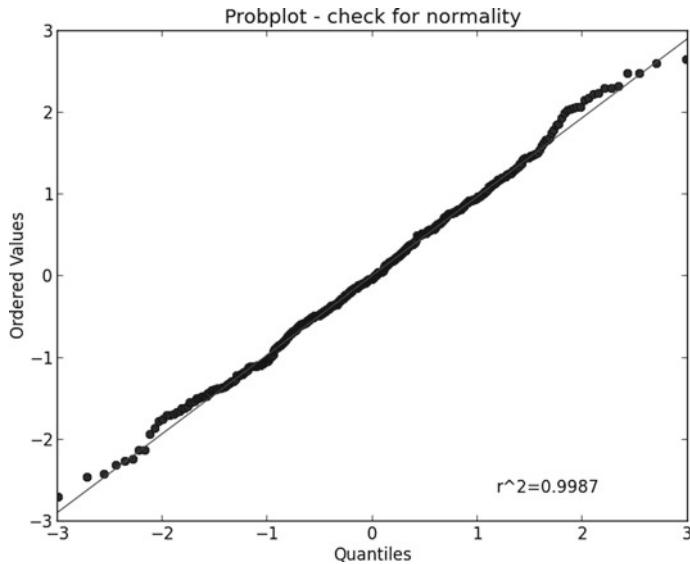


Fig. 7.1 Probability-Plot, to check for normality of a sample distribution

QQ-Plots The “Q” in QQ-plot stands for *quantile*. The quantiles of a given data set are plotted against the quantiles of a reference distribution, typically the standard normal distribution.

PP-Plots Plot the CDF (cumulative-distribution-function) of a given data set against the CDF of a reference distribution.

Probability Plots Plot the ordered values of a given data set against the quantiles of a reference distribution.

In all three cases the results are similar: if the two distributions being compared are similar, the points will approximately lie on the line $y = x$. If the distributions are linearly related, the points will approximately lie on a line, but not necessarily on the line $y = x$ (Fig. 7.1).

In Python, a probability plot can be generated with the command

```
stats.probplot(data, plot=plt)
```

and qq-plots with the pingouin command

```
pg.qqplot(data, plot=plt)
```

To understand the principle behind those plots, look at the right plot in Fig. 7.2. Here we have 100 random data points from a chi₂-distribution, which is clearly asymmetrical (Fig. 7.2, left). The x-value of the first data point is (approximately) the 1/100-quantile of a standard normal distribution (`stats.norm().ppf(0.01)`), which corresponds to -2.33 (The exact value is slightly shifted, because of a small correction, called “Filliben’s estimate”). The y-value is the smallest value of our data set. Similarly, the second x-value corresponds approximately to

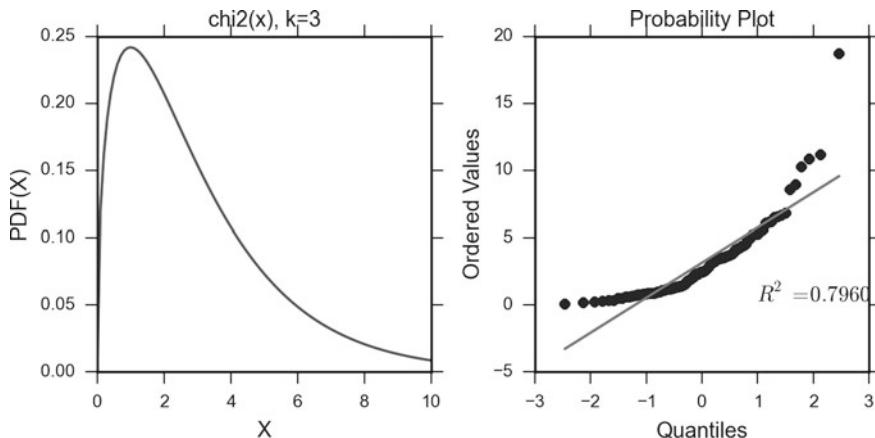


Fig. 7.2 **Left:** Probability-density-function for a Chi2-distribution ($k = 3$), which is clearly non-normal. **Right:** Corresponding probability plot

`stats.norm().ppf(0.02)`, and the second y-value is the second-lowest value of the data set, etc.

b) Tests for Normality

A number of challenges can arise in tests for normality: sometimes only few samples may be available, while other times one may have many data, but some extremely outlying values. To cope with the different situations different tests for normality have been developed. These tests to evaluate normality (or similarity to some specific distribution) can be broadly divided into two categories:

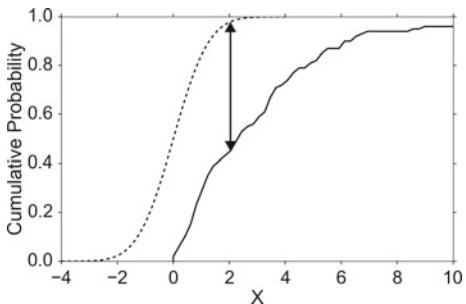
1. Tests based on comparison (“best fit”) with a given distribution, often specified in terms of its CDF. Examples are the Kolmogorov-Smirnov test, the Lilliefors test, the Anderson–Darling test, the Cramer–von Mises criterion, as well as the Shapiro–Wilk and Shapiro–Francia tests.
2. Tests based on descriptive statistics of the sample. Examples are the skewness test, the kurtosis test, the D’Agostino-Pearson omnibus test, or the Jarque–Bera test.

For example, the Lilliefors test, which is based on the Kolmogorov–Smirnov test (see Fig. 7.3), quantifies a distance between the empirical distribution function of the sample and the cumulative distribution function of the reference distribution, or between the empirical distribution functions of two samples. (The original Kolmogorov–Smirnov test should not be used if the number of samples is ca. ≤ 300 .)

The Shapiro–Wilk W–test, which depends on the covariance matrix between the order statistics of the observations, can also be used with ≤ 50 samples, and has been recommended by Altman (1999) and by Ghasemi and Zahediasl (2012).

The `pingouin` command `pg.normality(x)` uses by default the Shapiro–Wilk test. With the option `method=normaltest` it switches to the `scipy` command

Fig. 7.3 Illustration of the Kolmogorov–Smirnov statistic. The dashed line is the CDF for the normal distribution, the solid line is the empirical CDF for a chi²-distribution (Fig. 7.2), and the black arrow is the K-S statistic which is integrated



`stats.normaltest(x)` with the D’Agostino-Pearson *omnibus test*. That test combines a skewness and kurtosis test to produce a single, global “omnibus” statistic.

Below the output of the Python-module `ISP_checkNormality.py` is shown, which checks 1000 random variates from a normal distribution for normality. Note that while for the full data set all tests correctly indicate that the underlying distribution is normal, the effects of extreme values strongly depend on the type of test if only the first 100 random variates are included. Note that a p-value of 1 indicates a high probability that the null hypothesis—that the data are normally distributed—is true.

```
p-values for all 1000 data points: -----
Omnibus           0.913684
Shapiro--Wilk    0.558346
Lilliefors        0.569781
Kolmogorov--Smirnov 0.898967
```

```
p-values for the first 100 data points: -----
Omnibus           0.004530
Shapiro--Wilk    0.047102
Lilliefors        0.183717
Kolmogorov--Smirnov 0.640677
```

 **Code:** `ISP_checkNormality.py`¹ shows how to check graphically, as well as with different quantitative tests, if a given distribution is normal.

7.1.3 Transformation

If the data deviate significantly from a normal distribution, it is sometimes possible to make the distribution approximately normal by transforming the data. For exam-

¹ [`<ISP2e>/07_CheckNormality_CalcSamplesize/checkNormality/ISP_checkNormality.py`](#).

ple, data often have values that can only be positive (e.g., the size of persons), and that have long positive tail: such data can often be made normal by applying a log transform. This is demonstrated in Fig. 6.21.

7.2 Hypothesis Tests and Power Analyses

7.2.1 An Example

Assume that you are producing small bags of chocolate cookies. The contract with the supermarket says that your bags must contain 110 g of cookies per bag.

An inspection of 10 bags produce the following weights (Fig. 7.4):

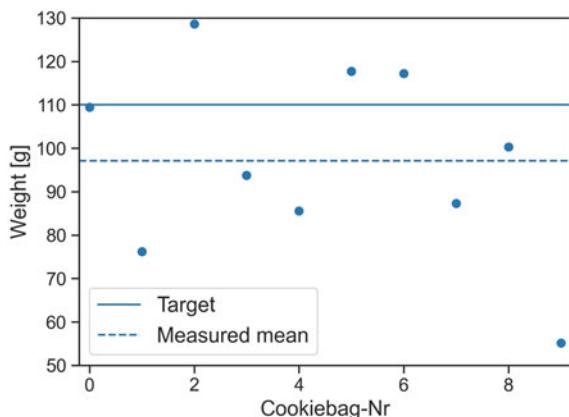
```
In [1]: import numpy as np
In [2]: weights = np.array([ 109.4, 76.2, 128.7, 93.7, 85.6,
                           117.7, 117.2, 87.3, 100.3, 55.1])
```

The question we want to answer: Is the mean weight of cookies per bag (97.1) significantly different from 110?

A *normality test* (`stats.normaltest(weights)`) indicates that the data are probably taken from a normal distribution. Since we don't know the population variance of the weights, we have to take our best guess, the sample variance (see also Fig. 5.1). And we know that the normalized difference between sample and the population mean, the t-statistic, follows the t-distribution (Eq. 6.13).

The difference between our sample mean and the value we want to compare it to, `np.mean(weights) - 110`, is -12.9 . Normalized by the sample standard error (Eq. 6.13), this gives a value of $t = -1.84$. Since the t-distribution is a known curve

Fig. 7.4 The question we ask: based on our sample mean (dashed line) and the observed variance of the data (the sample variance), do we believe that the population mean is different from 110 (solid line)? Or in other words: our null hypothesis is that the difference between the population mean and 110 is zero. Can we keep our null hypothesis, or do we have to reject it based on the data?



that depends only on the number of samples, we can calculate the probability that we obtain a t-statistic of $|t| > 1.84$:

```
In [3]: tval = (110-np.mean(weights))/stats.sem(weights)
      # 1.84
In [4]: td = stats.t(len(weights)-1)
      # "frozen" t-distribution
In [5]: p = 2*td.sf(tval)
      # 0.0995
```

The factor 2 in the last line of the code is required, since we have to combine the probability of $t < -1.84$ and $t > 1.84$. Expressed in words, given our sample data, we can state that the probability that the population mean is 110 is 9.95%. But since a *statistical difference* is only given by convention if the probability is less than 5%, we conclude that the observed value of 97.1 is not significantly different from 110, and your cookies can continue to be sold in the supermarket.

7.2.2 Generalization and Applications

a) Analysis Steps

Based on the previous example, the general procedure for hypothesis tests can be described as follows (the sketch in Fig. 5.1 indicates the meaning of many upcoming terms):

- A random sample is drawn from a population. (In our example, the random sample is our *weights*).
- A null hypothesis is formulated. (“There is null difference between the mean cookie weight and the value of 110.”)
- A test statistic is calculated, of which we know the probability distribution. (Here it is the normalized sample mean, since we know that the mean value of samples from a normal distribution follows the t-distribution.)
- Comparing the observed value of the statistic (here the obtained t-value) with the corresponding distribution (the t-distribution), we can find the probability that “a value as extreme as or more extreme than the observed one is found by chance”. This is the so-called *p-value*.
- If the p-value is $p < 0.05$, we reject the null hypothesis, and speak of a *statistically significant difference*. If a value of $p < 0.001$ is obtained, the result is typically called *highly significant*. The critical region of a hypothesis test is the set of all outcomes which cause the null hypothesis to be rejected.

In other words, the p-value states how likely it is to obtain a value as extreme or more extreme by chance alone, *if the null hypothesis is true*.

The value against which the p-value is compared is the *significance level*, and is often indicated with the letter α . The significance level is a user choice, and typically set to 0.05.

This way of proceeding to test a hypothesis is called *statistical inference*.

Remember, p only indicates the probability of obtaining a certain value for the test statistic if the null hypothesis is true—nothing else!

And keep in mind that improbable events do happen, even if not very frequently. For example, back in 1980 a woman named Maureen Wilcox bought tickets for both the Rhode Island lottery and the Massachusetts lottery. And she got the correct numbers for both lotteries. Unfortunately for her, she picked all the correct numbers for Massachusetts on her Rhode Island ticket, and all the right numbers for Rhode Island on her Massachusetts ticket. Seen statistically, the p-value for such an event would be extremely small—but it did happen anyway.

b) Additional Examples

Example 1: Let us compare the weight of two groups of subjects. The null hypothesis is that there is no difference in the weight between the two groups. If a statistical comparison of the weight produces a p-value of 0.03, this means that the probability that the null hypothesis is correct is 0.03, or 3%. Since this probability is quite low, we say that “there is a significant difference between the weight of the two groups”.

Example 2: If we want to check the assumption that the mean value of a group is 75 kg, then the corresponding null hypothesis would be: “We assume that there is no difference between the mean value in our population and the value 75”.

Example 3—Test for Normality: If we check if a data sample is normally distributed, the null hypothesis is that “there is no difference between my data and normally distributed data”: here a large p -value indicates that the data are in fact normally distributed!

7.2.3 *The Interpretation of the P-Value*

Stating a p-value alone is no longer state of the art for the statistical analysis of data. In addition, also the confidence intervals for the parameters under investigation should be given.

To reduce errors in the data interpretation, research is sometimes divided into *exploratory research* and *confirmatory research*. Take for example the case of Matt Motyl, a Psychology Ph.D. student at the University of Virginia. In 2010, data from his study of nearly 2000 people indicated that political moderates saw shades of grey more accurately than people with more extreme political opinions, with a p-value of 0.01. However, when he tried to reproduce the data, the p-value dropped down to 0.59. So while the *exploratory research* showed that a certain hypothesis may be likely, the *confirmatory research* showed that the hypothesis did not hold (Nuzzo 2014).

An impressive demonstration of the difference between exploratory and confirmatory research is a collaborative science study, where 270 researchers set down and tried to replicate the findings of 100 experimental and correlational studies pub-

lished in three leading psychology journals in 2008. While 97% of the studies had statistically significant results, only 36% of replications were statistically significant (OSC 2015)!

Sellke et al. (2001) has investigated this question in detail, and recommends to use a “calibrated p-value” to estimate the probability of making a mistake when rejecting the null hypothesis, when the data produce a p-value p :

$$\alpha(p) = \frac{1}{1 + \frac{1}{-e^p \log(p)}} \quad (7.1)$$

with $e = \exp(1)$, and \log the natural logarithm. For example, $p = 0.05$ leads to $\alpha = 0.29$, and $p = 0.01$ to $\alpha = 0.11$. However, I have to admit that I have not seen that idea applied in practical research.

A value of $p < 0.05$ for the null hypothesis has to be interpreted as follows: If the null hypothesis is true, the chance to find a test statistic as extreme or more extreme than the one observed is less than 5%.

This is not the same as saying that the null hypothesis is false, and even less so, that an alternative hypothesis is true!

7.2.4 Types of Errors

In hypothesis testing, two types of errors can occur:

a) Type I Errors

Type I errors are errors where the result is significant despite the fact that the null hypothesis is true.² The probability of a type I error is commonly indicated with α , and is set before the start of the data analysis. In quality control, a type I error is called *producer risk*, because you reject a batch of items despite the fact that they actually meet agreed requirements.

In Fig. 7.8, a type I error would be a diagnosis of cancer (“positive” test result), even though the subject is healthy.

Example Assume that the population of young Austrian adults has a mean IQ of 105 (i.e., if Austrian males were smarter than the rest) and a standard deviation of 15. We now want to check if the average student in Linz has the same IQ as the average Austrian, and we select 20 students. We set the significance level to $\alpha = 0.05$, i.e., we want to check if our sample mean lies within or outside the 95% confidence interval. Let us now assume that the average student from Linz has in fact the same IQ as the average Austrian. In that case, if we repeat our study 20 times, we will

² The way I personally remember this: we *first* start out with *normals*, so a type I error is when normals are classified incorrectly.

find on average one of those 20 times that our sample mean is significantly different from the Austrian average IQ. Such a finding would be a false result, despite the fact that our assumption is correct, and would constitute a type I error.

b) Type II Errors and Test Power

If we want to answer the question “How much chance do we have to reject the null hypothesis when the alternative is in fact true?”, or in other words, “What’s the probability of detecting a real effect?”, we are faced with a different problem. To answer these questions, we need an *alternative hypothesis*.

Type II errors are errors where the result is *not* significant, despite the fact that the null hypothesis is false. In quality control, a type II error is called a *consumer risk*, because the consumer obtains an item that does not meet the regulatory requirements.

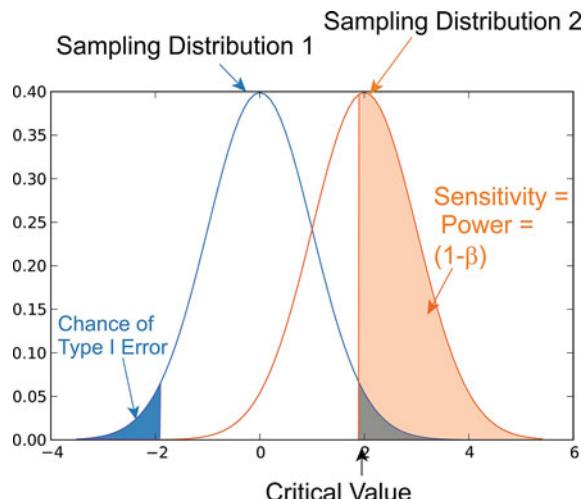
In Fig. 7.8, a type II error would be a “healthy” diagnosis (“negative” test result), even though the subject has cancer.

The probability for this type of error is commonly indicated with β . The “power” of a statistical test is defined as $(1 - \beta) * 100$, and is the chance of correctly accepting the alternative hypothesis. Or expressed with healthy subjects and patients: *power* is the percentage of patients that are correctly identified as sick. Figure 7.5 shows the meaning of the power of a statistical test. Note that for finding the power of a test, you need an alternative hypothesis (or a “patient group”).

c) Pitfalls in the Interpretation of P-Values

P-values measure evidence against a hypothesis. Unfortunately, they are often incorrectly viewed as an error probability for rejection of the hypothesis, or, even worse, as the posterior probability (i.e., after the data have been collected) that the hypothesis is true. As an example, take the case where the alternative hypothesis is that the

Fig. 7.5 Power of a statistical test, for comparing the mean value of two sampling distributions



mean is just a fraction of one standard deviation larger than the mean under the null hypothesis: in that case, a sample that produces a p-value of 0.05 may just as likely be produced if the alternative hypothesis is true as if the null hypothesis is true!

7.2.5 Sample Size

The calculation of sample size is arguably the most underestimated aspect of experimental design. For example, a (highly recommendable) article by Button et al. retrospectively checked the power of neuroscience studies that were published in 2011 (Button et al. 2013). They found that the median statistical power of the studies tested was only 21%! Empirical evidence from other fields has confirmed this finding (Ioannidis 2005). This has daunting consequences: first, four out of five of these studies would have missed a genuine effect; and second, found and published effects would tend to be significantly too large, because of a so-called “winner’s curse” (Button et al. 2013). On top of that, the lower the power of a study, the lower the probability that an observed “significant” effect actually reflects a true effect. (One can show that a lower power also implies a low *positive predictive value*, *PPV*, see below.)

This finding was so stunning that I had to replicate it with a simulation, which you can find in the *ISP2e*-repository.



Code: `7_powerAnalysis.ipynb`³: Power analysis for a study with a power of 25%.

The *power* or *sensitivity* of a binary hypothesis test is the probability that the test correctly rejects the null hypothesis when the alternative hypothesis is true (“percentage of patients recognized as sick”).

The determination of the power of a statistical test, and the calculation of the minimum sample size required to reveal an effect of a given magnitude, is called *power analysis*. It involves four factors:

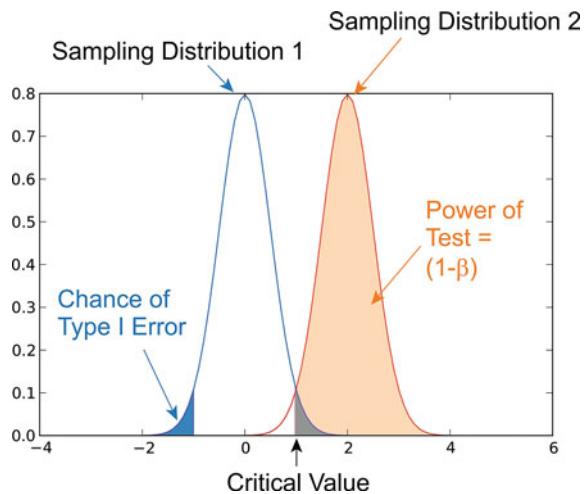
1. α , the probability for type I errors
2. β , the probability for type II errors (\Rightarrow power of the test)
3. d , also called “Cohen’s d ”: the effect size, i.e., the magnitude of the investigated effect relative to σ , the standard deviation of the sample
4. n , the sample size

Only 3 of these 4 parameters can be chosen, the 4th is then automatically fixed. For example, Fig. 7.6 shows that the power of a test goes up as the number of samples is increased.

The absolute size of the difference $D (= d * \sigma)$ between mean treatment outcomes that will answer a clinical question being posed is often called *clinical significance* or *clinical relevance*. For example, the motor function of arms and hands

³ [<ISP2e>/ipynbs/7_powerAnalysis.ipynb](https://ISP2e.ipynbs/7_powerAnalysis.ipynb).

Fig. 7.6 Effect of an increase in sample size on the power of a test, in comparison to the situation in Fig. 7.5



after a stroke is commonly graded with the *Fugl-Meyer Assessment of the Upper Extremities*, which has a maximum score of 66. In 2014, a multi-center study of task-specific robot therapy of the arm after stroke showed that this therapy leads to a significant improvement of 0.78 points (Klamroth-Marganska et al. 2014). But while this improvement is significant, it is so small that it does not make any difference in the treatment of the patient, and therefore is of no clinical relevance.

a) Examples of Power Analysis

The calculation of sample size is a surprisingly large topic, with whole books (e.g. Chow 2008) and websites (e.g. <https://sample-size.net> or <http://powerandsamplesize.com>) dedicated to it. Even for seemingly trivial cases, the exact analysis quickly becomes involved (Fig. 7.7).

Here we only provide two approximate solutions, using the normal distribution to approximate the t-distribution. The resulting numbers slightly underestimate the correct results. The Python examples in the next section provide the exact values.

Test on One Mean If we have the hypothesis that the population we draw our samples from has a mean value of x_1 and a standard deviation of σ , and the actual population has a mean value of $x_1 + D$ and the same standard deviation, we can detect a “two-sided” difference with a minimum sample number of

$$n = \frac{(z_{1-\alpha/2} + z_{1-\beta})^2}{d^2}. \quad (7.2)$$

Here z is the standardized normal variable, and $z_{...}$ refers to the PPF of a standard normal distribution (see also Sect. 6.3)

$$z = \frac{x - \mu}{\sigma} \quad (7.3)$$

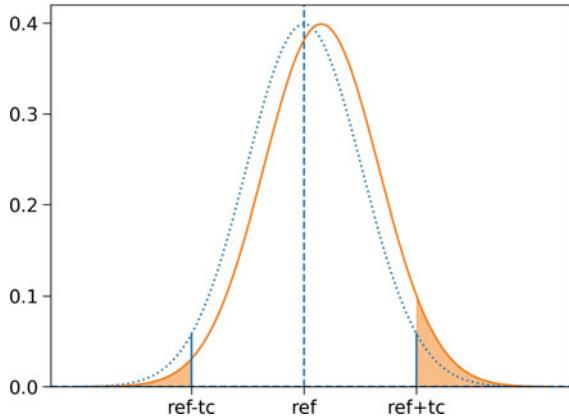


Fig. 7.7 Definition of “power” The *null hypothesis* for comparing the sample mean to a value `ref` is that the mean forms a t-distribution about the reference value `ref` (dotted line). If the true mean forms a (non-central) t-distribution about the measured mean \bar{x} , the orange shaded areas indicate the part of the sample distribution that are *more extreme than the critical value tc* , and would therefore be identified as *different from `ref`*. This is the definition of the “power of the test”, for a “two-sided” alternative hypothesis”

and $d = \frac{D}{\sigma}$ the effect size, sometimes also referred to as “Cohen’s d”.

In words, if the real mean has a value of x_1 , we want to detect this correctly in at least $1 - \alpha\%$ of all tests; and if the real mean is shifted by D or more, we want to detect this with a probability of at least $1 - \beta\%$.

Test Between Two Different Populations For finding a difference between two normally distributed means with standard deviations of σ_1 and σ_2 , the minimum number of samples we need in each group to detect an absolute difference D is

$$n_1 = n_2 = \frac{(z_{1-\alpha/2} + z_{1-\beta})^2 (\sigma_1^2 + \sigma_2^2)}{D^2}. \quad (7.4)$$

b) Python Solutions

The commands `tt_ind_solve_power` and `tt_solve_power` in the package *statsmodels*, and `power_ttest` from the package *pingouin*, make clever use of the fact that three of the four factors mentioned above are independent, and combine it with the Python feature of “named parameters” to provide a program that takes three of those parameters as input, and calculates the remaining 4th parameter. For example,

```
In [1]: from statsmodels.stats import power
In [2]: nobs = power.tt_ind_solve_power(effect_size=0.5,
                                      alpha=0.05, power=0.8)
# equivalent pingouin command:
# nobs = pg.power_ttest(d=0.5, alpha=0.05, power=0.8, n=None)
```

```
In [3]: print(np.ceil(nobs))
Out[3]: 64
```

tells us that if we compare two groups with the same number of subjects and the same standard deviation, require $\alpha = 0.05$ and a test power of 80%, and we want to detect a difference between the groups that is half the standard deviation, we need to test 64 subjects.

Similarly,

```
In [4]: effect_size = power.tt_ind_solve_power(alpha=0.05,
                                              power=0.8, nobs1=25)
In [5]: np.round(effect_size, decimals=2)
Out[5]: 0.81
```

tells us that if we have $\alpha = 0.05$, a test power of 80%, and 25 subjects in each group, then the smallest significant difference between the groups is 81% of the sample standard deviation.

The corresponding command for one-sample t-tests is `tt_solve_power`. Note that *pingouin*, and the module `stats.power` from *statsmodels*, and provide many additional commands for power analysis. For example, the package *pingouin* offers corresponding commands also for ANOVAs, chi2-tests, and correlation tests (see <https://pingouin-stats.org/api.html#power-analysis>).

c) Programs: Sample Size



Code: `ISP_sampleSize.py`⁴: direct sample size calculation for normally distributed data with arbitrary standard deviations, for detecting changes within a group, and for comparison of two independent groups with different variances; more flexible than the *statsmodel* function `power.tt_in_solve_power`.

7.3 Sensitivity and Specificity

Some of the more confusing terms in statistical analysis are *sensitivity* and *specificity*. A related topic are the *positive predictive value (PPV)* and the *negative predictive value (NPV)* of statistical tests. The following diagram shows how the four are related:

Sensitivity Is equivalent to the *power* described in the previous section. Proportion of positives that are correctly identified by a test (= probability of a positive test, given the subject is ill).

Specificity Proportion of negatives that are correctly identified by a test (= probability of a negative test, given that subject is well).

Positive Predictive Value (PPV) Proportion of patients with positive test results who are correctly diagnosed.

⁴ https://ISP2e/07_CheckNormality_CalcSamplesize/sampleSize/ISP_sampleSize.py.

		Condition		
		Condition Positive	Condition Negative	
Test Outcome	Test Outcome Positive	True Positive	False Positive (Type I error)	Positive predictive value = $\frac{\sum \text{True Positive}}{\sum \text{Test Outcome Positive}}$
	Test Outcome Negative	False Negative (Type II error)	True Negative	Negative predictive value = $\frac{\sum \text{True Negative}}{\sum \text{Test Outcome Negative}}$
	Sensitivity = $\frac{\sum \text{True Positive}}{\sum \text{Condition Positive}}$	Specificity = $\frac{\sum \text{True Negative}}{\sum \text{Condition Negative}}$		

Fig. 7.8 Relationship between sensitivity, specificity, positive predictive value and negative predictive value

Negative Predictive Value (NPV) Proportion of patients with negative test results who are correctly diagnosed.

For example, pregnancy tests have a high sensitivity: when a woman is pregnant, the probability that the test is positive is very high.

In contrast, an indicator for an attack with atomic weapons on the White House should have a very high specificity: if there is no attack, the probability that the indicator is positive should be very, very small.

While sensitivity and specificity characterize a test and are independent of prevalence, they do not indicate what portion of patients with abnormal test results are truly abnormal. This information is provided by the positive/negative predictive value (PPV/NPV). These are the values relevant for a doctor diagnosing a patient: when a patient has a positive test result, how likely is it that the patient is in fact sick? Unfortunately, as Fig. 7.9 indicates, these values are affected by the prevalence of the disease. The *prevalence* of a disease indicates how many out of 100'000 people are affected by it; in contrast, the *incidence* gives the number of newly diagnosed cases per 100'000 people (e.g. in the last week, or in the last year). In summary, we need to know the prevalence of the disease as well as the PPV/NPV of a test to provide a sensible interpretation of medical test results.

Take, for example, a test where a positive test result implies a 50% chance of having a certain medical condition (PPV = 50%). If half the population has this condition, a positive test result tells the doctor utterly nothing. But if the condition is very rare, a positive test result indicates that the patient has a fifty-fifty chance of having this rare condition—a piece of information that is very valuable.

Figure 7.9 shows how the prevalence of a disease affects the interpretation of diagnostic results, with a test with a given specificity and sensitivity: a high prevalence of the disease increases the PPV of the test, but decreases the NPV, and a low prevalence does exactly the opposite. Figure 7.10 gives a worked example:

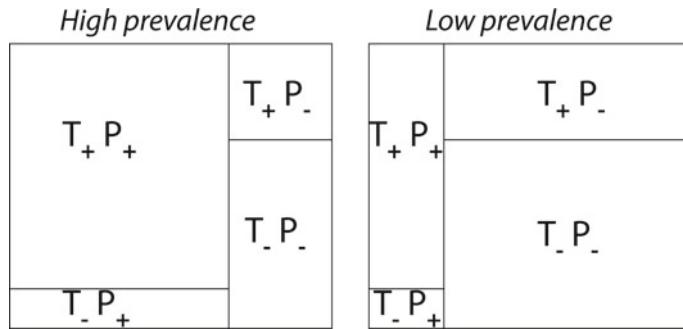


Fig. 7.9 Effect of prevalence on PPV and NPV. “T” stands for “test”, and “P” for “patient”. (For comparison with below: T+P+ = TP, T-P- = TN, T+P- = FP, and T-P+ = FN)

		Condition		Positive predictive value=
		Condition Positive		
Test Outcome	Test Outcome Positive	True Positive (TP) = 25	False Positive (FP) = 175	= TP / (TP+FP) = 25 / (25+175) = 12.5%
	Test Outcome Negative	False Negative (FN) = 10	True Negative (TN) = 2000	Negative predictive value=
	Sensitivity = = TP / (TP+FN) = 25 / (25+10) = 71%	Specificity = = TN / (FP+TN) = 2000 / (175+2000) = 92%	= TN / (FN+TN) = 2000 / (10+2000) = 99.5%	

Fig. 7.10 Worked example

7.3.1 Related Calculations

In the popular area of machine learning, simulation results are commonly characterized by their *accuracy*, *recall*, *precision* and *F1-score*:

- accuracy = $\frac{TP+TN}{P+N} = \frac{TP+TN}{TP+TN+FP+FN}$
- recall = sensitivity = power
- precision = PPV
- F1-score = $2 \cdot \frac{PPV * sensitivity}{PPV + sensitivity}$, i.e., the harmonic mean of precision and sensitivity

In evidence-based medicine, *likelihood ratios* are used for assessing the value of performing a diagnostic test. They use the sensitivity and specificity of the test to determine whether a test result changes the probability that a condition (such as a disease state) exists. Two versions of the likelihood ratio exist, one for positive and one for negative test results. They are known as the *positive likelihood ratio* (*LR+*) and *negative likelihood ratio* (*LR-*), respectively. For example, and *LR+* tells you how much to increase the probability of having a disease, given a positive test result. The *LR+* ratio is: Probability a person with the condition tests positive (a

true positive) vs. probability a person without the condition tests positive (a false positive) = (true positive rate) / (false positive rate).

For the example in Fig. 7.10, we get the following numbers:

- false positive rate (α) = type I error = $1 - specificity = \frac{FP}{FP+TN} = \frac{175}{175+2000} = 8\%$
- false negative rate (β) = type II error = $1 - sensitivity = \frac{FN}{TP+FN} = \frac{10}{25+10} = 29\%$
- power = sensitivity = $1 - \beta$
- *positive likelihood ratio* = $\frac{sensitivity}{1-specificity} = \frac{71\%}{1-92\%} = 8.9$
- *negative likelihood ratio* = $\frac{1-sensitivity}{specificity} = \frac{1-71\%}{92\%} = 0.32$

In subjects with the condition, the test is 8.9 times more likely to produce a positive result than in healthy subjects. But the prevalence of the condition is so low ($35/[35+2175] = 1.6\%$), that we nevertheless have more false positives than true positives. So a positive test outcome in this example is in itself poor at confirming cancer (PPV = 12.5%) and further investigations must be undertaken; it does, however, correctly identify 71% of all cancers (the sensitivity). However as a screening test, a negative result is very good at reassuring that a patient does not have cancer (NPV = 99.5%), and this initial screen correctly identifies 92% of those who do not have cancer (the specificity).

7.3.2 Example: Mammogram

As another specific example let us take a look at a frequently occurring test: a mammogram. Mammograms have a *sensitivity* of approximately 80% and a *specificity* of approximately 90% (see Fig. 7.11). Now imagine you go to a doctor, get a mammogram, and the doctor tells you, “I am sorry to inform you that your mammogram came out positive.”. What are you supposed to think?

Not knowing the PPV or the incidence of breast cancer, you *cannot* interpret the result! But with additional information about the incidence of breast cancer (which is for example about 1% for 50 year old women) we can work out the PPV, which indicates to you how many of all positive tests are *true positive*: as shown in Fig. 7.11 the PPV is only $\frac{80}{1030} \approx 7.8\%$. So before you get nervous—get a second test!

7.4 Receiver-Operating-Characteristic (ROC) Curve

Closely related to sensitivity and specificity is the *Receiver-Operating-Characteristic (ROC)* curve. This technique provides a clear procedure on how to determine the optimal threshold for parameters that divide two populations, e.g., healthy subjects and patients. The ROC curve is a graph that shows the relationship between the true positive rate (on the vertical axis) and the false positive rate (on the horizontal axis). The technique comes from the field of engineering, where it was developed to find the predictor which best discriminates between two given distributions: ROC

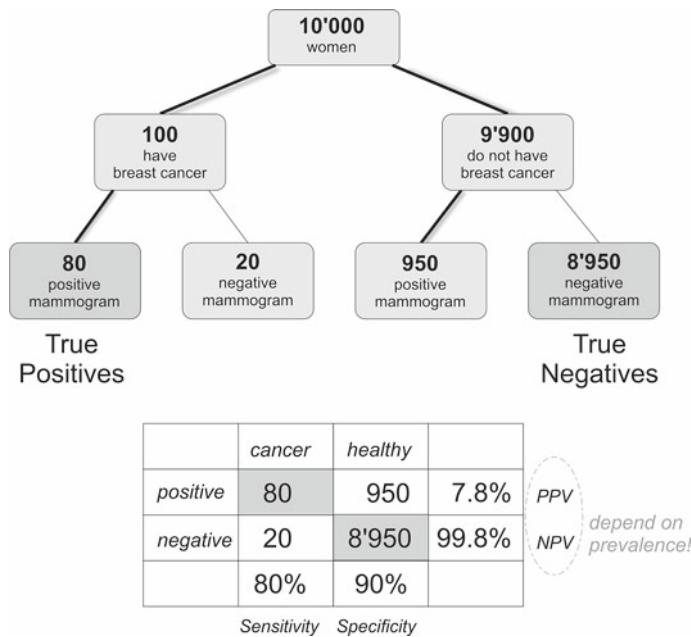


Fig. 7.11 How likely are you to have breast cancer, when your mammogram came out positive?

curves were first used during WWII to analyze radar effectiveness. In the early days of radar, it was sometimes hard to tell a bird from a plane. The British pioneered using ROC curves to optimize the way that they relied on radar for discriminating between incoming German planes and birds.

Take the case that we have two different distributions, for example, one from the radar signal of birds and one from the radar signal of German planes, and we have to determine a cut-off value for an indicator in order to assign a test result to distribution one (“bird”) or to distribution two (“German plane”). The only parameter that we can change is the cut-off value, and the question arises: is there an optimal choice for this cut-off value?

The answer is *yes*: it is the point on the ROC curve with the largest distance to the diagonal (arrow in Fig. 7.12).⁵

⁵ Strictly speaking this only holds if type I errors and type II errors are equally important. Otherwise, the weight of each type of error also has to be considered.

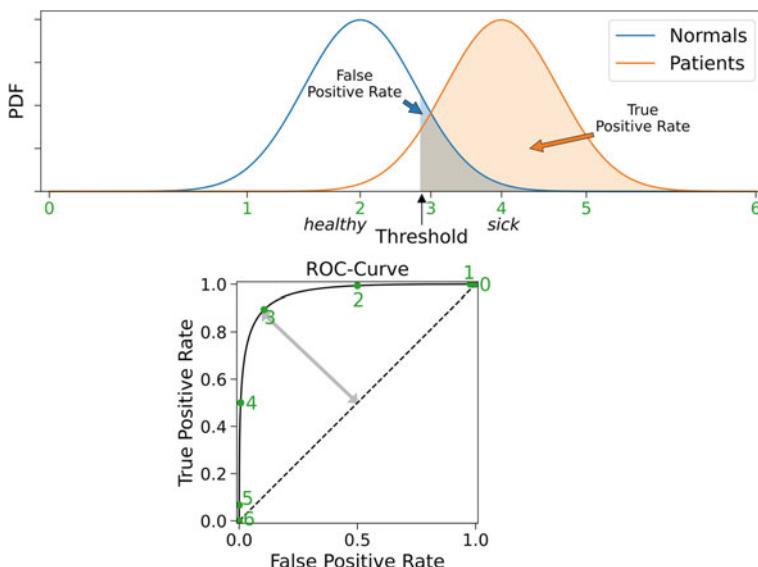


Fig. 7.12 **Top:** Probability density functions for two distributions. **Bottom:** corresponding ROC curve, with the largest distance between the diagonal and the curve (gray arrow) indicating the optimal distinction threshold. ROC-points corresponding to integer threshold values are indicated with green dots and numbers

7.5 Exercises

1. Sensitivity and Specificity

A new test is compared to the “gold standard”. The results are

True Positive: 10

False Positive: 2

False Negative: 3

True Negative: 8

- Calculate sensitivity, specificity, positive predictive value, and negative predictive value.
- Which of these are affected by the prevalence of the disease?

2. ROC Curves

In an effort to automate the detection of back problems, an exercise called “Waiter’s bow” was executed by different subjects. The execution of this movement was recorded with a movement sensor, and an experienced physiologist rated each movement as *healthy*, *pathological*, or *uncertain*. The results were stored in the file Trunk_flexion.xlsx.

- Read in the data
- Eliminate all the data with comments, and data where the physiotherapist was “uncertain”.
- Calculate and display sensitivity, specificity, PPV, and NPV for the data, if calculated values below a threshold=9 indicate a healthy subject.
- For the same data, plot the ROC curve and calculate the best discrimination threshold.

3. Precision and Recall

You are given a classification program for hand-written numbers, which for each classification also provides a “confidence-level”. Now you have to decide at which confidence-level you want to use the program. For selected confidence-levels between 0 and 12, the following numbers indicating [8, 7, 3, 9, 5, 2, 5, 5, 6, 5, 5] are classified as “5”:



So for example, for a confidence-level of “10”, the last two numbers are correctly identified as “5”, but the 5-s with a lower confidence- level are missed.

- Plot the Recall/Sensitivity/Power and the Precision/PPV as a function of the confidence-level.
- What are the Recall- and Precision-values for a confidence-level of “7”?

Chapter 8

Tests of Means of Numerical Data



This chapter covers hypothesis tests for the mean values of groups, and shows how to implement each of these tests in Python:

- comparison of one group with a fixed value,
- comparison of two groups with respect to each other,
- comparison of three or more groups with each other.

In each case, we distinguish between two cases. If the data are approximately normally distributed, so-called *parametric tests* can be used. These tests are more sensitive than *non-parametric tests*, but require that certain assumptions are fulfilled. If the data are not normally distributed, or if they are only available in ranked form, the corresponding non-parametric tests should be used.

8.1 Distribution of a Sample Mean

8.1.1 One Sample T-Test for a Mean Value

The t-distribution typically describes the distribution of the mean value of samples taken from a normal distribution. So to check the mean value of normally distributed data against a reference value we typically use the *one sample t-test*, which is based on the t-distribution.

If we knew the mean and the standard deviation of a normally distributed population, we could calculate the corresponding standard error, and use values from the normal distribution to determine how likely it is to find a certain value. However, in practice we have to estimate the mean and standard deviation from the sample; and the t-distribution, which characterizes the distribution of sample means for normally distributed data, deviates slightly from the normal distribution.

a) Example

Since it is very important to understand the basic principles of how to calculate the t-statistic and the corresponding p-value for this test, let me illustrate the underlying statistics by going through the analysis of a specific example, step-by-step. As an example we take 100 normally distributed data, with a mean of 7 and with a standard deviation of 3. What is the chance of finding a mean value at a distance of 0.5 or more from the mean? *Answer: The probability from the t-test in the example is 0.057, and from the normal distribution 0.054.*

- We have a population, with a mean value of 7 and a standard deviation of 3.
- From that population an observer takes 100 random samples. The sample mean of the example shown in Fig. 8.1 is 7.10, close to but different from the real mean. The sample standard deviation is 3.12, and the standard error of the mean 0.312. This gives the observer an idea about the variability of the population.
- The observer knows that the distribution of the sample mean follows a t-distribution, and that the standard error of the mean (SEM) characterizes the width of that distribution.

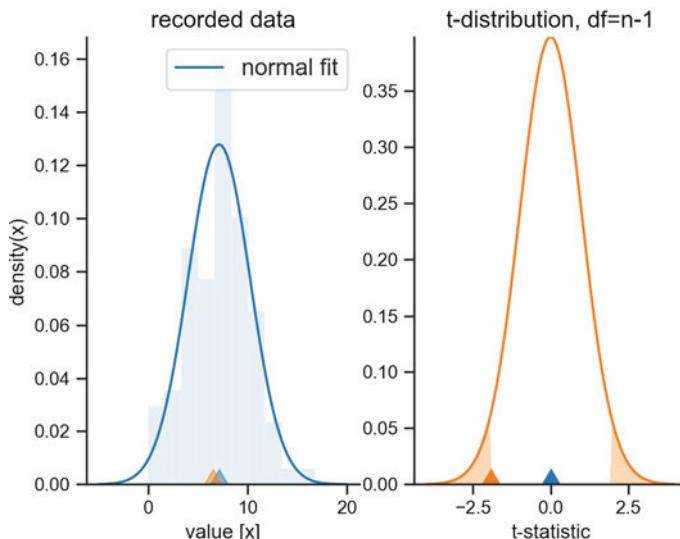


Fig. 8.1 **Left:** Frequency histogram of the sample data, together with a normal fit (blue line). The sample mean, which is very close to the population mean, is indicated with a blue triangle; the value to be checked is indicated with an orange triangle. **Right:** sampling distribution of the mean value (t-distribution, for $n-1$ degrees of freedom). At the bottom the normalized value of the sample mean (blue triangle) and the normalized value to be checked (orange triangle). The sum of the orange shaded areas, indicating values as extreme or more extreme than the orange arrow, corresponds to the p-value

- How likely it is that the real mean has a value of x_0 (e.g., 6.5, indicated by the orange triangle in Fig. 8.1, left)? To find this out, the value has to be transformed by subtracting the sample mean and dividing by the standard error (Fig. 8.1, right, and Eq. 6.13). This provides the *t-statistic* for this test (-1.93).
- The corresponding p-value, which tells us how likely it is that the real mean has a value of 6.5 or more extreme relative to the sample mean, is given by the orange shaded area under the curve-wings: $2 \cdot CDF(t\text{-statistic}) = 0.057$, which means that the difference to 6.5 is just not significant. The factor “2” comes from the fact that we have to check in both tails, and this test is therefore referred to as a *two-tailed t-test*.
- The probability to find a value of 6.5 or less is half as much ($p = 0.0285$). Since in this case we only look in one tail of the distribution, this is called a *one-tailed t-test*.

In Python, test statistic and p-value for the one sample t-test can be calculated with

```
t, pVal = stats.ttest_1samp(data, checkValue)
```

or with

```
result = pg.ttest(data, checkValue)
```

b) Pingouin

The Python statistics package, *pingouin*—commonly abbreviated as `pg`—aims to make life simpler for the user by (a) offering a simplified interface, and by (b) presenting the results in a richer, more useful format. I will use the one-sample t-test here to explain some of the common output parameters of that package for parametric tests.

```
# Get the required packages
import numpy as np
from scipy import stats
import pingouin as pg

# Generate the data
np.random.seed(12345)
data = stats.norm(7, 3).rvs(100)

# T-test with scipy.stats
ref_val = 6.5
stats.ttest_1samp(data, ref_val)
# >>> Ttest_1sampResult(statistic
#                      =1.9252254884316808, pvalue=0.05707107880872914)

# T-test with pingouin
result = pg.ttest(data, ref_val)
print(result.round(3))

#          T   dof      tail    p-val       CI95%  cohen-d    BF10   power
# T-test  1.925   99  two-sided  0.057  [6.48, 7.72]  0.193  0.651  0.479
```

Most *pingouin* programs return the individual parameters such that those can be accessed easily. For the example above, the power of the test can be obtained by `result['power']` or by `result.power`. The return parameters of `pg.ttest` can be interpreted as follows:

T Difference between sample mean and reference value, normalized by the SEM

dof `len(data) - 1`

tail By default a two-sided comparison is performed, i.e., it is checked if the data are *different* from the reference value. Comparisons using `alternative = 'greater'` or `alternative = 'less'` employ the corresponding one-sided comparisons.

p-val The area under the T-distribution for T-values more extreme than the observed value

CI95% 95% confidence interval around the sample mean. Since in the example above this does include the reference value (6.5), the result is *not significant*.

cohen-d “Cohen’s d”, also called *effect size*. Difference between the sample mean and the reference value, normalized by the sample standard deviation

BF10 (“Bayes Factor one-zero”) A Bayesian alternative to classical hypothesis testing. While Bayesian analysis corresponds more intuitively to our common-sense thinking than hypothesis testing, it requires significantly better mathematical skills, and therefore won’t be dealt with in much detail in this book. For a short introduction to Bayesian analysis in Python see Chap. 14.

power The *power of the test* is the area under the curve of the probability distribution of the sample mean, outside of the critical values (shaded areas in Fig. 7.7). Here this distribution is a *non-central T-distribution* relative to the reference value.

Since *pingouin* is rather new I will in the following present the established commands first, but try to indicate where *pingouin* offers a helpful alternative.



Code: `ISP_oneGroup.py`¹ Sample analysis for one group of continuous data.

8.1.2 Wilcoxon Signed Rank Sum Test

If the data are not normally distributed, the one-sample t-test should not be used (although this test is fairly robust against deviations from normality, see Fig. 6.17). Instead, we must use a non-parametric test on the mean value. We can do this by performing a *Wilcoxon signed rank sum test*. Note that in contrast to the one-sample t-test, this test checks for a difference from null:

¹ [<ISP2e>/08_TestsMeanValues/oneGroup/ISP_oneGroup.py](#).

Table 8.1 Daily energy intake of 11 healthy women with rank order of differences (ignoring their signs) from the recommended intake of 7725 kJ

Subject	Daily energy intake (kJ)	Difference from 7725 kJ	Ranks of differences
1	5260	2465	11
2	5470	2255	10
3	5640	2085	9
4	6180	1545	8
5	6390	1335	7
6	6515	1210	6
7	6805	920	4
8	7515	210	1.5
9	7515	210	1.5
10	8230	-505	3
11	8770	-1045	5

This method has three steps²:

1. Calculate the difference between each observation and the value of interest.
2. Ignoring the signs of the differences, rank them in order of magnitude.
3. Calculate the sum of the ranks of all the negative (or positive) ranks, corresponding to the observations below (or above) the chosen hypothetical value.

In Table 8.1, you see an example where the significance to a deviation from the value of 7725 is tested. The rank sum of the negative values gives $3 + 5 = 8$, and can be looked up in the corresponding tables to be significant. In practice, your computer program will nowadays do this for you. This example also shows another feature of rank evaluations: tied values (here 7515) get accorded their mean rank (here 1.5).

a) Pingouin

For non-parametric tests “power” and “BF10”-value are not defined. For such tests *pingouin* instead returns the following values:

CLES (“common language effect size”) the probability that a randomly selected score from the one population will be greater than a randomly sampled score from the other population (McGraw and Wong, 2022)

RBC (“rank-biserial correlation”) Another way to characterize the effect size for continuous variables that are not normally distributed. Related to the CLES.

```
result = pg.wilcoxon(data, checkValue*np.ones_like(data))
```

² The following description and example have been taken from Altman (1999), Table 9.2.

8.2 Comparison of Two Groups

8.2.1 Paired T-Test

In the comparison of two groups with each other, two cases have to be distinguished. In the first case, two values recorded from the same subject at different times are compared to each other. For example, the size of students when they enter primary school and after their first year, to check if they have grown (Fig. 8.2). Since we are only interested in the difference in each subject between the first and the second measurement, this test is called *paired t-test*, and is essentially equivalent to a one-sample t-test for the mean difference.

Therefore, the two tests `stats.ttest_1samp` and `stats.ttest_rel` provide the same result (apart from minute numerical differences):

```
In [1]: import numpy as np
In [2]: from scipy import stats

In [3]: np.random.seed(1234)
In [4]: diffs = np.random.randn(10)+0.1
In [5]: data_1 = np.random.randn(10)*5 # dummy data
In [6]: data_2 = data_1 + diffs # same group-difference
                           # as 'diffs'
In [7]: stats.ttest_1samp(diffs, 0)
Out[7]: (-0.1246, 0.90359)

In [8]: stats.ttest_rel(data_2, data_1)
Out[8]: (-0.1246, 0.90359)
```

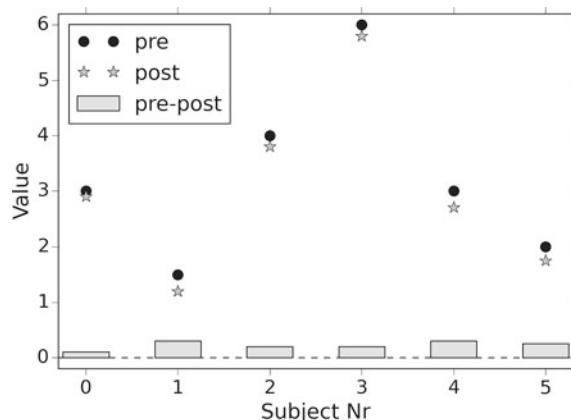


Fig. 8.2 Paired t-tests can detect differences that would be insignificant otherwise. For this example all the inner-subject differences are positive, and the paired t-test yields a p-value of $p < 0.001$, while the unpaired t-test leads to $p = 0.81$

8.2.2 T-Test Between Independent Groups

An *unpaired t-test*, or *t-test for two independent groups*, compares two groups. An example would be the comparison of the effect of two medications given to two different groups of patients.

The basic idea is the same as for the one-sample t-test. But instead of the variance of the mean, we now need the variance of the difference between the means of the two groups. Since *the variance of a sum (or difference) of independent random variables equals the sum of the variances*, we have

$$\begin{aligned} sd(\bar{x}_1 \pm \bar{x}_2) &= \sqrt{\text{var}(\bar{x}_1) + \text{var}(\bar{x}_2)} \\ &= \sqrt{\{sd(\bar{x}_1)\}^2 + \{sd(\bar{x}_2)\}^2} \\ &= \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}} \end{aligned} \tag{8.1}$$

where \bar{x}_i is the mean of the i th sample, and $sd(\bar{x})$ indicates here the *standard error of the mean*.

```
t_statistic, pVal = stats.ttest_ind(data_1, data_2)
```

8.2.3 T-Tests with Pingouin

Most t-tests from `scipy.stats` can be replaced by the command `pg.ttest`, making use of its various input parameters:

```
# one-sample T-test
stats.ttest_1samp(data, value)
pg.ttest(data, value)

# unpaired T-test
stats.ttest_ind(data_1, data_2)
pg.ttest(data_1, data_2)

# paired T-test
stats.ttest_rel(data_1, data_2)
pg.ttest(data_1, data_2, paired=True)
```

8.2.4 Non-parametric Comparison of Two Groups: Mann-Whitney Test

If the measurement values from two groups are not normally distributed, we have to resort to a non-parametric test. The most common non-parametric test for the comparison of two independent groups is the *Mann-Whitney(-Wilcoxon) test*. Watch out, because this test is sometimes also referred to as *Wilcoxon rank-sum test*. This is different from the *Wilcoxon signed rank sum test*! The test statistic for this test is commonly indicated with u :

```
# with scipy.stats ....
u_statistic, pVal = stats.mannwhitneyu(data_1, data_2)

# ... or with pingouin
results = pg.mwu(data_1, data_2)
```



Code: ISP_twoGroups.py³ Comparison of two groups, paired and unpaired.

With the advent of cheap computing power, statistical modeling has been a booming field. This has also affected classical statistical analysis, as most problems can be viewed from two perspectives: one can either make a statistical hypothesis, and verify or falsify that hypothesis; or can make a statistical model, and analyze the significance of the model parameters.

Let me use a classical t-test as an example.

Listing 8.1: hypothesisTests_vs_modeling.py

```
""" Equivalence between T-Test and Statistical Model """

# Import standard packages
import numpy as np
import scipy.stats as stats
import pandas as pd
import statsmodels.formula.api as sm

# Generate normally distributed data around 'reference' + 0.2
np.random.seed(123)
reference = 5
diffs = 0.2 + np.random.randn(100)
values = reference + diffs
diffs_df = pd.DataFrame({'diffs': diffs}) # diffs in a DataFrame

# t-test
(t, pVal) = stats.ttest_1samp(values, reference) # >> p=0.048
print('The probability that the sample mean is different' +
      f' from {reference} is {pVal:.3f}\n')
```

³ <ISP2e>/08_TestsMeanValues/twoGroups/ISP_twoGroups.py.

```
# Equivalent linear model
result = sm.ols(formula='diffs ~ 1', data=diffs_df).fit()
print(result.summary())
```

The command `random.seed(123)` in the code above initializes the random number generator with the number `123`, which ensures that two consecutive runs of this code produce the same result. And the output from the t-test ($p = 0.048$) indicates that the sample mean is just significantly different from the `reference`.

Expressed as a statistical model, we assume that the difference between sample data and the `reference` is simply a constant value. (The null hypothesis would be that this value is equal to zero.) This model has one parameter, a constant value, and this is expressed in the formula '`diffs ~ 1`'. We can find this parameter, as well as its confidence interval and a lot of additional information, with the Python code listed above.

The important line is the last but one, which produces the `result`. Thereby the `statsmodels` function `sm.ols` ("ordinary least square") tests the model which describes the `diffs` with only an offset, which in the language of modeling is also called "intercept". The results below show that the probability that this intercept is zero is 0.048, which is equivalent to the outcome of the t-test.

The probability that the sample mean is different than 5 is 0.048.

```
OLS Regression Results
=====
Dep. Variable:          diffs    R-squared:      -0.000
Model:                 OLS     Adj. R-squared:   -0.000
Method:                Least Squares   F-statistic:    nan
Date:        Tue, 24 Aug 2021   Prob (F-statistic):   nan
Time:        16:07:28         Log-Likelihood:  -153.96
No. Observations:      100      AIC:            309.9
Df Residuals:          99      BIC:            312.5
Df Model:                  0
Covariance Type:       nonrobust
=====
            coef    std err        t     P>|t|      [0.025    0.975]
-----
Intercept    0.2271     0.113     2.003     0.048     0.002     0.452
=====
Omnibus:             2.725   Durbin-Watson:    1.975
Prob(Omnibus):       0.256   Jarque-Bera (JB):  1.734
Skew:                 0.033   Prob(JB):       0.420
Kurtosis:              2.358   Cond. No.:      1.00
=====
```

The output from OLS-models is explained in more detail in Chap. 12. The important point here is that the t- and p-value for the intercept obtained with the statistical model are the same as with the classical t-test.

8.3 Comparison of Multiple Groups

8.3.1 Analysis of Variance (ANOVA)

The idea behind the Analysis of Variance (ANOVA) is to divide the variance into the variance *between* groups, and that *within* groups, and see if those distributions match the null hypothesis that all groups come from the same distribution. The variables that distinguish the different groups are often called *factors* or *treatments* (Fig. 8.3).

(By comparison, t-tests look at the mean values of two groups, and check if those are consistent with the assumption that the two groups come from the same distribution.)

For example, if we compare three groups (one group with *no treatment*, another with *treatment A*, and a third with *treatment B*), then we perform a *one factor ANOVA*, sometimes also called *one-way ANOVA*, with *treatment* the one analysis factor. If we do the same test with men and with women, then we have a *two-factor* or *two-way ANOVA*, with *gender* and *treatment* as the two treatment factors. Note that with ANOVAs, it is quite important to have exactly the same number of samples in each analysis group! (This is called a *balanced ANOVA*: a balanced design has an equal number of observations for all possible combinations of factor levels.)

Because the null hypothesis is that there is *no difference between the groups*, the test is based on a comparison of the observed variation between the groups (i.e., between their means) with that expected from the observed variability between subjects. The comparison takes the general form of an F-test to compare variances,

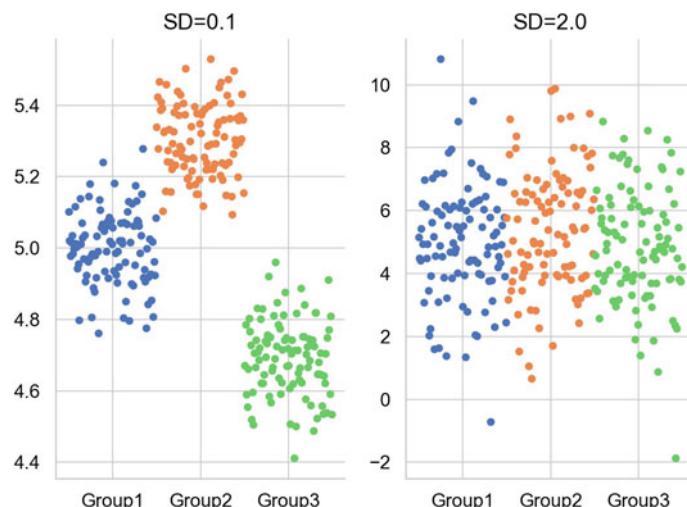


Fig. 8.3 In both cases, the difference between the two groups is the same. But left, the difference within the groups is smaller than the differences between the groups, whereas right, the difference within the groups is larger than the difference between

but for two groups the t-test leads to exactly the same result, as demonstrated in the code-sample [ISP_anovaOneway.py](#).

The one-way ANOVA assumes all the samples are drawn from normally distributed populations, that these populations have equal variance, and that the samples are independent from each other. The assumption of equal variance can be checked with the *Levene test*.

ANOVA uses traditional terminology. DF indicates the degrees of freedom (DF) (see also Sect. 5.4), the summation is called the Sum-of-Squares (SS), the ratio between the two is called the Mean Square (MS), and the squared terms are deviations from the sample mean. In general, the sample variance is defined by

$$s^2 = \frac{1}{DF} \sum (y_i - \bar{y})^2 = \frac{SS}{DF}. \quad (8.2)$$

The fundamental technique is a partitioning of the total sum of squares SS into components related to the effects used in the model (Fig. 8.4). Thereby ANOVA estimates three sample variances: a *total variance* based on all the observation deviations from the grand mean (calculated from SS_{Total}), a *treatment variance* (from $SS_{Treatments}$), and an *error variance* based on all the observation deviations from their appropriate treatment means (from SS_{Error}). The treatment variance is based on the deviations of treatment means from the grand mean, the result being multiplied by the number of observations in each treatment to account for the difference between the variance of observations and the variance of means. The three sums-of-squares are related by

$$SS_{Total} = SS_{Error} + SS_{Treatment} \quad (8.3)$$

where SS_{Total} is sum-squared deviation from the overall mean, the SS_{Error} the sum-squared deviation from the mean within a group, and the $SS_{Treatment}$ the sum-squared deviation between each group and the overall mean (Fig. 8.4). If the null hypothesis is true, all three variance estimates (Eq. 8.2) are equal (within sampling error).

The number of degrees of freedom DF can be partitioned in a similar way: one of these components (that for error) specifies a chi-squared distribution which describes the associated sum of squares, while the same is true for *treatments* if there is no treatment effect.

$$DF_{Total} = DF_{Error} + DF_{Treatments}. \quad (8.4)$$

a) Example: One-way ANOVA

As an example, take the red cell folate levels ($\mu\text{g/l}$) in three groups of cardiac bypass patients given different levels of nitrous oxide ventilation (Amess et al. 1978), described in the Python code example below. In total 22 patients were included in the analysis.

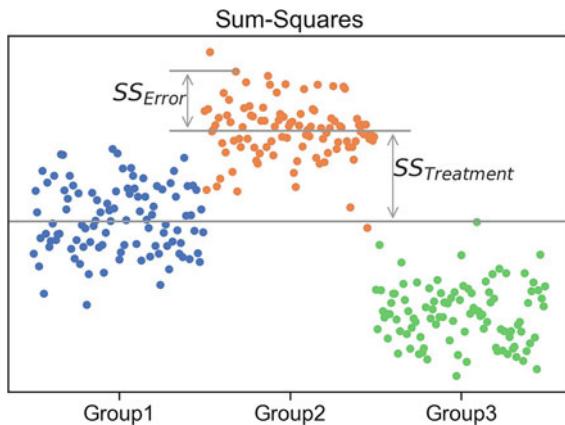


Fig. 8.4 The long blue line indicates the grand mean over all data. The SS_{Error} describes the variability *within* the groups, and the $SS_{Treatment}$ (summed over all respective points!) the variability *between* groups

The null hypothesis of ANOVAs is that all groups come from the same population. A test whether to keep or reject this null hypothesis can be done with

```
from scipy import stats
F_statistic, pVal = stats.f_oneway(group_1, group_2, group_3)
```

where the data of group i are in a vector `group_i`. (The whole program can be found in [ISP_anovaOneway.py](#).)

A more detailed output of the ANOVA is provided by the implementation in `pingouin` or `statsmodels`:

```
import pandas as pd
from statsmodels.formula.api import ols
from statsmodels.stats.anova import anova_lm
import pingouin as pg

df = pd.DataFrame(data, columns)

# Pingouin
pg_results = pg.anova(data=df,
                      dv='value', between='treatment')

# Statsmodels
model = ols('value ~ C(treatment)', df).fit()
sm_results = anova_lm(model)

print('pingouin -----')
print(pg_results.round(4))

print('\nstatsmodels -----')
print(sm_results.round(4))
```

where the numerical values are in the first column of the array `data`, and the (categorical) group variable `treatment` in the second column. This produces the following output:

```
pinguin -----
      Source   ddof1   ddof2          F    p-unc      np2
0  treatment       2       19   3.7113   0.0436   0.2809

statsmodels -----
              df      sum_sq      mean_sq          F  PR(>F)
C(treatment)  2.0  15515.7664  7757.8832  3.7113  0.0436
Residual      19.0  39716.0972  2090.3209      NaN      NaN
```

The `statsmodel` output can be interpreted as follows:

- First the “Sums of squares (SS)” are calculated. Here the SS between treatments is 15515.88, and the SS of the residuals is 39716.09. The total SS is the sum of these two values.
- The mean squares (MS) is the SS divided by the corresponding degrees of freedom (DF).
- The *F-test* or *variance ratio test* is used for comparing the factors of the total deviation. The F-value is the larger mean squares value divided by the smaller value. (If we only have two groups, the F-value is the square of the corresponding t-value. See [ISP_anovaOneway.py](#).

$$F = \frac{\text{variance_between_treatments}}{\text{variance_within_treatments}}$$

$$F = \frac{MS_{\text{Treatments}}}{MS_{\text{Error}}} = \frac{SS_{\text{Treatments}}/(n_{\text{groups}} - 1)}{SS_{\text{Error}}/(n_{\text{total}} - n_{\text{groups}})}. \quad (8.5)$$

- Under the null hypothesis that two normally distributed populations have equal variances we expect the ratio of the two sample variances to have an *F Distribution* (see Sect. 6.5). From the F-value, we can look up the corresponding p-value.



Code: `ISP_anovaOneway.py`⁴ Different aspects of one-way ANOVAs: how to check the assumptions (normality, and equality of variances), different ways to calculate a one-way ANOVA, and a demonstration that for the comparison between two groups, a one-way ANOVA is equivalent to a t-test.

⁴ [ISP2e>/08_TestsMeanValues/anovaOneway/ISP_anovaOneway.py](#).

8.3.2 Multiple Comparisons

The null hypothesis in a one-way ANOVA is that the means of all the samples are the same. So if a one-way ANOVA yields a significant result, we only know that they are *not* the same.

However, often we are not just interested in the joint hypothesis if all samples are the same, but we would also like to know for which pairs of samples the hypothesis of equal values is rejected. In this case we conduct several tests at the same time, one test for each pair of samples. (Typically, this is done with t-tests.)

These tests are sometimes referred to as *post-hoc analysis*. In the design and analysis of experiments, a post-hoc analysis (from Latin post-hoc, “after this”) consists of looking at the data—after the experiment has concluded—for patterns that were not specified beforehand. Here this is the case, because the null hypothesis of the ANOVA is that there is no difference between the groups.

This results, as a consequence, in a *multiple testing problem*: since we perform multiple comparison tests, we should compensate for the risk of getting a significant result, even if our null hypothesis is true. This can be done by correcting the p-values to account for this. We have a number of options to do so:

- Tukey HSD test
- Bonferroni correction
- Holms correction
- ... and others ...

a) Tukey's Test

Tukey's test, sometimes also referred to as the Tukey *Honest Significant Difference test (HSD)* method, controls for the Type I error rate across multiple comparisons and is generally considered an acceptable technique. It is based on a statistic that we have not come across yet, the *studentized range*, which is commonly represented by the variable q . The *studentized range* computed from a list of numbers x_1, \dots, x_n is given by

$$q_n = \frac{\max\{x_1, \dots, x_n\} - \min\{x_1, \dots, x_n\}}{s} \quad (8.6)$$

where s is the sample standard deviation. In the Tukey HSD method, the sample x_1, \dots, x_n is a sample of means and q is the basic test statistic. It can be used as post-hoc analysis to test between which two groups means there is a significant difference (pairwise comparisons) after rejecting the null hypothesis that all groups are from the same population (i.e., all means are equal) (Fig. 8.5).

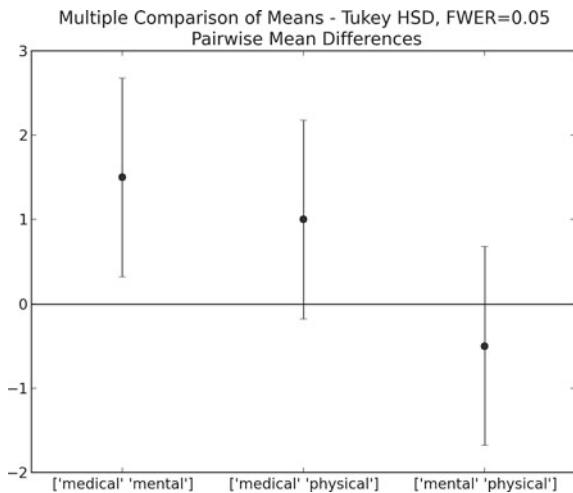


Fig. 8.5 Comparing the means of multiple groups—here three different treatment options



Code: `ISP_multipleTesting.py`.⁵ This script provides an example where three treatments are compared.

b) Bonferroni Correction

Tukey's studentized range test (HSD) is a test specific to the comparison of all pairs of k independent samples. Instead we can run t-tests on all pairs, calculate the p-values and apply one of the p-value corrections for multiple testing problems. The simplest—and at the same time quite conservative—approach is to divide the required p-value by the number of tests that we do (*Bonferroni correction*). For example, if you perform 4 comparisons, you check for significance not at $p = 0.05$, but at $p = 0.05/4 = 0.0125$.

While multiple testing is not included in Python, you can get a number of multiple-testing corrections done with the *statsmodels* package:

```
In [7]: from statsmodels.stats.multitest import multipletests

In [8]: multipletests([.05, 0.3, 0.01], method='bonferroni')
Out[8]:
(array([False, False,  True]),
 array([ 0.15,   0.9 ,   0.03]),
 0.0170,
 0.0167)
```

⁵ [ISP2e>/08_TestsMeanValues/multipleTesting/ISP_multipleTesting.py](#).

c) Holm Correction

The Holm adjustment, sometimes also referred to as *Holm-Bonferroni method*, sequentially compares the lowest p-value with a Type I error rate that is reduced for each consecutive test. For example, if you have three groups (and thus three comparisons), this means that the first p-value is tested at the 0.05/3 level (0.017), the second at the 0.05/2 level (0.025), and third at the 0.05/1 level (0.05). As stated by Holm (1979) “Except in trivial non-interesting cases the sequentially rejective Bonferroni test [i.e. the Holm-Bonferroni method] has strictly larger probability of rejecting false hypotheses, and thus it ought to replace the classical Bonferroni test at all instants where the latter usually is applied.”

8.3.3 Kruskal–Wallis Test

When we compare *two* groups to each other, we use the *t-test* when the data are normally distributed, and the non-parametric *Mann-Whitney test* otherwise.

For *three or more* groups, the analysis starts with the check of normality and the *Levene test*, which checks if the variables are comparable. If that is the case, one can proceed with the *ANOVA-test*; if it is not the case, then the *Kruskal–Wallis test* has to be used. For the post hoc pairwise multiple comparisons after the Kruskal-Wallis test is passed, I would recommend the use of the package [scikit-posthocs](#).



Code: `ISP_kruskalWallis.py`⁶ Example of a Kruskal-Wallis test (for not normally distributed data).

8.3.4 Two-Way ANOVA

Compared to one-way ANOVAs, the analysis with two-way ANOVAs has a new element. We can look not only if each of the factors is significant; we can also check if the *interaction* of the factors has a significant influence on the distribution of the data. Let us take for example measurement of fetal head circumference, by four observers in three fetuses, from a study investigating the reproducibility of ultrasonic fetal head circumference data.

The most elegant way of implementing a two-way ANOVAs for these data is with *statsmodels*:

```
# Import standard packages
import numpy as np
import pandas as pd
import pingouin as pg

# additional packages
from statsmodels.formula.api import ols
from statsmodels.stats.anova import anova_lm
```

⁶ [<ISP2e>/08_TestsMeanValues/kruskalWallis/ISP_kruskalWallis.py](#).

```

# Get the data
inFile = 'altman_12_6.txt'
data = np.genfromtxt(inFile, delimiter=',')

# Bring them in DataFrame-format
df = pd.DataFrame(data, columns=['hs', 'fetus', 'observer'])

# Use ANOVA with interaction, either with statsmodels ...
formula = 'hs ~ C(fetus) + C(observer) + C(fetus):C(observer)'

lm = ols(formula, df).fit()
sm_results = anova_lm(lm)

# ... or with pingouin
pg_results = pg.anova(dv='hs', between=['fetus', 'observer'],
                      data=df)

print(sm_results.round(4))

```

This leads to the following result:

	df	sum_sq	mean_sq	F	PR(>F)
C(fetus)	2	324.00	162.00	2113.10	0.000
C(observer)	3	1.19	0.39	5.21	0.0065
C(fetus):C(observer)	6	0.56	0.09	1.22	0.3296
Residual	24	1.84	0.07	NaN	NaN

In words: While—as expected—different fetuses show highly significant differences in their head size ($p < 0.001$), also the choice of the observer has a significant effect ($p < 0.05$). However, no individual observer was significantly off with any individual fetus ($p > 0.05$).



Code: `ISP_anovaTwoway.py`⁷ Two-way analysis of variance (ANOVA).

8.3.5 Three-Way ANOVA

With more than two factors, it is recommendable to use *statistical modeling* for the data analysis (see Chap. 12). However, as always with the analysis of statistical data, one should first inspect the data visually. *Seaborn* makes this quite simple. Figure 8.6 shows for example the pulse-rate after (1/15/30) minutes of (*resting/walking/running*), in two groups who are eating different diets.

```

import matplotlib.pyplot as plt
import seaborn as sns

```

⁷ ISP2e/08_TestsMeanValues/anovaTwoway/ISP_anovaTwoway.py.

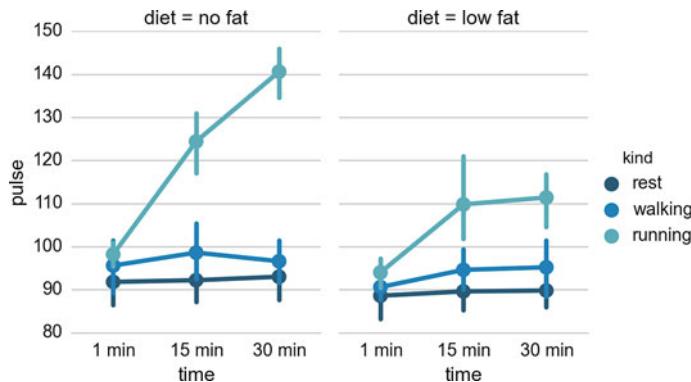


Fig. 8.6 Three-Way ANOVA

```

sns.set(style="whitegrid")

df = sns.load_dataset("exercise")

sns.factorplot("time", "pulse", hue="kind", col="diet",
               data=df, hue_order=["rest", "walking", "running"],
               palette="YlGnBu_d", aspect=.75).despine(left=True)
plt.show()

```

8.3.6 Friedman Test

The rank test for more than two groups of matched data is the *Friedman test*.

An example for the application of the Friedman test: Ten professional piano players are blindfolded, and are asked to judge the quality of three different pianos. Each player rates each piano on a scale of 1–10 (1 being the lowest possible grade, and 10 the highest possible grade). The null hypothesis is that all three pianos rate equally. To test the null hypothesis, the Friedman test is used on the ratings of the ten piano players.

It may be worth mentioning that in a blog Thom Baguley suggested that in cases where one-way repeated measures ANOVA is not appropriate, rank transformation followed by ANOVA will provide a more robust test with greater statistical power than the Friedman test (<http://www.r-bloggers.com/beware-the-friedman-test/>).

8.4 Summary: Selecting the Right Test for Comparing Groups

When we have univariate data and two groups, we can ask the question: “Are they different?” The answer is provided by hypothesis tests: by a t-test if the data are normally distributed, or by a Mann-Whitney test otherwise.

So what happens when we have more than two groups?

To answer the question “Are they different?” for more than two groups, we have to start out with checks for normality, and with the Levene test to check for the equality of the variances. If that is the case, we proceed with the Analysis of Variance (ANOVA)-test. If this condition is not fulfilled, the Kruskal-Wallis Test has to be used. If these tests indicate that the null hypothesis has to be rejected, then post-hoc tests have to be used to find out which groups are significantly different from each other.

What should we do if we have paired data?

If we have matched pairs for two groups, and the differences are not normally distributed, we can use the Wilcoxon-signed-rank-sum test. The rank test for more than two groups of matched data is the *Friedman test* (Table 8.2).

Hypothetical Examples

2 groups, nominal male/female, blond-eyed/dark-eyed. E.g., “Do females have blue eyes more frequently than males?”

2 groups, nominal, paired 2 labs, analysis of blood samples. E.g., “Does the blood analysis from Lab1 indicate more infections than the analysis from Lab2?”

1 group, ordinal Sequence of giant-planets. E.g., “In our solar system, are giant planets further out than average in the sequence of planets?”

Table 8.2 Typical tests for statistical problems, for nominal and ordinal data. Note that the tests for comparing one group to a fixed value are the same as comparing two groups with paired samples. Tests for nominal data will be discussed in detail in the next chapter

Groups compared	Independent samples	Paired samples
Nominal Data		
2	Fisher's exact test, or Chi-Square test (also for >2 groups)	McNemar's test
Ordinal Data		
1	Wilcoxon signed-rank-sum test	–
2	Mann-Whitney U test	Wilcoxon signed-rank sum test
3 or more	Kruskal-Wallis test	Friedman test
Continuous Data		
1	One-sample t-test, or Wilcoxon signed-rank sum test	–
2	Student's t-test, or Mann-Whitney test	Paired t-test, or Wilcoxon signed-rank sum test
3 or more	ANOVA or Kruskal-Wallis test	Repeated Measures ANOVA or Friedman test

- 2 groups, ordinal** Jamaican/American, ranking 100 m sprint. E.g., “Are Jamaican sprinters more successful than American sprinters?”
- 2 groups, ordinal, paired** sprinters, before/after diet. E.g., “Does a chocolate diet make sprinters more successful?”
- 3 or more groups, ordinal** single/married/divorces, ranking 100 m sprint. E.g. “Does the marital status have an effect on the success of sprinters?”
- 3 or more groups, ordinal, paired** US, Chinese, and Russian sprinters, before/after diet. E.g., “Does a rice diet make Chinese sprinters more successful?”
- 1 group, continuous** Average caloric intake. E.g. “Do our children eat more than they should?”
- 2 groups, continuous** male/female, IQ. E.g., “Are women more intelligent than men?”
- 2 groups, continuous, paired** male, looking at sport cars. E.g., “Does looking at sports cars raise the male heart-beat more than the female heartbeat?”
- 3 groups, continuous** Tyroleans, Viennese, Styrians; IQ. E.g., “Are Tyroleans smarter than people from other Austrian federal states?”
- 3 groups, continuous, paired** Tyroleans, Viennese, Styrians; looking at mountains. E.g., “Does looking at mountains raise the heartbeat of Tyroleans more than those of other people?”
- 2-factor ANOVA** small/large and male/female. E.g., “Do tall male persons have a higher income?”

8.5 Exercises

1. One or Two Groups

- **One sample t-test for the mean and Wilcoxon signed rank sum test**
The daily energy intake from 11 healthy women is [5260., 5470., 5640., 6180., 6390., 6515., 6805., 7515., 7515., 8230., 8770.] kJ.
Is this value significantly different from the recommended value of 7725?
- **t-test of independent samples**
In a clinic, 15 lazy patients weigh [76, 101, 66, 72, 88, 82, 79, 73, 76, 85, 75, 64, 76, 81, 86.] kg, and 15 sporty patients weigh [64, 65, 56, 62, 59, 76, 66, 82, 91, 57, 92, 80, 82, 67, 54] kg.
Are the lazy patients significantly heavier?
- **Normality test**
Are the two data sets normally distributed?
- **Mann–Whitney test**
Are the lazy patients still heavier, if you check with the Mann–Whitney test?

2. Multiple Groups

- Give a specific example of a two-factor ANOVA.

The following example is taken from the really good, but somewhat advanced book by AJ Dobson: “An Introduction to Generalized Linear Models”:

- **Get the data**

The file *Data/data_others/Table 6.6 Plant experiment.xls*, which can also be found on <https://github.com/thomas-haslwanger/statsintro-python-2e/tree/master/data>, contains data from an experiment with plants in three different growing conditions. Read the data into Python. Hint: use the module *xlrd*.

- **Perform an ANOVA**

Are the three groups different?

- **Multiple Comparisons**

Using the Tukey test, which of the pairs are different?

- **Kruskal–Wallis**

Would a non-parametric comparison lead to a different result?

Chapter 9

Tests on Categorical Data



In a data sample the number of data falling into a particular group is called the *frequency*, so the analysis of categorical data is the *analysis of frequencies*. When two or more groups are compared the data are often shown in the form of a *frequency table*, sometimes also called *contingency table*. For example, Table 9.1 gives the number of right/left-handed subjects, *contingent* on the subject being male or female.

If we have only one *factor* (i.e., a table with only one row), the analysis options are somewhat limited (Sect. 9.2.1). In contrast, a number of statistical tests exist for the analysis of frequency tables:

Chi-square test This is the most common type. It is a hypothesis test, which checks if the entries in the individual cells in a frequency table (e.g., Table 9.1) all come from the same distribution. In other words, it checks the null hypothesis H_0 that the results are independent of the row or column in which they appear. The alternative hypothesis H_a does not specify the type of association, so close attention to the data is required to interpret the information provided by the test correctly.

Fisher's Exact Test While the chi-square test is approximate, the *Fisher's Exact Test* is an exact test. It is computationally more expensive and intricate than the chi-square test, and was originally used only for small sample numbers. However, in general, it is now the more advisable test to use.

McNemar's Test This is a *matched pair test* for 2×2 tables. For example, if you want to see if two doctors obtain comparable results when checking (the same) patients if they are fit for work, you would use this test.

Cochran's Q Test Cochran's Q test is an extension to the McNemar's test for related samples that provides a method for testing for differences between three or more matched/paired sets of frequencies or proportions. For example, if you have exactly the same samples analyzed by three different laboratories, and you want to check if the results are statistically equivalent, you would use this test.

Table 9.1 Example of a frequency table

	Right handed	Left handed	Total
Males	43	9	52
Females	44	4	48
Total	87	13	100

9.1 Proportions and Confidence Intervals

If we have one sample group of data, we can check if the sample is representative of the standard population. To do so, we have to know the proportion p of the characteristic in the standard population. The occurrence of a characteristic in a group of n people is described by the binomial distribution, with $mean = p * n$. The standard error of samples with this characteristic is given by (see also Table 6.1)

$$se(p) = \sqrt{p(1 - p)/n} \quad (9.1)$$

and the corresponding 95% confidence interval is

$$ci = mean \pm se * t_{n,0.025}.$$

Thereby $t_{n,0.025}$ can be calculated as the inverse survival function (ISF) of the t-distribution, at the value 0.025. If the data lie outside this confidence interval, they are *not* representative of the population.

9.1.1 Explanation

The innocent looking Eq. 9.1 is more involved than it seems at first:

If we have n independent samples from a binomial distribution $B(k, p)$, the variance of their sample mean is

$$\text{var}\left(\frac{1}{n} \sum_{i=1}^n X_i\right) = \frac{1}{n^2} \sum_{i=1}^n \text{var}(X_i) = \frac{n \text{ var}(X_i)}{n^2} = \frac{\text{var}(X_i)}{n} = \frac{kpq}{n}$$

where $q = 1 - p$. This follows since

1. $\text{var}(cX) = c^2 \text{var}(X)$, for any random variable, X , and any constant c .
2. the variance of a sum of independent random variables equals the sum of the variances.

The standard error of the sample mean \bar{X} is the square root of the variance: $\sqrt{\frac{kpq}{n}}$. Therefore,

- When $k = n$, we get $se = \sqrt{pq}$.
- When $k = 1$, and the Binomial variables are just Bernoulli trials, the standard error is given by $se = \sqrt{\frac{pq}{n}}$.

9.1.2 Example

For example, let us look at incidence and mortality for breast cancer, and try to answer the following two questions: among the students at the Upper Austrian University of Applied Sciences (“FH”), how many occurrences of breast cancer should we expect per year? And how many of the female FH-students will probably die from breast cancer at the end of their life?

We know that:

- the Upper Austrian University of Applied Sciences has about 5'000 students, about half of which are female.
- breast cancer hits predominantly women.
- the *incidence* of breast cancer in the age group 20–30 is about 10, where *incidence* is typically defined as the new occurrences of a disease per year per 100'000 people.
- 3.8% of all women die of cancer.

From these pieces of information, we can obtain the following parameters for our calculations:

- $n = 2'500$
- $p_{\text{incidence}} = 10/100'000$
- $p_{\text{mortality}} = 3.8/100$.

The 95% confidence interval for the incidence of breast cancer is [-0.7, 1.2], and for the number of deaths [76, 114]. So we expect that every year most likely none or one of the FH-students will be diagnosed with breast cancer, but between 76 and 114 of the current female students will eventually die from this disease.

9.2 Tests Using Frequency Tables

If the data can be organized in a set of categories, and they are given as *frequencies*, i.e., the total number of samples in each category (not as percentages), the tests described in this section are appropriate for the data analysis.

Many of these tests analyze the *deviation from an expected value*. Since the chi-square distribution characterizes the variability of data (in other words, their deviation from a mean value), many of these tests refer to this distribution, and are accordingly termed *chi-square tests*.

Table 9.2 Corresponding *expected values* for Table 9.1

	Right handed	Left handed	Total
Males	45.2	6.8	52
Females	41.8	6.2	48
Total	87	13	100

Table 9.3 Representation of data as raw-data; *id* indicates the *subject-ID*

Id	Gender	Handedness
0	Male	Right
1	Female	Left
2	Female	Right
3	Male	Right
4	Female	Left
:	:	:

When n is the total number of observations included in the table, the expected value for each cell in a two-way table is (Table 9.2)

$$\text{expected Frequency} = \frac{\text{RowTotal} * \text{ColumnTotal}}{n}. \quad (9.2)$$

Assume that we have observed absolute frequencies o_i and expected absolute frequencies e_i . Under the null hypothesis all the data come from the same population, and the test statistic

$$V = \sum_i \frac{(o_i - e_i)^2}{e_i} \approx \chi_f^2 \quad (9.3)$$

follows a chi-square distribution with f degrees of freedom.¹ i might denote a simple index running from 1, ..., f , or a multi-index (i_1, \dots, i_n) running from (1, ..., 1) to (f_1, \dots, f_n) , and $f = \sum_{i=1}^n f_i$.

Some analysis programs require the input data not in a frequency table format, but in a format corresponding to the original raw data. For example, the Table 9.1 with left- and right-handed males and females would be represented as (Table 9.3)

¹ The statistic in Eq. 9.3 is also called *Pearson's chi-square statistic*, and is actually only of the of the *power divergence family of statistics* used for categorical data. The output of the *pingouin* command `pg.chi2_independence` also gives access to a number of other statistics from this family.

Table 9.4 Representation of data as raw-data, with numerical representation of the groups

Id	Gender	Handedness
0	0	0
1	1	1
2	1	0
3	0	0
4	1	1
:	:	:

Equivalently, the categorical values might be represented as integers:

`male/female` → 0/1 and `right/left` → 0/1. For more than two variables, so-called “dummy coding” of categorical variables should be used.²

A Python function converting frequency tables (e.g., Table 9.1) into the corresponding integer format (Table 9.4) can be found in `ISP_compGroups.py` (see the function `ISP_compGroups.frequency2events`, link at the end of this section).

9.2.1 One-Way Chi-Square Test

For example, assume that you go hiking with your friends. Every evening, you draw lots who has to do the washing up. But at the end of the trip, you seem to have done most of the work:

You	Peter	Hans	Paul	Mary	Joe
10	6	5	4	5	3

You expect that there has been some foul play, and calculate how likely it is that this distribution came up by chance. The

$$\text{expectedFrequency} = \frac{n_{\text{total}}}{n_{\text{people}}} \quad (9.4)$$

is 5.5. The probability that this distribution came up by chance is

```
V, p = stats.chisquare(data)
print(p)
>>> 0.3731
```

In other words, you doing a lot of the washing up really could have been by chance!

² <https://www.statsmodels.org/stable/examples/notebooks/generated/contrasts.html>.

9.2.2 Chi-Square Contingency Test

When data can be arranged in rows and columns, we can check if the numbers in the individual columns are contingent on the row value. For this reason, this test is sometimes called *contingency test*. Using the example in Table 9.1, if females were more left-handed than males, the ratio $\frac{\text{left-handed}}{\text{right-handed}}$ would be contingent on the row and larger for females than for males.

The chi-square contingency test is based on a test statistic that measures the divergence of the observed data from the values that would be expected under the null hypothesis of “no association” (e.g., Table 9.2).

a) Assumptions of the Chi-Square Contingency Test

The test statistic V is approximately χ^2 distributed, if

- for all absolute expected frequencies e_i holds: $e_i \geq 1$, and
- for at least 80% of the absolute expected frequencies e_i holds: $e_i \geq 5$.

For small sample numbers, corrections should be made for some bias that is caused by the use of the continuous chi-squared distribution, while the frequencies are by definition integers. This correction is referred to as *Yates correction*.

b) Degrees of Freedom

The degrees of freedom (DOF) can be computed by the numbers of absolute observed frequencies which can be chosen freely. For example, only one cell of a 2×2 table with the sums at the side and bottom needs to be filled, and the others can be found by subtraction. In general, an $r \times c$ table with r rows and c columns has

$$df = (r - 1) \times (c - 1) \quad (9.5)$$

degrees of freedom. We know that the sum of absolute expected frequencies is

$$\sum_i o_i = n. \quad (9.6)$$

We might have to subtract from the number of degrees of freedom the number of parameters we need to estimate from the sample, since this implies further relationships between the observed frequencies.

Table 9.5 General Structure of 2×2 Frequency Tables

		B		Total
		0	1	
A	0	a	b	$a+b$
	1	c	d	$c+d$
Total		$a+c$	$b+d$	$N=a+b+c+d$

c) Example 1

The Python command `stats.chi2_contingency` returns the following list: (χ^2 -value, p-value, degrees of freedom, expected values).

```
data = np.array([[43, 9],
                [44, 4]])
V, p, dof, expected = stats.chi2_contingency(data)
print(p)
>>> 0.3004
```

For the example data in Table 9.1, the results are ($\chi^2 = 1.1$, $p = 0.3$, $df = 1$). In other words, there is no indication that there is a difference in left-handed people vs right-handed people between males and females.

Note: These values assume the default setting, which uses the *Yates correction*. Without this correction, i.e., using Eq. 9.3, the results are $\chi^2 = 1.8$, $p = 0.18$.

d) Example 2

The Chi-square test can be used to generate a “quick and dirty” test of normality, e.g.,

H_0 : The random variable X is symmetrically distributed versus

H_1 : the random variable X is not symmetrically distributed.

We know that in case of a symmetrical distribution the arithmetic mean \bar{x} and median should be nearly the same. So a simple way to test this hypothesis would be to count how many observations are less than the mean (n_-) and how many observations are larger than the arithmetic mean (n_+). If mean and median are the same then 50% of the observation should smaller than the mean and 50% should be larger than the mean. It holds

$$V = \frac{(n_- - n/2)^2}{n/2} + \frac{(n_+ - n/2)^2}{n/2} \approx \chi^2_1. \quad (9.7)$$

e) Comments

The Chi-square test is a pure hypothesis test. It tells you if the observed frequency can be due to a random sample selection from a single population. A number of different expressions have been used for chi-square tests, which are due to the original derivation of the formulas (from the time before computers were pervasive). Expression such as 2×2 tables, r - c tables, or Chi-square test of contingency all refer to frequency tables and are typically analyzed with chi-square tests.

9.2.3 Fisher's Exact Test

If the requirement that 80% of cells should have expected values of at least 5 is not fulfilled, *Fisher's exact test* should be used. This test is based on the observed row and column totals. The method consists of evaluating the probability associated with all possible 2×2 tables which have the same row and column totals as the observed data, making the assumption that the null hypothesis (i.e., that the row and column variables are unrelated) is true. In most cases, Fisher's exact test is preferable to the chi-square test. But until the advent of powerful computers, it was not practical. You should use it up to approximately 10–15 cells in the frequency tables. It is called “exact” because the significance of the deviation from a null hypothesis can be calculated exactly, rather than relying on an approximation that becomes exact in the limit as the sample size grows to infinity, as with many statistical tests.

In using the test, you have to decide if you want to use a one-tailed test or a two-tailed test. The former one looks for the probability to find a distribution as extreme as or more extreme than the observed one. The latter one (which is the default in Python) also considers tables as extreme in the opposite direction.

Note: The python command `stats.fisher_exact` returns by default the p-value for *finding a value as extreme or more extreme than the observed one*. According to Altman (1999), this is a reasonable approach, although not all statisticians agree on that point.

Example: “A Lady Tasting Tea”

³R. A. Fisher was one of the founding fathers of modern statistics. One of his early experiments, and perhaps the most famous, was to test an English lady’s claim that she could tell whether milk was poured before tea or not. Here is an account of the seemingly trivial event that had the most profound impact on the history of modern statistics, and hence, arguably, modern quantitative science (Box 1978) (Fig. 9.1).

Already, quite soon after he had come to Rothamstead, his presence had transformed one commonplace tea time to an historic event. It happened one afternoon when he drew a cup of tea from the urn and offered it to the lady beside him, Dr. B. Muriel Bristol, an algologist. She declined it, stating that she preferred a cup into which the milk had been poured first. “Nonsense,” returned Fisher, smiling, “Surely it makes no difference.” But she maintained, with emphasis, that of course it did. From just behind, a voice suggested, “Let’s test her.” It was William Roach who was not long afterward to marry Miss Bristol. Immediately, they embarked on the preliminaries of the experiment, Roach assisting with the cups and exulting that Miss Bristol divined correctly more than enough of those cups into which tea had been poured first to prove her case.

³ Adapted from Stat Labs: Mathematical statistics through applications by D. Nolan and T. Speed, Springer-Verlag, New York, 2000.



Fig. 9.1 First milk, then tea (left)—or first tea, then milk (right): Could you taste the difference? (Published with the kind permission of ©Thomas Haslwanter 2015. All rights reserved)

Miss Bristol's personal triumph was never recorded, and perhaps Fisher was not satisfied at that moment with the extempore experimental procedure. One can be sure, however, that even as he conceived and carried out the experiment beside the trestle table, and the onlookers, no doubt, took sides as to its outcome, he was thinking through the questions it raised.

The real scientific significance of this experiment is in these questions. These are, allowing incidental particulars, the questions one has to consider before designing an experiment. We will look at these questions as pertaining to the “lady tasting tea”, but you can imagine how these questions should be adapted to different situations.

- *What should be done about chance variations in the temperature, sweetness, and so on?* Ideally, one would like to make all cups of tea identical except for the order of pouring milk first or tea first. But it is never possible to control all of the ways in which the cups of tea can differ from each other. If we cannot control these variations, then the best we can do—we do mean the “best”—is by randomization.
- *How many cups should be used in the test? Should they be paired? In what order should the cups be presented?* The key idea here is that the number and ordering of the cups should allow a subject ample opportunity to prove his or her abilities and keep a fraud from easily succeeding at correctly discriminating the order of pouring in all the cups of tea served.
- *What conclusion could be drawn from a perfect score or from one with one or more errors?* If the lady is unable to discriminate between the different orders of pouring, then by guessing alone, it should be highly unlikely for that person to determine correctly which cups are which for all of the cups tested. Similarly, if she indeed possesses some skill at differentiating between the orders of pouring, then it may be unreasonable to require her to make no mistakes so as to distinguish her ability from a pure guesser.

An actual scenario described by Fisher and told by many others as the “lady tasting tea” experiment is as follows.

- For each cup, we record the order of actual pouring and what the lady says the order is. We can summarize the result by a table like this:

		<i>Order of actual pouring</i>		<i>Total</i>
<i>Lady says</i>	<i>Tea first</i>	<i>Milk first</i>		
	<i>Tea first</i>	<i>Milk first</i>	<i>Total</i>	
<i>Lady says</i>	a	b	$a + b$	
<i>Milk first</i>	c	d	$c + d$	
<i>Total</i>	$a + c$	$b + d$	n	

Here n is the total number of cups of tea made. The number of cups where tea is poured first is $a + c$ and the lady classifies $a + b$ of them as tea first. Ideally, if she can taste the difference, the counts b and c should be small. On the other hand, if she cannot really tell, we would expect a and c to be about the same.

- Suppose now that to test the lady’s abilities, 8 cups of tea are prepared, 4 tea first, 4 milk first, and she is informed of the design (that there are 4 cups milk first and 4 cups tea first). Suppose also that the cups are presented to her in random order. Her task then is to identify the 4 cups milk first and 4 cups tea first.

This design fixes the row and column totals in the table above to be 4 each. That is,

$$a + b = a + c = c + d = b + d = 4.$$

With these constraints, when any one of a, b, c, d is specified, the remaining three are uniquely determined:

$$b = 4 - a, \quad c = 4 - a, \quad \text{and} \quad d = a.$$

In general, for this design, no matter how many cups (n) are served, the row total $a + b$ will equal $a + c$ because the subject knows how many of the cups are “tea first” (or one kind as supposed to the other). So once a is given, the other three counts are specified.

- We can test the discriminating skill of the lady, if any, by randomizing the order of the cups served. If we take the position that she has no discriminating skill, then the randomization of the order makes the 4 cups chosen by her as tea first equally likely to be any 4 of the 8 cups served. There are $\binom{8}{4} = 70$ (in Python given by `scipy.misc.comb(8, 4, exact=True)`) possible ways to classify 4 of the 8 cups as “tea first”. If the subject has no ability to discriminate between two preparations, then by the randomization, each of these 70 ways is equally likely. Only one of 70 ways leads to a completely correct classification. So someone with no discriminating skill has 1/70 chance of making no errors.

- It turns out that, if we assume that she has no discriminating skill, the number of correct classifications of tea first (“a” in the table) has a “hypergeometric” probability distribution (`hd=stats.hypergeom(8,4,4)`) in Python, see Sect. 6.2.4). There are 5 possibilities for “a”: 0, 1, 2, 3, 4, and the corresponding probabilities (and Python commands for computing the probabilities) are tabulated below.

Number of correct calls	Python command	Probability
0	<code>hd.pmf(0)</code>	1/70
1	<code>hd.pmf(1)</code>	16/70
2	<code>hd.pmf(2)</code>	36/70
3	<code>hd.pmf(3)</code>	16/70
4	<code>hd.pmf(4)</code>	1/70

- With these probabilities, we can compute the p-value for the test of the hypothesis that the lady cannot tell between the two preparations. Recall that the p-value is the probability of observing a result as extreme or more extreme than the observed result assuming the null hypothesis. If she makes all correct calls, the p-value is 1/70 and if she makes one error (3 correct calls) then the p-value is $1/70 + 16/70 \sim 0.24$.

The test described above is known as “Fisher’s exact test”, and the implementation is quite trivial:

```
odds_ratio, p = stats.fisher_exact(obs, alternative='greater')
```

where *obs* is the matrix containing the observations.

9.2.4 McNemar’s Test

In statistics, McNemar’s test is a statistical test used on paired nominal data. It is applied to 2×2 contingency tables with a dichotomous trait (“0/1”), with matched pairs of subjects. Although the McNemar test bears a superficial resemblance to a test of categorical association, as might be performed by a 2×2 chi-square test or a 2×2 Fisher exact probability test, it is doing something quite different. The test of association examines the relationship that exists among the cells of the table. The McNemar test examines the difference between the proportions that derive from the marginal sums of the table (see Table 9.5): $p_A = (a + b)/N$ and $p_B = (a + c)/N$. The question in the McNemar test is: do these two proportions, p_A and p_B , significantly differ? And the answer it receives must take into account the fact that the two proportions are not independent. The correlation of p_A and p_B is occasioned by the fact that both include the quantity in the upper left cell of the table.

McNemar's test can be used, for example, in studies in which patients serve as their own control, or in studies with "before and after" design. Or in other words: in contrast to the chi-square test, the *same* subjects/parts are categorized in the rows and columns.

Example

In the following example, a researcher attempts to determine if a drug has an effect on a particular disease. Counts of individuals are given in the table, with the diagnosis (disease: present or absent) before treatment given in the rows, and the diagnosis after treatment in the columns. The test requires the same subjects to be included in the before-and-after measurements (matched pairs) (Table 9.6).

In this example, the null hypothesis of "marginal homogeneity" would mean there was no effect of the treatment. From the above data, the McNemar test statistic with Yates's continuity correction is

$$\chi^2 = \frac{(|b - c| - \text{correction Factor})^2}{b + c} \quad (9.8)$$

where χ^2 has a chi-squared distribution with 1 degree of freedom. For small sample numbers the *correction Factor* should be 0.5 (*Yates's correction*) or 1.0 (*Edward's correction*). (For $b + c < 25$, the binomial calculation should be performed, and indeed, most software packages simply perform the binomial calculation in all cases, since the result then is an exact test in all cases.) Using Yates's correction, we get

$$\chi^2 = \frac{(|121 - 59| - 0.5)^2}{121 + 59}. \quad (9.9)$$

The resulting value is 21.01, which is extremely unlikely from the distribution implied by the null hypothesis ($p_b = p_c$). Thus the test provides strong evidence to reject the null hypothesis of no treatment effect.

Table 9.6 McNemar's Test: example

	After: present	After: absent	Total
Before: present	101	121	222
Before: absent	59	33	92
Total	160	154	314

To implement the McNemar's test in Python use

```
from statsmodels.stats.contingency_tables import mcnemar

obs = [[a,b], [c, d]]
chi2, p = mcnemar(obs)
```

with *obs* again representing the observation matrix.

9.2.5 Cochran's Q Test

Cochran's Q test is an extension to McNemar's test. It provides a method for testing for differences between three or more matched sets of frequencies. You would use this test, for example, if you have exactly the same samples analyzed by three different laboratories, and you want to check if the results are statistically equivalent.

Like the McNemar's test, Cochran's Q test is a hypothesis test where the response variable can take only two possible outcomes (coded as 0 and 1). It is a non-parametric statistical test to verify if k treatments have identical effects. Cochran's Q test should not be confused with *Cochran's C test*, which is a variance outlier test.

Example

12 subjects are asked to perform 3 tasks. The outcome of each task is *success* or *failure*. The results are coded 0 for *failure* and 1 for *success*. In the example, subject 1 was successful in task 2, but failed tasks 1 and 3 (see Table 9.7).

Table 9.7 Cochran's Q Test: Success or failure for 12 subjects on 3 tasks

Subject	Task 1	Task 2	Task 3
0	0	1	0
1	1	1	0
2	1	1	1
3	0	0	0
4	1	0	0
5	0	1	1
6	0	0	0
7	1	1	0
8	0	1	0
9	0	1	0
10	0	1	0
11	0	1	0

The null hypothesis for the Cochran’s Q test is that there are no differences between the variables. If the calculated probability p is below the selected significance level, the null hypothesis is rejected, and it can be concluded that the proportions in at least 2 of the variables are significantly different from each other. For our example (Table 9.7), the analysis of the data provides $Cochran’s\ Q = 8.6667$ and a significance of $p = 0.013$. In other words, at least one of the three tasks is easier or harder than the others.

To implement the Cochran’s Q test in Python, use

```
from statsmodels.stats.contingency_tables import cochrans_q

q_stat, p = cochrans_q(obs)
```



Code: `ISP_compGroups.py`⁴ Analysis of categorical data, using `scipy`, `statsmodels`, and `pingouin`: once the correct test is selected, the computational steps are trivial. Note that since `pingouin` requires a somewhat different input format, the required conversion functions are also provided in this module.

9.3 Exercises

1. Fisher’s Exact Test—The Tea Experiment

At a party, a lady claimed to be able to tell whether the tea or the milk was added first to a cup. Fisher proposed to give her eight cups, four of each variety, in random order. One could then ask what the probability was for her getting the number she got correct, but just by chance.

The experiment provided the Lady with 8 randomly ordered cups of tea—4 prepared by first adding milk, 4 prepared by first adding the tea. She was to select the 4 cups prepared by one method. (This offered the Lady the advantage of judging cups by comparison.)

The null hypothesis was that the Lady had no such ability. (In the real, historical experiment, the lady got all eight cups correct.)

- Calculate if the claim of the Lady is supported if she gets three out of the four pairs correct.

(Correct answer: No. If she gets three correct, that chance that a selection of “three or greater” was random is 0.243. She needs to get all four correct, if we set the rejection threshold at 0.05.)

⁴ [`<ISP2e>/09_TestsCategoricalData/compGroups/ISP_compGroups.py`](#).

2. Chi² Contingency Test (1 DOF)

A test of the effect of a new drug on the heart rate has yielded the following results:

	Heart rate		<i>Total</i>
	<i>Increased</i>	<i>NOT-increased</i>	
<i>Treated</i>	36	14	50
<i>Not treated</i>	30	25	55
<i>Total</i>	66	39	105

- Does the drug affect the heart rate? (Correct answer: no)
- What would be the result if the response in one of the not-treated persons would have been different? Perform this test with and without the Yates correction.

(Correct answer:

without Yates correction: yes, $p = 0.042$

with Yates correction: no, $p = 0.067$)

	Heart rate		<i>Total</i>
	<i>Increased</i>	<i>NOT-increased</i>	
<i>Treated</i>	36	14	50
<i>Not treated</i>	29	26	55
<i>Total</i>	65	40	105

3. One way Chi²-Test (>1 DOF)

The city of Linz wants to know if people want to build a long beach along the Danube. They interview local people, and decide to collect 20 responses from each of the five age groups: (<15, 15–30, 30–45, 45–60, >60).

The questionnaire states: “A beach-side development will benefit Linz.” and the possible answers are

1	2	3	4
Strongly agree	Agree	Disagree	Strongly Disagree

The city council wants to find out if the age of people influenced feelings about the development, particularly of those who felt negatively (i.e., “disagreed” or “strongly disagreed”) about the planned development.

Age group (Type)	Frequency of negative responses (Observed values)
<15	4
15–30	6
30–45	14
45–60	10
>60	16

The categories seem to show large differences of opinion between the groups.

- Are these differences significant? (Correct answer: yes, $p = 0.034$.)
- How many degrees of freedom does the resulting analysis have? (Correct answer: 4.)

4. McNemar's Test

In a lawsuit regarding a murder, the defense uses a questionnaire to show that the defendant is insane. As a result of the questionnaire, the accused claims “not guilty by reason of insanity”.

In reply, the state attorney wants to show that the questionnaire does not work. He hires an experienced neurologist, and presents him with 40 patients, 20 of whom have completed the questionnaire with an “insane” result, and 20 with a “sane” result. When examined by the neurologist, the result is mixed: 19 of the “sane” people are found sane, but 6 of the 20 “insane” people are labeled as sane by the expert.

	Sane by expert	Insane by expert	Total
Sane	19	1	20
Insane	6	14	20
Total	22	18	40

- Is this result significantly different from the questionnaire? (Correct answer: no)
- Would the result be significantly different, if the expert had diagnosed all “sane” people correctly? (Correct answer: yes.)

Chapter 10

Analysis of Survival Times



When analyzing survival times, different problems come up than the ones discussed so far. One question is how to deal with subjects dropping out of a study. For example, assume that we test a new cancer drug. While some subjects die, others may believe that the new drug is not effective and decide to drop out of the study before the study is finished.

The term used for this type of study is *survival analysis*, although the same methods are also used to analyze similar problems in other areas. For example, these techniques can be used to investigate how long a machine lasts before it breaks down, or how long people subscribe to mailing lists (where the “death” corresponds to unsubscribing from a mailing list).

10.1 Survival Distributions

The Weibull distribution is often used for modeling reliability data or survival data. Since it was first identified by Fréchet (in 1927), but described in detail by Weibull (in 1951), it is sometimes also found under the name Fréchet distribution.

In `scipy.stats`, the Weibull distribution is available under the name `weibull_min`, or equivalently `freshet_r` for *Fréchet right*. (The complementary `weibull_max`, also called `freshet_l` for *Fréchet left*, is simply mirrored about the origin.)

The Weibull distribution is characterized by a shape parameter, the *Weibull Modulus k* (see also Sect. 6.5.2). All Python-distributions offer a convenient method `fit`, which allows quick fitting of the distribution parameters:

Listing 10.1: L10_1_weibull_demo.py

```
""" Example of fitting the Weibull modulus. """

# author: Thomas Haslwanter, date: June-2022

# Import standard packages
import matplotlib.pyplot as plt
import scipy as sp
from scipy import stats

# Generate some sample data, with a Weibull modulus of 1.5
WeibullDist = stats.weibull_min(1.5)
data = WeibullDist.rvs(500)

# Now fit the parameter
fitPars = stats.weibull_min.fit(data)

# Note: fitPars contains (WeibullModulus, Location, Scale)
print(f'The fitted Weibull modulus is {fitPars[0]:5.2f}, ' +
      'compared to the exact value of 1.5 .')
```

10.2 Survival Probabilities

For the statistical analysis of survival data, Cam Davidson-Pilon has developed the Python package *lifelines*. It can be installed with

```
pip install lifelines
```

A very extensive documentation, which also includes an introduction to survival analysis and survival regression modeling, is available under <http://lifelines.readthedocs.org/>.

10.2.1 Censorship

The difficulty of using data for survival analysis is that at the end of a study, many individuals may be still “alive”. In statistics, the expression for measurement values that are only partially known is *censorship* or *censoring*. As an example let us consider a mailing list, whose subscribers fall into two subgroups. Group One quickly gets tired of the emails and unsubscribes after three months. Group Two enjoys it and typically subscribes for one and a half years. We perform a study which lasts one year and want to investigate the average subscription duration (Fig. 10.1):

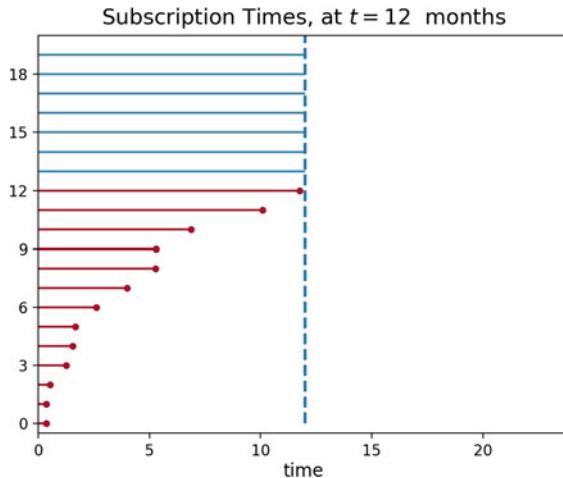


Fig. 10.1 Dummy results from a study on the subscription behavior of a mailing list



Code: `ISP_lifelinesDemo.py`¹ Graphical representation of lifelines.

The red lines denote the subscription time of individuals where the dropout event has been observed, and the blue lines denote the subscription time of the right-censored individuals (dropouts have not been observed). If we are asked to estimate the average subscription time of our population, and we naively decided not to include the right-censored individuals, it is clear that we would be severely underestimating the true average subscription time.

A similar, further problem occurs if some subjects increase their privacy-settings in the middle of the study, i.e., they forbid us to monitor them before the study is over. Also these data are right-censored data.

10.2.2 Kaplan–Meier Survival Curve

A clever way to deal with these problems is the description of such data with the Kaplan–Meier curve, described in detail in Altman (1999). First, the time is subdivided into small periods. Then the probability is calculated that a subject survives a given period. The survival probability is given by

$$p_k = p_{k-1} * \frac{r_k - f_k}{r_k}, \quad (10.1)$$

¹ [<ISP2e>/10_SurvivalAnalysis/lifelinesDemo/ISP_lifelinesDemo.py](#).

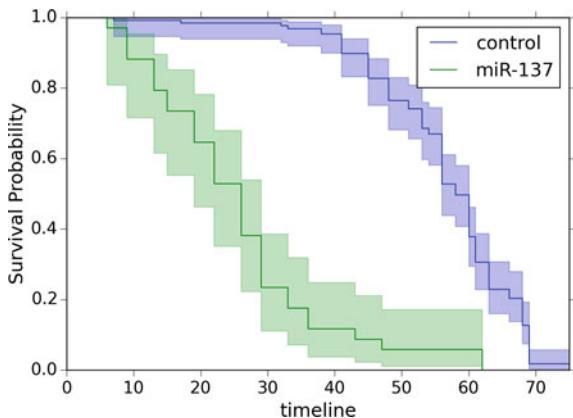


Fig. 10.2 Survival Probability of two groups of Drosophila flies. The shaded areas indicate the 95% confidence intervals

where p_k is the probability to survive period k ; r_k is the number of subjects still at risk (i.e., still being followed up) immediately before the k^{th} day, and f_k is the number of observed failures on the day k . The curve describing the resulting survival probability is called *life table*, *survival curve*, or *Kaplan–Meier curve* (see Fig. 10.2).

The following data show results from a study with fruit flies of the genus *Drosophila*. The numbers give the genotypes of the flies, and number of days survived. Since we work with flies, we do not need to worry about left-censoring: we know the birth date of all flies. We do have issues with accidentally killing some or if some escape. These would be right-censored as we do not actually observe their death due to “natural” causes.

Listing 10.2: lifelines_survival.py

```
""" Graphical representation of survival curves, and
comparison of two curves with logrank test.
"miR-137" is a short non-coding RNA molecule that functions
to regulate the expression levels of other genes.
"""

# author: Thomas Haslwanter, date: June-2022

# Import standard packages
import matplotlib.pyplot as plt

# additional packages
import sys
sys.path.append(r'..\Code_Quantlets\Utilities')
import ISP_mystyle
```

```

from lifelines.datasets import load_waltons
from lifelines import KaplanMeierFitter
from lifelines.statistics import logrank_test

# Set my favorite font
ISP_mystyle.setFonts(18)

# Load and show the data
df = load_waltons() # returns a Pandas DataFrame

print(df.head())
"""
    T   E   group
0   6   1  miR-137
1  13   1  miR-137
2  13   1  miR-137
3  13   1  miR-137
4  19   1  miR-137
"""

T = df['T']
E = df['E']

groups = df['group']
ix = (groups == 'miR-137')

kmf = KaplanMeierFitter()

kmf.fit(T[~ix], E[~ix], label='control')
ax = kmf.plot()

kmf.fit(T[ix], E[ix], label='miR-137')
kmf.plot(ax=ax)

plt.ylabel('Survival Probability')
outFile = 'lifelines_survival.png'
ISP_mystyle.showData(outFile)

# Compare the two curves
results = logrank_test(T[ix], T[~ix], event_observed_A=E[ix],
                       event_observed_B=E[~ix])
results.print_summary()

```

This code produces the following output:

```

Results
t 0: -1
alpha: 0.95
df: 1
test: logrank
null distribution: chi squared

p-value | test statistic | test result | is significant
0.00000 |          122.249 | Reject Null |      True

```

Note that the survival curve changes only when a “failure” occurs, i.e., when a subject dies. *Censored* entries, describing either when a subject drops out of the study or when the study finishes, are taken into consideration at the “failure” times, but otherwise do not affect the survival curve.

10.3 Comparing Survival Curves in Two Groups

The most common test for comparing independent groups of survival times is the *logrank test*. This test is a nonparametric hypothesis test, testing the probability that both groups come from the same underlying population. To explore the effect of different variables on survival, more advanced methods are required. For example, the *Cox Regression model*, also called *Cox Proportional Hazards model* introduced by Cox in 1972 is used widely when it is desired to investigate several variables at the same time.

These tests, as well as other models for the analysis of survival data, are available in the *lifelines* package and are easy to apply once you know how to use Python.

Part III

Statistical Modeling

Hypothesis tests can decide if two or more sets of data samples come from the same population or from different ones. But they cannot quantify the strength of a relationship between two or more variables. They also cannot find repeating patterns in those relationships. Such questions, which also include the quantitative prediction of variables, are addressed in the third part of this book. While the basic algebraic tools that come with Python may suffice for simple problems like line fits or the determination of correlation coefficients, a number of packages significantly extend the power of Python for statistical data analysis and modeling. Part III of the book will show applications of the following packages:

- *Statsmodels*
- *PyMC*
- *scikit-learn*
- *scikits.bootstrap*

This part comprises four chapters. The first chapter describes how patterns in signals can be detected: the occurrence of a shorter pattern in a longer signal (“cross correlation”), covariation of two signals of the same length (“correlation coefficient”), and repeating patterns within signals (“autocorrelation”). A brief foray into “time series analysis” gives an example of practical applications of autocorrelation. The second chapter of this part introduces linear regression analysis, the standard tool to quantify the linear dependence of an output signal on one or more inputs. It forms the basis of statistical modeling. The third chapter about “generalized linear models” presents two examples how linear models can be made more flexible, to extend the power of linear models to nonlinear relationships. And the last chapter contains an introduction to Bayesian statistics, where probability expresses a *degree of belief* in an event (in contrast to the *frequency* of events). That chapter is rounded off by a practical example of a running Markov Chain–Monte Carlo simulation.

Chapter 11

Finding Patterns in Signals



The previous chapters have all looked at the analysis of groups of data where the sequence of the data was irrelevant. However, for many real-life quantities the sequence is crucial. For example, in medicine, geology, and econometrics, the timing of events makes a big difference: sell your stocks one day too late, and you may have turned from a millionaire to a pauper.

This chapter describes different aspects of finding patterns in signals. It starts out with the description of *cross correlation*, where one looks at the occurrence of a shorter feature in a longer signal. Two special cases are investigated in more detail. First, when the two signals have the same length one can ask how strong the linear relationship is between the two variables. This comparison of signals requires a normalization to eliminate trivial artifacts, and leads to the definition of the *correlation coefficient*. For multivariate data, the generalization from the correlation coefficient to the *correlation matrix* is explained. An intuitive interpretation of the correlation coefficient is obtained by looking how it is related to the best line fit to the two variables. In that case the square of the correlation coefficient, the *coefficient of determination*, quantifies which part of the signal change in one variable is explained by the corresponding change in the other variable. The second special case is obtained by comparing one signal to shifted versions of itself. This is called *autocorrelation*, and can be used to find hidden systematic patterns in signals. The final section of this chapter shows how the autocorrelation is used in time-series analysis (TSA), to obtain the maximum amount of information from a time series of data.

11.1 Cross Correlation

Cross correlation is a measure of similarity of two series as a function of the displacement of one relative to the other. This is also known as a *sliding dot product* or *sliding inner-product*. It is commonly used for searching a long signal for a shorter, known feature. It has applications in diverse areas such as pattern recognition, single

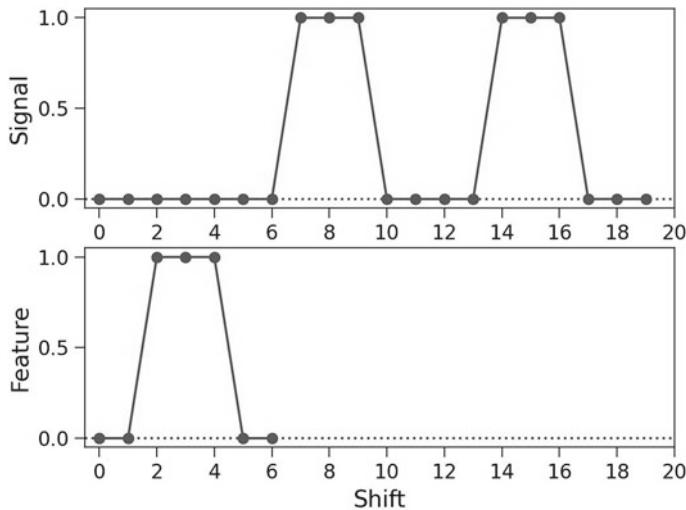


Fig. 11.1 Sample signal and feature to visualize the principle of cross correlation

particle analysis, electron tomography, averaging, crypto-analysis, and neurophysiology. The cross correlation is similar in nature to the convolution of two functions: the only difference is that the ordering of the second signal (“feature”) is inverted.

Let us consider how we might assess the similarity of two signals, which we call here **signal** and **feature** (see Fig. 11.1). To find similarities we need some kind of “similarity function” such that the function has a maximum when the feature matches the signal, and that decreases as the difference between signal and feature increases.

It can be shown that the dot-product satisfies both of these properties. Thus, all we need to do to compare part of the signal with the features is to multiply that part of the signal with the feature! If we want to find out how much the feature needs to be shifted to match the signal, we calculate the similarity for different relative shifts and choose the shift with the maximum similarity.

The attached program `ISP_corrVis.py`¹ allows an interactive exploration of the sliding dot-product to produce the cross correlation of **signal** and **feature**. The result is shown in Fig. 11.2. The following characteristics can be readily seen:

- In the starting position shown in Fig. 11.1 the dot product between signal and feature is zero.
- A shift of the feature by 5 or 12 steps produces maximum overlap.
- With maximum overlap the dot product between signal and feature is 3.
- The feature can be shifted by six points to the left, before it goes “out of range”.

¹ [ISP2e>/11_Pattern/correlation/ISP_corrVis.py](https://ISP2e.readthedocs.io/en/latest/11_Pattern/correlation/ISP_corrVis.py).

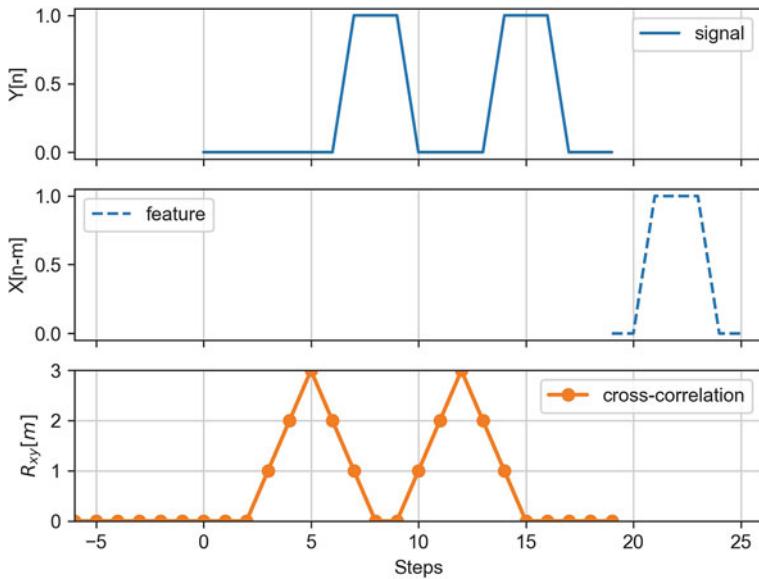


Fig. 11.2 Output of the program `ISP_corrVis.py`, to interactively visualize the principle of cross correlation (from CQ `ISP_corrVis.py`)

Table 11.1 Manual calculation of the cross correlation between $sig_1 = [2, 1, 4, 3]$ and $sig_2 = [1, 3, 2]$. Missing elements at the beginning and at the end of sig_1 are here set to 0

0	0	2	1	4	3	0	0	
1	3	2						$\rightarrow 2*2 = 4$
	1	3	2					$\rightarrow 2*3 + 1*2 = 8$
		1	3	2				$\rightarrow 2*1 + 1*3 + 4*2 = 13$
			1	3	2			$\rightarrow 1*1 + 4*3 + 3*2 = 19$
				1	3	2		$\rightarrow 4*1 + 3*3 = 13$
					1	3	2	$\rightarrow 3*1 = 3$

For multiplications with elements outside the given signal/feature range (e.g., point 20 in Fig. 11.2), the corresponding missing data are replaced by zeros.

To demonstrate the principle of cross correlation, Table 11.1 explicitly goes through the calculation for the signals $sig_1 = [2, 1, 4, 3]$ and $sig_2 = [1, 3, 2]$:

The same result is obtained with the `numpy` command

`np.correlate(sig_1, sig_2, mode='full')`. (The last option specifies how missing values at the beginning and at the end should be handled; in Table 11.1 we have set them to 0.)

To summarize, cross correlation provides two pieces of information:

- *How similar* signal and feature are (through the maximum of the cross correlation).
- *Where* the similarity occurs (through the location of the maximum).

11.2 Correlation Coefficient

Data sets with two variables of the same length are called “bivariate data”. The *correlation coefficient* measures the linear association between the two variables. In contrast, a *linear regression* is used for the prediction of the value of one variable from another.

11.2.1 Covariance

The simplest way to describe the *correlation coefficient* is by first defining the term “covariance”. In the first visual analysis of data it is helpful to see if two variables “vary together” or “co-vary”. For example, Fig. 11.3 shows two characteristics of the *iris data set*, one of the best known databases to be found in the pattern recognition literature (<https://archive.ics.uci.edu/ml/datasets/iris>).

```
# Get the data
import seaborn as sns
iris = sns.load_dataset('iris')

# Make the plot
iris.plot('petal_length', 'petal_width',
          kind='scatter')
```

As the petal length increases in Fig. 11.3, also the petal width goes up. This signal feature is quantified by the *covariance*. For two variables x and y , the corresponding *sample covariance* s_{xy} is defined as

$$s_{xy} = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y}). \quad (11.1)$$

Note that s_{xx} and s_{yy} thus give the sample variance of x and y , respectively. A *positive covariance* s_{xy} indicates that x and y rise and fall together; in contrast, a *negative covariance* indicates that the rise of x is accompanied by the fall of y , and vice versa. While this information can be helpful in itself, the usefulness of the covariance is hobbled by the fact that its magnitude depends on the units used. For example, if the petal dimensions of the data in Fig. 11.3 are expressed in *mm* instead of *cm*, the corresponding covariance increases by 100!

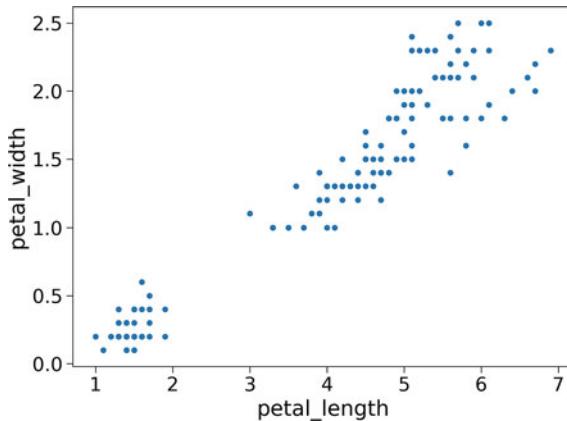


Fig. 11.3 The *petal_length* and *petal_width* (here in cm) are clearly correlated. Clusters in these data correspond to different plant classes (see Fig. 11.7)

To eliminate that problem, the variables x and y in Eq. 11.1 can be normalized by division through the corresponding standard deviation, leading to the definition of *correlation* described next.

11.2.2 Pearson Correlation Coefficient

The *correlation* between two variables answers the question: “Are the two variables linearly related? I.e., if one variable changes, does the other also change?” If the two variables are normally distributed, the standard measure of determining the *correlation coefficient*, often ascribed to Pearson, is

$$r(x, y) = \sum_{i=1}^n \left(\frac{(x_i - \bar{x})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2}} * \frac{(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}} \right). \quad (11.2)$$

With the sample covariance s_{xy} defined as above, and using the notation that $s_x = \sqrt{s_{xx}}$, i.e., that s_x denotes the sample standard deviations of the x values, Eq. 11.2 can also be written as

$$r = \frac{s_{xy}}{s_x \cdot s_y}. \quad (11.3)$$

Since the values are now normalized, Pearson’s correlation coefficient, sometimes also referred to as *population correlation coefficient* or *sample correlation*, is dimensionless, and can take any value from -1 to $+1$. The simplest way to calculate

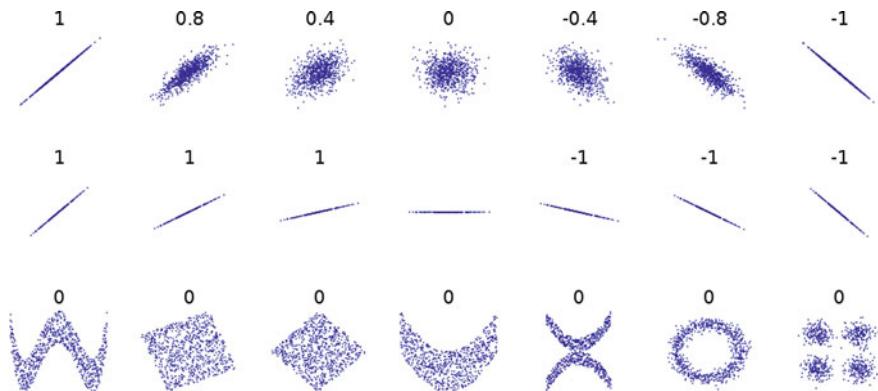


Fig. 11.4 Several sets of (x, y) points, with the correlation coefficient of x and y for each set. Note that the correlation reflects the non-linearity and direction of a linear relationship (top row), but not the slope of that relationship (middle), nor many aspects of nonlinear relationships (bottom). N.B.: the figure in the center has a slope of 0 but in that case the correlation coefficient is undefined because the variance of Y is zero. (In Wikipedia. Retrieved Aug 21, 2021, from http://en.wikipedia.org/wiki/Correlation_and_dependence)

Pearson's correlation coefficients is with `stats.pearsonr` or with `pg.corr`. The `pingouin` function has the advantage that not only the correlation coefficient is calculated but also the confidence interval:

```
result = pg.corr(iris.petal_length, iris.petal_width)
print(result.round(3))
#      n      r      CI95%   p-val      BF10    power
# pearson 150  0.963  [0.95, 0.97]  0.0  1.113e+82  1.0
pearson_r = results.r
```

For the data shown in Fig. 11.3 this indicates a correlation coefficient of $r = 0.96$. Other examples are given in Figure 11.4. Note that the formula for the correlation coefficient is symmetrical between x and y —which is not the case for linear regression!

11.2.3 Rank Correlation

If the data distribution is not normal a different approach is necessary. In that case one can rank the set of data for each variable and compare the orderings. There are two commonly used methods of calculating the rank correlation.

Spearman's ρ is exactly the same as the Pearson correlation coefficient r , but calculated on the ranks of the observations and not on the original numbers.

Kendall's τ is also a rank correlation coefficient, measuring the association between two measured quantities. It is harder to calculate than Spearman's ρ ,

but it has been argued that confidence intervals for Spearman's ρ are less reliable and less interpretable than confidence intervals for Kendall's τ -parameters.

These correlation coefficients, as well as others, can be obtained through adjusting the `methods` parameter for `pg.corr`:

```
spearman_rho = pg.corr(x, y, method='spearman')
kendall_tau = pg.corr(x, y, method='kendall')
```

 **Code:** `ISP_bivariate.py`² Analysis of multivariate data (regression, correlation).

11.3 Coefficient of Determination

The correlation coefficient is closely related to the slope in linear regression. So for the interpretation of its square r^2 , often called the *coefficient of determination*, we make a detour into linear fits. (Details on linear fits will be covered in Chap. 12).

11.3.1 General Linear Regression Model

We can use the method of *linear regression* when we want to predict the value of one variable from the value(s) of one or more other variables. For example, when we search for the best-fit line to a given data set (x_i, y_i) , we are looking for the parameters (m, b) which minimize the sum of the squared residuals ϵ_i in

$$y_i = m * x_i + b + \epsilon_i \quad (11.4)$$

where m (“multiplier”) is the *slope* or *inclination* of the line, and b (“bias”) the *intercept*. ϵ_i are the *residuals*, i.e., the differences between observed values and predicted values (see Fig. 11.5). This is in fact just the one-dimensional example of a more general technique, described in the next chapter.

Since the linear regression equation is solved to minimize the square sum of the residuals, linear regression is sometimes also called *Ordinary Least-Squares (OLS) regression*.

Writing out the equations that determine m and b leads to

$$m = r * \frac{s_y}{s_x} \quad (11.5)$$

² [ISP2e>/12_LinearModels/bivariate/ISP_bivariate.py](#).

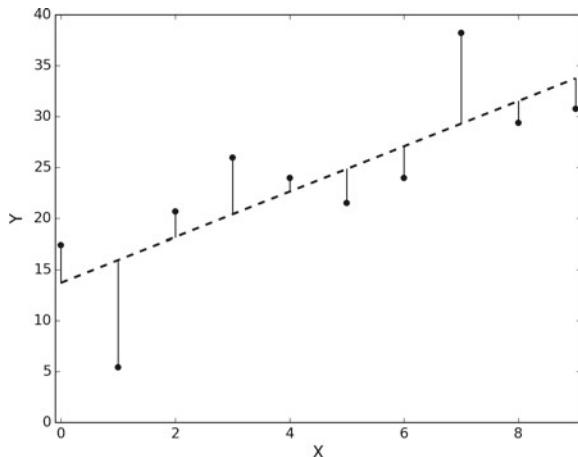


Fig. 11.5 Best-fit linear regression line (dashed line) and residuals (solid lines). Note that in contrast to the correlation, this relationship between x and y is not symmetrical anymore: it is assumed that the x -values are known exactly, and that all the variability lies in the residuals

where s_x and s_y are the sample standard deviations in the x - and y -direction, respectively, showing the close relationship between the slope m of a linear fit and the correlation coefficient r .

11.3.2 Interpretation

A data set has values y_i , each of which has an associated modeled value \hat{y}_i . Here, the values y_i are called the *observed values*, and the modeled values \hat{y}_i are sometimes called the *predicted values*.

In the following \bar{y} denotes the mean of the observed data:

$$\bar{y} = \frac{1}{n} \sum_{i=1}^n y_i \quad (11.6)$$

where n is the number of observations.

The “variability” of the data set is measured through different sums of squares:

- $SS_{\text{mod}} = \sum_{i=1}^n (\hat{y}_i - \bar{y})^2$ is the *Model Sum of Squares*, or the sum of squares for the regression. This value is sometimes also called the *Explained Sum of Squares*.
- $SS_{\text{res}} = \sum_{i=1}^n (y_i - \hat{y}_i)^2$ is the *Residuals Sum of Squares*, or the sum of squares for the errors.
- $SS_{\text{tot}} = \sum_{i=1}^n (y_i - \bar{y})^2$ is the *Total Sum of Squares*, and is equivalent to the sample variance multiplied by $n - 1$.

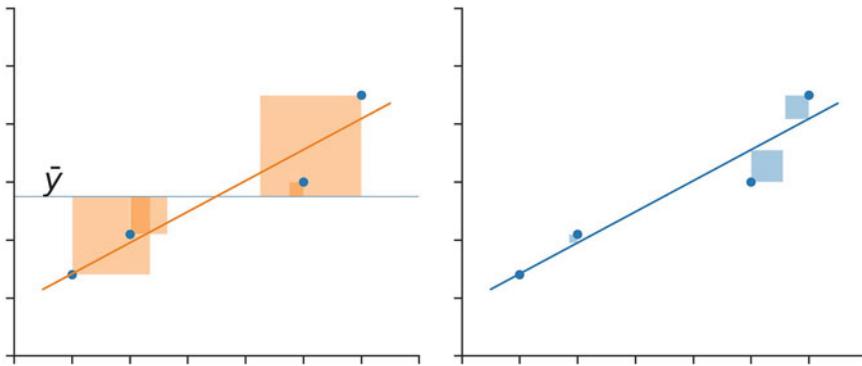


Fig. 11.6 The better the linear regression (on the right) fits the data in comparison to the simple average (on the left graph), the closer the value of R^2 is to one. The areas of the blue squares represent the squared residuals with respect to the linear regression. The areas of the orange squares represent the squared residuals with respect to the average value

For multiple regression models,

$$SS_{\text{mod}} + SS_{\text{res}} = SS_{\text{tot}}. \quad (11.7)$$

The notations SS_R and SS_E should be avoided, since “R” can stand for either “regression” or “residual”, and “E” for either “error” or “explained”.

With these expressions, the most general definition of the *coefficient of determination*, R^2 , is

$$R^2 \equiv 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}}. \quad (11.8)$$

Since

$$SS_{\text{tot}} = SS_{\text{mod}} + SS_{\text{res}}. \quad (11.9)$$

Equation 11.8 is equivalent to

$$R^2 = \frac{SS_{\text{mod}}}{SS_{\text{tot}}}. \quad (11.10)$$

In words: The coefficient of determination is the ratio between the sum-of-squares explained by the model, and the total sum-of-squares ([sum of blue squares] / [sum of red squares] in Fig. 11.6). In a general form R^2 can be seen to be related to the unexplained variance, since the second term in Eq. 11.8 compares the unexplained variance (variance of the model’s errors) with the total variance (of the data).

For simple linear regression (i.e., line-fits), the coefficient of determination or R^2 is the square of the correlation coefficient r . It is easier to interpret than the correlation coefficient r : values of R^2 close to 1 correspond to a close correlation, values close to 0 to a poor one. Note that for general models it is common to write R^2 , whereas for simple linear regression r^2 is used.

How large R^2 values must be to be considered as “good” depends on the discipline. They are usually expected to be larger in the physical sciences than in biology or the social sciences. In finance or marketing, it also depends on what is being modeled.

Caution: the sample correlation and R^2 are misleading if there is a nonlinear relationship between the independent and dependent variables (see Fig. 11.4)!

11.4 Scatterplot Matrix

If we have three to six variables that may be related to each other, we can use a *scatterplot matrix* to visualize the correlations between the different variables (Fig. 11.7). The off-diagonal elements of the scatterplot matrix consist of the correlation plots, and the diagonal elements are the histograms or KDE-plots of the corresponding variables.

```
import seaborn as sns
sns.set()

df = sns.load_dataset("iris")
sns.pairplot(df, hue="species", size=2.5)
```

11.5 Correlation Matrix

When moving from two to three or more variables $x_i, i = 1, \dots, n$, the correlation coefficient gets replaced by the *correlation matrix*. Thereby the element r_{ij} indicates the correlation coefficient between the variables x_i and x_j . And if we want to and *predict* the value of more than one other variables, linear regression has to be replaced by *multilinear regression*, sometimes also referred to as *multiple linear regression*.

However, many pitfalls loom when working with many variables! Consider the following example: golf tends to be played by richer people; and it is also known that on average the number of children goes down with increasing income. In other words, we have a fairly strong negative correlation between playing golf and the number of children, and one could be tempted to (falsely) draw the conclusion that playing golf reduces the fertility. But in reality it is the higher income which causes both effects. Kaplan (2009) nicely describes where those problems come from, and how best to avoid them.

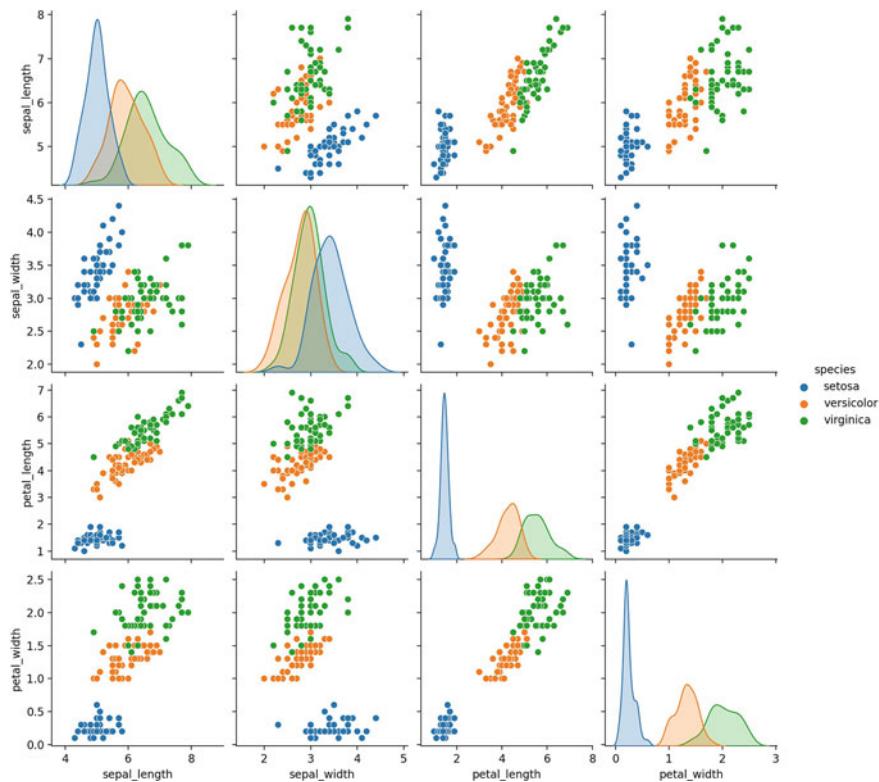


Fig. 11.7 Scatterplot matrix

Visualization

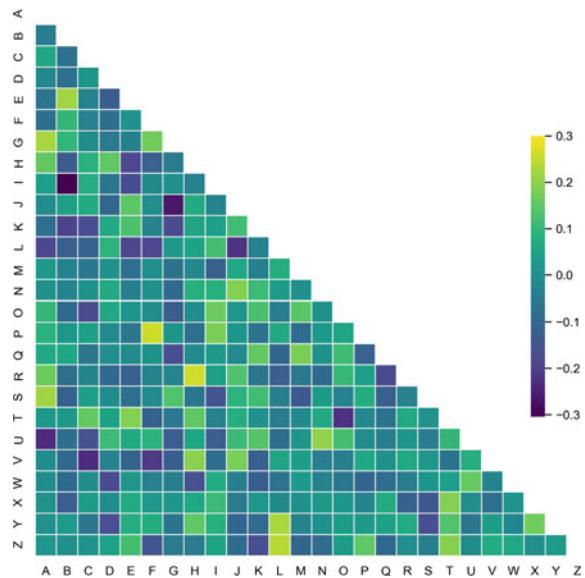
An elegant way to visualize the correlation between a large number of variables is the *correlation matrix*. Using *seaborn*, the following example shows how to implement a correlation matrix. In the example, the parameter for `np.random.RandomState` is the seed for the random number generation. The data are normally distributed dummy data, simulating 100 recordings from each of 26 different variables. The listing below calculates and visualizes the cross-correlation between each possible combination of variables (Fig.11.8):

Listing 11.1: corr_matrix.py

```
""" Plotting a diagonal correlation matrix
```

```
With permission from Michael Waskom, from
http://seaborn.pydata.org/examples/many_pairwise_correlations
.html
"""
```

Fig. 11.8 Visualization of the Correlation matrix



```

from string import ascii_letters
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

sns.set_theme(style="white")

# Generate a large random dataset
# The syntax here is slightly different from the previously
# used np.random.seed. For details, see
# https://stackoverflow.com/questions/5836335/consistently-
# create-same-random-numpy-array
rs = np.random.RandomState(1234)
df = pd.DataFrame(data=rs.normal(size=(100, 26)),
                   columns=list(ascii_letters[26:]))

# Compute the correlation matrix
corr = df.corr()

# Generate a mask for the upper triangle
mask = np.triu(np.ones_like(corr, dtype=bool))

# Set up the matplotlib figure
f, ax = plt.subplots(figsize=(11, 9))

# Generate a custom colormap
cmap = sns.color_palette("viridis", as_cmap=True)

```

```

# Draw the heatmap with the mask and correct aspect ratio
sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.3, center=0,
    square=True, linewidths=.5, cbar_kws={"shrink": .5})
out_file = 'many_pairwise_correlations.jpg'
plt.savefig(out_file, dpi=300)
print(f'Correlation-matrix saved to {out_file}')

plt.show()

```

11.6 Autocorrelation

To find repeating, unknown patterns within a signal, the signal can be compared to shifted versions of itself, leading to the so-called “autocorrelation”. In statistics and time-series analysis the *autocorrelation coefficients* are commonly defined as the “autocovariance coefficients”, normalized by the variance.³ Thereby the *autocovariance coefficient at lag k* is given by the cross correlation between a signal and a copy of itself that is shifted by k points, where the mean value has been subtracted. This may sound more daunting than it actually is: in practice, the autocorrelation coefficients are calculated by computing the series of sample auto-covariance coefficients at lag k as

$$c_k = \frac{1}{N} \sum_{t=1}^{N-k} (x_t - \bar{x})(x_{t+k} - \bar{x}) . \quad (11.11)$$

The autocorrelation coefficients are then given by

$$r_k = c_k/c_0 . \quad (11.12)$$

While the cross correlation described in Sect. 11.1 is commonly used to find the occurrence of a shorter feature in a longer signal, the autocorrelation function is commonly used to detect repetitive features in longer signals. An example will be presented in the next section.

³ In contrast, in signal processing and engineering the autocorrelation is simply the cross correlation of a function with itself.

11.7 Time-Series Analysis

A powerful tool for the in-depth analysis of time-stamped data, and a good example for the application of the autocorrelation function, is *time series analysis*. But since not everybody needs this tool, this section can be skipped during a first reading, as it relates to an advanced aspect of data analysis.

In the stock market and with the weather, among many other things, timing and the sequence of events are very important. So information about everything that is happening there is stored with a time-stamp. And not just there: in our sensor- and data-driven world, almost everything is nowadays recorded and stored digitally. *Time-series analysis (TSA)* is the science that tries to extract information out of such data recorded as a function of time. Thereby it pursues four goals:

1. The concise description of the recorded signals,
2. the explanation of the observed patterns,
3. the prediction of future events, and
4. the control of the systems under observation.

With applications in so many disciplines, it is not too surprising that TSA is a field that goes way beyond the scope of this book. For an in-depth introduction to time series the classic text by Chapman.

Chatfield and King (2019) is a very recommendable starting point.

TSA mainly concentrates on data recorded with fixed time intervals. Methods for the analysis of such data may be divided into two classes: frequency-domain methods and time-domain methods. The former includes spectral analysis and wavelet analysis; the latter includes autocorrelation and cross correlation analysis. A good introduction to the frequency domain methods is Smith (2007). But since the frequency domain analysis is of more relevance in signal processing than in statistics, these methods won't be covered here.

This section will present a simple example of a time-series analysis, to demonstrate some of the basic principles of the time domain methods for the description of time series. For the Python implementations, we will use the `tsa` module of the package `statsmodels`. Other important areas of time-series analysis, such as forecasting, will not be covered here. For an introduction to forecasting, Hyndman and Athanasopoulos (2018) is a recommendable text that is also available online.

Since global warming is arguably the most pressing challenge that mankind currently faces, the example below will use data on the trends in atmospheric carbon dioxide from the *National Oceanic and Atmospheric Administration (NOAA, Fig. 11.9)*. In order to be representative of the state of the global atmosphere, these data were recorded on a remote mountain top on an island in the Pacific, on Mauna Loa in Hawaii. They constitute the longest available record of direct measurements of CO_2 in the atmosphere, and show how massively we have already changed the air on our planet.

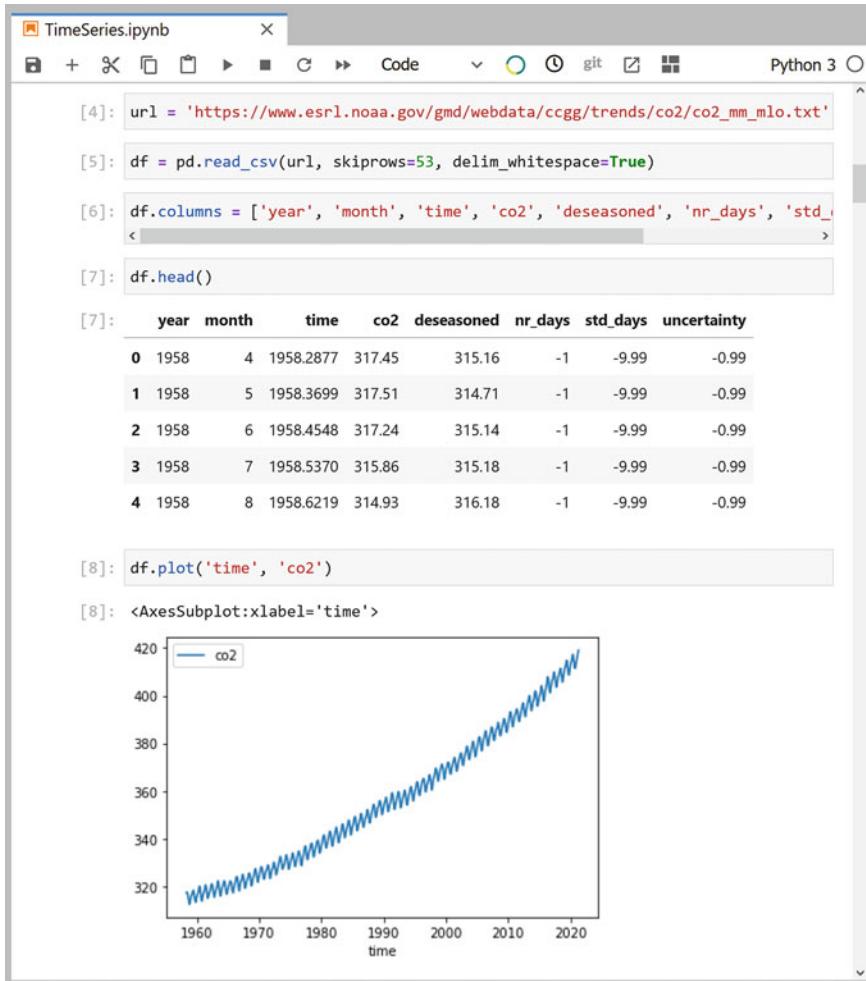


Fig. 11.9 Reading in the data for the TSA-example, from the Jupyter notebook 11_timeSeries.ipynb in the github archive of this book. Note that the pandas command `pd.read_csv` can directly import data from the web

11.7.1 Data Decomposition

Most models that are used to explain observed data features assume that the process is *stationary*. So the first step in the data handling is to decompose the data into the components Trend, Seasonal, and Residuals. That decomposition can be *additive*

$$\text{data} = \text{Trend} + \text{Seasonal} + \text{Residuals} \quad (11.13)$$

or it can be *multiplicative*

$$\text{data} = \text{Trend} * \text{Seasonal} * \text{Residuals} \quad (11.14)$$

The example will use the following commands from *statsmodels*, which will be described as we use them:

```
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import acf, pacf
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.arima.model import ARIMA
```

The command `seasonal_decompose` performs the decomposition indicated in Eq. 11.13 using a rather simple algorithm (Fig. 11.10).

```
result_add = seasonal_decompose(df['co2'], model='additive',
                                 period=12, extrapolate_trend='freq')
result_add.plot()
plt.show()
```

We specify here that we use an additive model (instead of a multiplicative one), and that the Seasonal component has a duration of 12 time-steps (the data used here have been recorded monthly). The parameter `extrapolate_trend='freq'` ensures that there are no NaN values in the Trend or the Residuals.

The resulting Fig. 11.10 deserves a few comments. The overall CO_2 level in the atmosphere (top plot) is the sum of the Trend, the Seasonal component, and

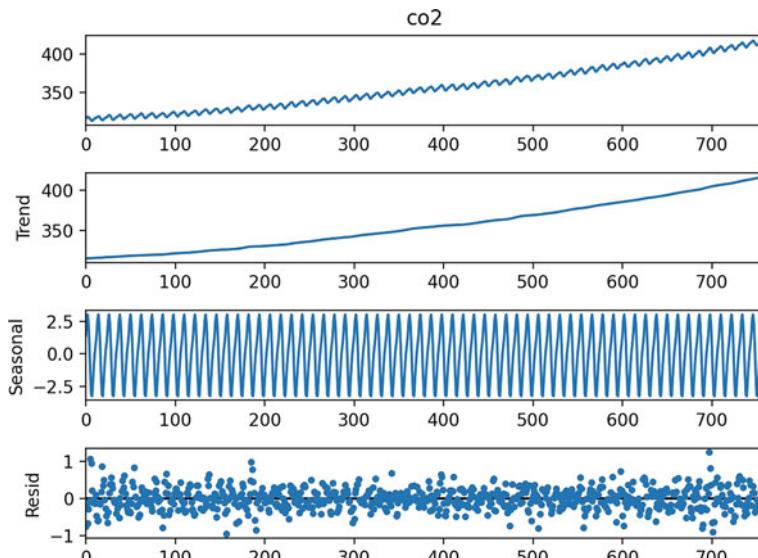


Fig. 11.10 The x-axis indicates the number of the month recorded. The CO_2 level is expressed in *ppm*, i.e., “parts per million” of the molecules in the air

the Residuals. From a climate perspective, the trend is the scary part. Before the industrial revolution that value was pretty much constant, around 280 ppm—now it is above 420 ppm. The last time the earth experienced similar CO_2 values was approximately 20 million years ago! The Seasonal component is by definition assumed to stay constant—here we see that the CO_2 level varies with an annual cycle. Any additional variation is moved into the Residuals, which also contains all the other remaining information. Note that the y-scales on the three plots in this figure are different!

The next step is to see which additional information can be extracted from the residuals.



Code: `ISP_TimeSeries.py`⁴ Time series analysis of global CO_2 -data from Mauna Loa (Hawaii).

11.7.2 Analysis of Residuals

a) Autocorrelation

The autocorrelation can now be used to find unknown systematic patterns in the Residuals. However, to quote Chatfield has said “Considerable experience is required to interpret sample autocorrelation coefficients. In addition it is necessary to study the probability theory of stationary series and learn about the classes of models that may be appropriate.” (Chatfield and Xing 2019).

The autocorrelation function (ACF) can be visualized with

```
plot_acf(result_add.resid)
```

which for the current example leads to (Fig. 11.11).

The remaining oscillations with a duration of 12 time-steps indicate that some seasonal effects still remain in the Residuals. We won’t dig into the cause of these remaining seasonal effects, which here are caused by a change in amplitude over time (which can be shown by a more in-depth visual inspection of the residuals). Instead, it is helpful to see how information from the ACF may be further analyzed in TSA models.

Once the main trend and seasonal components have been eliminated, models are used to see how the remaining features in the data can best be explained. The next section introduces the most common models that are used to interpret the residual data. But before we introduce the models, we need one more tool for the subsequent model selection: the “partial autocorrelation function”.

b) Partial Autocorrelation

One point to notice in Fig. 11.11 is that the (remaining) cyclical patterns in the signal induce repeated peaks in the ACF, since cyclical signal-shifts by one, two, or more

⁴ [<ISP2e>/11_Pattern/timeSeriesAnalysis/ISP_TimeSeries.py](https://ISP2e/11_Pattern/timeSeriesAnalysis/ISP_TimeSeries.py).

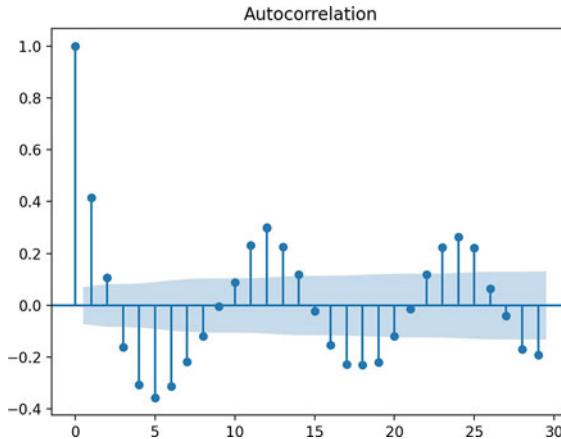


Fig. 11.11 The systematic patterns in the autocorrelation function show that some seasonal components are not constant and have been missed. The blue shaded area indicates the 95%-confidence interval for the autocorrelation coefficients

cycles lead to matching signal patterns. To remove that “redundancy”, the *partial autocorrelation function (PACF)* for a lag k is defined as the remaining correlation that is not explained by smaller lags. The correlation function and the partial correlation function are used in the assessment of autoregressive models, as will be explained below. (Details can be found in Shumway and Stoffer, 2017).

To formally define the PACF for mean-zero stationary time series, let \hat{x}_{t+h} , for $h \geq 2$, denote the regression of x_{t+h} on $x_{t+h-1}, x_{t+h-2}, \dots, x_{t+1}$, which we write as

$$\hat{x}_{t+h} = \beta_1 x_{t+h-1} + \beta_2 x_{t+h-2} + \cdots + \beta_{h-1} x_{t+1}. \quad (11.15)$$

In addition, let \hat{x}_t denote the regression of x_t on $x_{t+1}, x_{t+2}, \dots, x_{t+h-1}$ with

$$\hat{x}_t = \beta_1 x_{t+1} + \beta_2 x_{t+2} + \cdots + \beta_{h-1} x_{t+h-1}. \quad (11.16)$$

Because of stationarity, the coefficients, $\beta_1, \dots, \beta_{h-1}$ are the same in Eqs. 11.15 and 11.16. The *partial autocorrelation function (PACF)* of a stationary process x_t is then defined as

$$pacf(1) = \text{corr}(x_{t+1}, x_t) \quad (11.17)$$

$$pacf(h) = \text{corr}(x_{t+h} - \hat{x}_{t+h}, x_t - \hat{x}_t), \quad h \geq 2 \quad (11.18)$$

where `corr` is the correlation coefficient. Or in words: $pacf(h)$ is the correlation between x_{t+h} and x_t , with the linear dependence of $x_{t+1}, \dots, x_{t+h-1}$ on each removed.

11.7.3 ARMA models

The most common models used to explain time series are *autoregressive integrated moving average* models, or in short *ARIMA* models. The name is more fearsome than the ideas behind it. Since the mathematical details behind these models are quite intricate, this section is *not* a mathematically rigid analysis of ARIMA models. Instead, it simply tries to convey the main ideas behind them.

a) Assumptions of ARMA models

Autoregressive moving average (ARMA) models make the assumption that the data come from a *stationary process*. Roughly speaking a time series is said to be “stationary” if there is no systematic change in the mean and/or in the variance, and if strictly periodic variations have been removed. The second assumption is that the process is driven by random external events that have zero mean and are normally distributed. In the following let Z_t denote the random external inputs that are the assumed drivers of the observed processes, which are assumed to have a variance of σ_Z^2 .

b) Moving Average Processes (MA)

A process X_t is said to be a “moving average process of order q ”, short an *MA(q) process*, if

$$X_t = \beta_0 Z_t + \beta_1 Z_{t-1} + \cdots + \beta_q Z_{t-q} \quad (11.19)$$

where the MA-coefficients β_i are constants. The Z_i are usually scaled so that $\beta_0 = 1$. Expressed in words this means that an event at time t_i modifies the signal at a later time t_{i+k} by β_k , for $k = 1, 2, \dots, q$. No more effects of this event will be felt after t_{i+q} .

c) Autoregressive Processes (AR)

A process X_t is said to be an “autoregressive process of order p ”, short an *AR(p) process*, if

$$X_t = \alpha_1 X_{t-1} + \cdots + \alpha_p X_{t-p} + Z_t. \quad (11.20)$$

Or in words: an event at time t_i is remembered at a later time t_{i+k} with a strength determined by the AR-coefficients α_k , for $k = 1, 2, \dots, p$. No more memory effects of this event will be felt after t_{i+p} . Since this is like a multiple regression model, but with X_t regressed on past values of X_t rather than on separate predictor variables, this is referred to as an “autoregressive” process.

d) ARMA Models

Autoregressive processes and moving average processes can be combined, leading to so-called ARMA processes.

11.7.4 Integrated ARMA (or ARIMA) Models

In practice, most time series are non-stationary. However, non-stationary sources of variation can often be removed by simple or multiple differentiation. Such a model is called an *integrated model* (ARIMA) because the stationary model that is fitted to the differenced data has to be summed or “integrated” to provide a model for the original non-stationary data.

For indicating the differentiation, the following notation will be used:

B is the “time-shift operator”

$$BX_t = X_{t-1} \quad (11.21)$$

$$B^2X = X_{t-2} \quad (11.22)$$

$$etc. \quad (11.23)$$

With this, a d -times differentiation of X_t can be written as

$$W_t = \Delta^d X_t = (1 - B)^d X_t, \quad d = 0, 1, 2, \dots \quad (11.24)$$

Using this notation, the general “autoregressive integrated moving average” (ARIMA) process is of the form

$$W_t = \alpha_1 W_{t-1} + \dots + \alpha_p W_{t-p} + Z_t + \beta_1 Z_{t-1} + \dots + \beta_q Z_{t-q}. \quad (11.25)$$

or (with proper adjustment of the sign of the α_i -s)

$$\sum_{i=0}^p \alpha_i W_{t-i} = \sum_{j=0}^q \beta_j Z_{t-j} \quad (11.26)$$

where $\alpha_0 = \beta_0 = 1$. The differentiated time series W_t forms an $ARMA(p, q)$ process, and since W_t is obtained from X_t through d -times differentiation, this is commonly referred to as an *ARIMA process of the order (p,d,q)* .

11.7.5 Examples of Simple ARIMA Models

a) Identifying AR and MA Models

Figure 11.12 shows the two $ARMA(1, 0)$ moving average models

$$X_t = 1 * Z_t \pm 0.9 * Z_{t-1}. \quad (11.27)$$

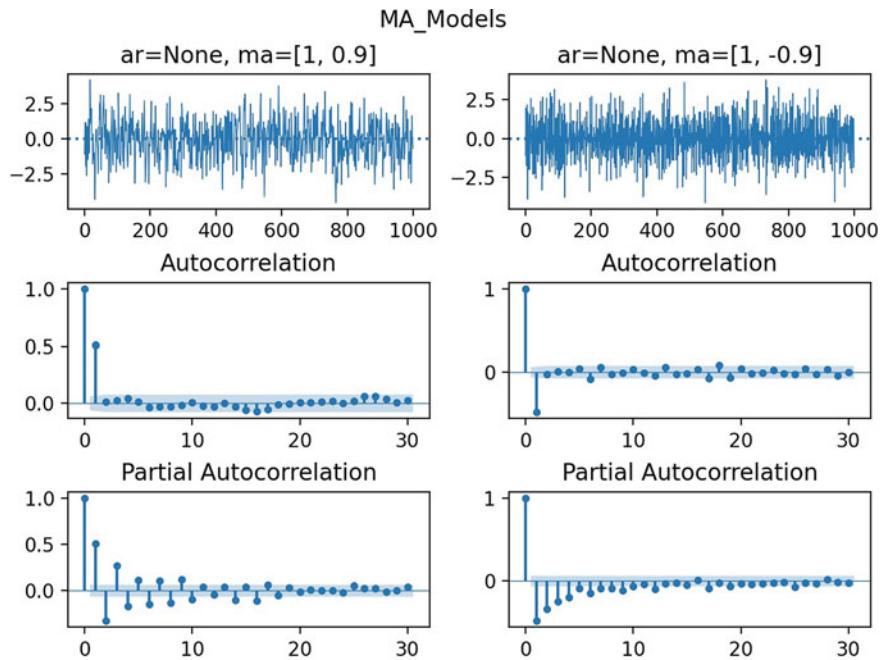


Fig. 11.12 MA models (top), with their corresponding ACF (middle) and PACF (bottom). The process in the left column corresponds to $X_t = Z_t + 0.9 * Z_{t-1}$. Note in the middle row that in this model, which has only two MA-coefficients, only the first two values of the ACF are significant. The blue shaded areas again indicate the 95% confidence interval

These models correspond to a moving average process of order one (or “MA(1)-process”), and the corresponding MA-coefficients are $ma=[1, \pm 0.9]$.

While the MA-model on the left, $ma = [1, 0.9]$, is effectively a low-pass filter averaging over two random numbers, the MA-model on the right, $ma = [1, -0.9]$, effectively differentiates two adjacent random numbers. The second row shows the corresponding ACF. Note that for an MA(1) process, only the first two values of the ACF are significant!

The next figure, Fig. 11.13, shows two ARMA(0,1) autoregressive models

$$X_t = \pm 0.9 * X_{t-1} + Z_t. \quad (11.28)$$

The coefficients corresponding to these AR(1) processes are $ar=[1, \mp 0.9]$.

While the process in the left column essentially jumps between positive and negative values, the process on the right corresponds to a “relaxed random walk”. (This can be seen only roughly in Fig. 11.13, and better when running F11_TimeSeries.py from the ISP-archive and zooming in on the top-most time plots.) Compared to the MA processes above, the roles of the ACF and the PACF are now reversed: for AR(1) processes, only the first two values of the PACF are

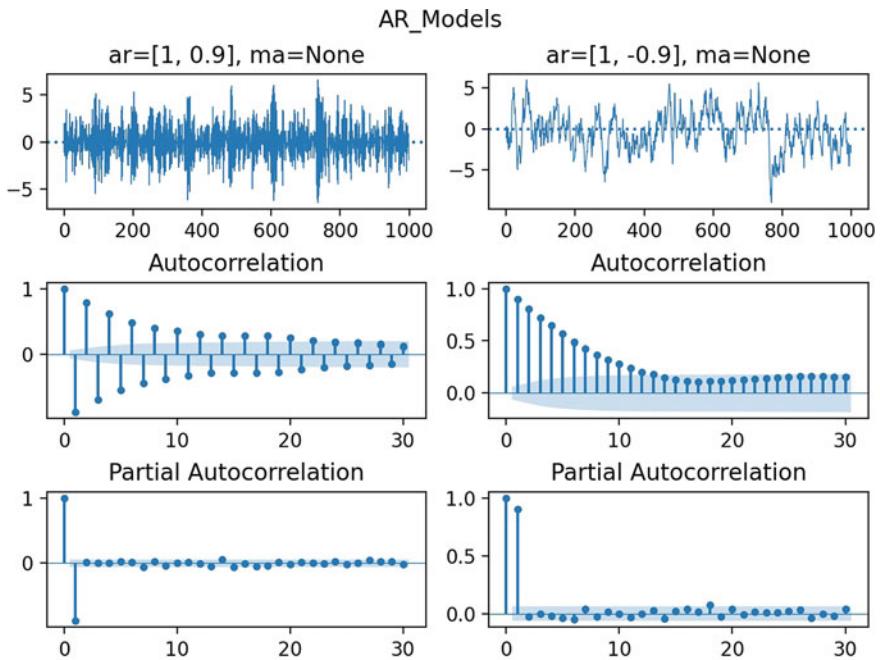


Fig. 11.13 AR models (top), with their corresponding ACF (middle) and PACF (bottom). The process in the right column corresponds to $X_t = 0.9 * X_{t-1} + Z_t$. Note in the bottom row that in this model, which has only two AR-coefficients, only the first two values of the PACF are significant

Table 11.2 Behavior of the ACF and the PACF for ARMA models

	AR(p)	MA(q)	ARMA(p, q)
ACF	Tails off	Cuts off after lag q	Tails off
PACF	Cuts off after lag p	Tails off	Tails off

significant; in contrast, the ACF shows a slowly decaying behavior, which does not help to clearly identify the underlying process.

Table 11.2 summarizes these findings, and shows how to determine the order of the underlying process for pure AR and MA processes.

b) Fitting ARMA-Models

Once the order or an ARIMA model is determined (or in practice: estimated), the corresponding model coefficients can be found easily. For example, to generate the AR(1) process $X_t = 0.9 * X_{t-1} + Z_t$ (shown in the right column of Fig. 11.13), and to find the corresponding best-fit AR coefficients one can use

```

from statsmodels import tsa

# Generate the data
ar = [1, -0.9]
ma = None
n_samples = 100
np.random.seed(123)      # to make it reproducible

arma_process = tsa.arima_process.ArmaProcess(ar, ma)
y = arma_process.generate_sample(n_samples)

# Fit the model
model = tsa.arima.model.ARIMA(y, order=(1, 0, 0))
model_fit = model.fit()

print(f'ARMA = {(ar, ma)}')
print(model_fit.summary())

```

This produces

```

ARMA = ([1, -0.9], None)
                SARIMAX Results
=====
Dep. Variable:                      y    No. Observations:          100
Model:                 ARIMA(1, 0, 0)   Log Likelihood:      -154.409
Date:                  Fri, 28 May 2021   AIC:                   314.818
Time:                          12:06:34   BIC:                   322.633
Sample:                           0   HQIC:                  317.981
                                    - 100
Covariance Type:                  opg
=====
            coef    std err        z     P>|z|      [0.025      0.975]
-----
const    0.1630     1.013     0.161     0.872     -1.823     2.149
ar.L1    0.8907     0.048    18.564     0.000      0.797     0.985
sigma2   1.2643     0.216     5.866     0.000      0.842     1.687
=====
Ljung-Box (L1) (Q):                  0.04   Jarque-Bera (JB):       1.62
Prob(Q):                           0.84   Prob(JB):             0.45
Heteroskedasticity (H):               0.70   Skew:                  0.03
Prob(H) (two-sided):                0.31   Kurtosis:              2.38
=====
```

The top box of the SARIMAX (“Seasonal AutoRegressive Integrated Moving Average exogenous model”) Results contains parameters characterizing the model, which will be described in the next chapter (Sect. 12.4). For example, the *Akaike Information Criterion (AIC)* can be used to compare different models: the best model will typically be the one with the lowest AIC value.

The middle section of SARIMAX Results provides the best-fit parameters, as well as information about their significance. `const` is a non-significant offset, `ar.L1` the auto-regressive coefficient at lag 1, and `sigma2` (“sigma squared”) represents the variance of the residual values. This value is used to test the normality of residuals against the alternative of non-normality.

And the bottom section contains information about the distribution of the remaining model residuals.

Care should be taken with the following points:

- By convention, the coefficients α_0 and β_0 in Eq. 11.26 are taken to be 1.
- Also due to the definition in Eq. 11.26, the signs of the AR-coefficients are inverted. This means that the result here, `ar.L1=0.9`, corresponds to the AR(1) model with `ar=[1, -0.9]`.
- Due to the relatively small number of samples, an (insignificant) offset appears in the fitted ARMA parameters.
- For combined ARMA models, the order of the model can be determined by fitting ARMA models of different orders, and using, e.g., the AIC to identify the best fitting model order.
- Accurately estimating the correct order of the model, as well as the number of differentiations (the “I” in the “ARIMA”) that are necessary to obtain a “stationary” process, requires a substantial amount of experience! For details, please check out the literature specializing on TSA, such as Chatfield and Xing (2019) or Shumway and Stoffer (2017).

Chapter 12

Linear Regression Models



There is a substantial difference in approach between hypothesis tests and statistical modeling. With hypothesis tests, one typically starts out with a null hypothesis. Based on the question and the data, one then selects the appropriate statistical test as well as the desired significance level, and either accepts or rejects the null hypothesis.

In contrast, statistical modeling typically involves a more interactive analysis of the data. One starts out with a visual inspection of the data, looking for correlations and/or relationships. Based on this first inspection, a statistical model is selected that may describe the data. In simple cases, the relationship in the data can be described with a linear model model

$$y = m * x + b.$$

Next

- the model parameters (here, e.g., m for “multiplier” and b for “bias”) are determined,
- the quality of the model is assessed,
- and the residuals (i.e., the remaining errors) are inspected, to check if the proposed model has missed essential features in the data.

If the residuals are too large, or if the visual inspection of the residuals shows outliers or suggests another model, the model is modified. This procedure is repeated until the results are satisfactory.

This chapter describes how to implement and solve linear regression models in Python, and how to quantify the correlation between sets of data. The resulting model parameters are discussed, as well as the assumptions of the models and interpretations of the model results. Since *bootstrapping* can be helpful in the evaluation of some models, the final section in this chapter shows a Python implementation of a bootstrapping example.

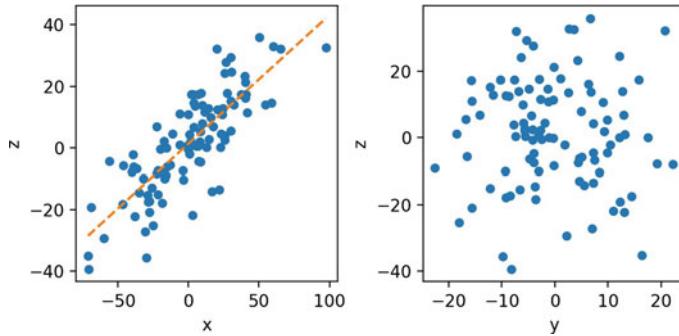


Fig. 12.1 x/z (“side view”) and y/z (“front view”) plots of the sample data. In the left plot also the best-fit line to the data is indicated

12.1 Simple Fits

To get comfortable with linear regressions, this section starts with a noisy data set where we know the true parameters, a noisy plane in 3D (Figs. 12.1 and 12.2):

```
# Generate some data
np.random.seed(12345)
x = np.random.randn(100)*30
y = np.random.randn(100)*10
z = 3 + 0.4*x + 0.05*y + 10*np.random.randn(len(x))

# Put them into a DataFrame
df = pd.DataFrame({'x':x, 'y':y, 'z':z})

# Show them
fig, axs = plt.subplots(1,2)
df.plot('x', 'z', kind='scatter', ax=axs[0])
df.plot('z', 'y', kind='scatter', ax=axs[1])
```

Since visual inspection of Fig. 12.1 (left) shows a clear linear correlation, we can fit a line to the x/z data with

```
# Simple linear regression
results = pg.linear_regression(df.x, df.z)

print(results.round(2))
#      names  coef      se       T  pval      r2  adj_r2    CI[2.5%]  CI[97.5%]
# 0  Intercept  1.27  0.96  1.33  0.19  0.65     0.65   -0.63      3.18
# 1        x  0.42  0.03 13.59  0.00  0.65     0.65     0.36      0.48

# show how to access individual parameters
print(f'p = {results.pval.values[1]:.1e}')
# p = 2.9e-24
```

The output, which is listed in the comments to the code segment above, can be interpreted as follows:

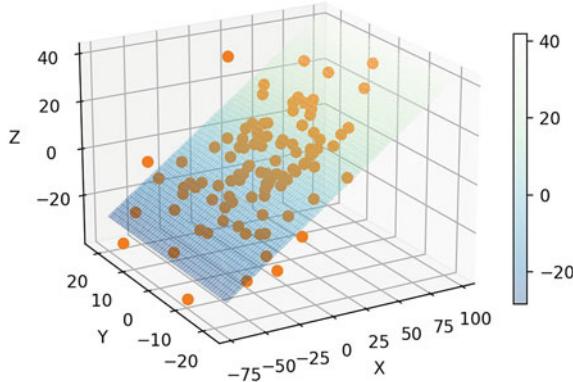


Fig. 12.2 While 3D plots can be generated with *matplotlib*, I recommend to use them sparingly: in most cases quantitative relationships can be visualized much more clearly with 2D projections

- The best-fit line to the data is $z = 0.42 * x + 1.27$.
- The *intercept* is *not significant* because the corresponding *pval* (i.e., the probability that the intercept is equal to zero) is >0.05 . This can also be seen in the fact that the corresponding 95% confidence interval (95%-CI = [-0.63, 3.18]) overlaps *zero*.
- The line is *rising significantly*, since both ends of the 95% confidence interval for “x” are larger than 0 (95%-CI = [0.36, 0.48]). (If both would be below zero, the line would be called *significantly falling*; and if the 95%-CI would overlap zero, the line would have *no significant slope*). Again, this is reflected in $p <0.05$. (Since $p <0.001$, the line is rising even *highly significantly*.)

Since in our case the *x*- and the *y*-variable are completely independent from each other (we know that here both are just random numbers; in general, this has to be checked!), we can also use `pg.linear_regression` to compute the multilinear regression of *z* on *x* and *y* (Fig. 12.2):

```
# Multiple linear regression
results = pg.linear_regression(df[['x', 'y']], df['z'])
print(results.round(2))

#      names   coef     se      T  pval    r2  adj_r2  CI[2.5%]  CI[97.5%]
# 0  Intercept  1.28  0.96  1.33  0.19  0.65    0.65   -0.63      3.19
# 1        x   0.42  0.03 13.52  0.00  0.65    0.65    0.36      0.48
# 2        y   0.02  0.10  0.20  0.84  0.65    0.65   -0.18      0.22
```

The resulting output shows that there is *no significant dependence of z on y*, since (a) the corresponding p-value is much larger than 0.05 (*pval* = 0.84), and (b) the corresponding 95%-CI = [-0.18, 0.22] overlaps zero.

Note that `pg.linear_regression` can be fine-tuned with a number of powerful options, allowing, e.g., the calculation of 99.9%-CIs, weighted linear regression, etc.

To unleash the full power of statistical modeling, the following sections provide some background basics for linear regression analysis.



Code: `ISP_linearRegression.py`¹ The full source code for this section.

12.2 Design Matrix and Formulas

12.2.1 Example 1: Simple Linear Regression

Suppose there are 7 data points $\{y_i, x_i\}$, where $i = 1, 2, \dots, 7$. The simple linear regression model is

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i, \quad (12.1)$$

where β_0 is the y-intercept and β_1 is the slope of the regression line. This model can be represented in matrix form as

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \\ 1 & x_4 \\ 1 & x_5 \\ 1 & x_6 \\ 1 & x_7 \end{bmatrix} \cdot \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \\ \epsilon_6 \\ \epsilon_7 \end{bmatrix} \quad (12.2)$$

where the first column of ones in the matrix on the right-hand side represents the y-intercept term, while the second column contains the x-values. This matrix is called “design matrix”. (Section 12.3 shows how to solve these equations for β_i with Python.)

12.2.2 Example 2: Quadratic Fit

The equation for a quadratic fit to the given data is

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \epsilon_i, \quad (12.3)$$

This can be rewritten in matrix form:

¹ ISP2e/12_LinearModels/linearRegression/ISP_linearRegression.py.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ 1 & x_4 & x_4^2 \\ 1 & x_5 & x_5^2 \\ 1 & x_6 & x_6^2 \\ 1 & x_7 & x_7^2 \end{bmatrix} \cdot \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \\ \epsilon_6 \\ \epsilon_7 \end{bmatrix} \quad (12.4)$$

Note that unknown parameters β_i enter only linearly, and the quadratic components are restricted to the (known) data matrix. Therefore the fit required to determine the parameters β_i is a *linear* fit, even though the curve is quadratic!

12.2.3 Multilinear Regression

If we have truly independent variables, *multilinear regression* (or *multiple regression*) is a straightforward extension of the simple linear regression.

As an example, let us look at a *multiple regression* with covariates (i.e., independent variables) w_i and x_i . Suppose that the data are 7 observations, and for each observed value to be predicted (y_i) there are two covariates that were also observed, w_i and x_i . The model to be considered is

$$y_i = \beta_0 + \beta_1 w_i + \beta_2 x_i + \epsilon_i, \quad (12.5)$$

This model can be written in matrix terms as

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & w_1 & x_1 \\ 1 & w_2 & x_2 \\ 1 & w_3 & x_3 \\ 1 & w_4 & x_4 \\ 1 & w_5 & x_5 \\ 1 & w_6 & x_6 \\ 1 & w_7 & x_7 \end{bmatrix} \cdot \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \\ \epsilon_6 \\ \epsilon_7 \end{bmatrix}, \quad (12.6)$$

12.2.4 Patsy—The Formula Language

The mini-language describing formulas in statistics was first used in the languages *R* and *S*. It is now also available in Python through the Python package *patsy*.

For instance, if we have a variable y , and we want to regress it against another variable x , we can simply write

Table 12.1 The most important elements of the formula syntax

Operator	Meaning
\sim	Separates the left-hand side from the right-hand side. If omitted, a formula is assumed right-hand side only
$+$	Combines terms on either side (set union)
$-$	Removes terms on the right from set of terms on the left (set difference)
$*$	$a*b$ is shorthand for the expansion $a + b + a:b$
$/$	a/b is shorthand for the expansion $a + a:b$. It is used when b is nested within a (e.g., states and counties)
$:$	Computes the interaction between terms on the left and right
$**$	Takes a set of terms on the left and an integer n on the right and computes the $*$ of that set of terms with itself n times

$$y \sim x \quad (12.7)$$

By default an offset is automatically assumed in the formula language. A more complex situation where y depends on the variables x, a, b , and the interaction of a and b can be expressed by

$$y \sim x + a + b + a:b \quad (12.8)$$

This formula language is based on the notation introduced by Wilkinson and Rogers (Wilkinson and Rogers, 1973). The symbols in Table 12.1 are used on the right-hand side to denote different interactions. A complete set of the description can be found under <http://patsy.readthedocs.org>.

12.2.5 Design Matrix

a) Definition

A very general definition of a regression model is the following:

$$y = f(x, \epsilon) \quad (12.9)$$

In the case of a linear regression model, the model can be rewritten as:

$$y = \mathbf{X} \cdot \boldsymbol{\beta} + \epsilon, \quad (12.10)$$

The matrix \mathbf{X} is sometimes called the *design matrix* for the model. For a simple linear regression and multiple regression, the corresponding design matrices are given in Eqs. 12.2 and 12.6, respectively.

² Given a data set $\{y_i, x_{i1}, \dots, x_{ip}\}_{i=1,\dots,n}$ of n statistical units, a linear regression model assumes that the relationship between the dependent variable y_i and the p -vector of regressors x_i is linear. This relationship is modelled through a disturbance term or error variable ε_i , an unobserved random variable that adds noise to the linear relationship between the dependent variable and regressors. Thus the model takes the form

$$y_i = \beta_0 + \beta_1 x_{i1} + \dots + \beta_p x_{ip} + \varepsilon_i = \beta_0 + \mathbf{x}_i^T \cdot \boldsymbol{\beta} + \varepsilon_i, \quad i = 1, \dots, n, \quad (12.11)$$

where T denotes the transpose so that $\mathbf{x}_i^T \cdot \boldsymbol{\beta}$ is the inner product between the vectors \mathbf{x}_i and $\boldsymbol{\beta}$.

Often these n equations are stacked together and written in vector form as

$$\mathbf{y} = \beta_0 + \mathbf{X} \cdot \boldsymbol{\beta} + \boldsymbol{\varepsilon}, \quad (12.12)$$

where

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, \quad \mathbf{X} = \begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_n^T \end{pmatrix} = \begin{pmatrix} x_{11} & \cdots & x_{1p} \\ x_{21} & \cdots & x_{2p} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{np} \end{pmatrix}, \quad \boldsymbol{\beta} = \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_p \end{pmatrix}, \quad \boldsymbol{\varepsilon} = \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{pmatrix}. \quad (12.13)$$

Some remarks on terminology and general use:

- y_i is called the *regressand*, *endogenous variable*, *response variable*, *measured variable*, or *dependent variable*. The decision as to which variable in a data set is modeled as the dependent variable and which are modeled as the independent variables may be based on a presumption that the value of one of the variables is caused by or directly influenced by the other variables. Alternatively, there may be an operational reason to model one of the variables in terms of the others, in which case there need be no presumption of causality.
- \mathbf{x}_i are called *regressors*, *exogenous variables*, *explanatory variables*, *covariates*, *input variables*, *predictor variables*, or *independent variables*. (The expression *independent variables* is meant in contrast to *dependent variables*, but not to be confused with *independent random variables*, where “independent” indicates that those variables do not depend on anything else).
 - Usually a constant is included as one of the regressors. For example we can take $x_{i0} = 1$ for $i = 1, \dots, n$, simplifying Eq. 12.12 to Eq. 12.10. The corresponding element of $\boldsymbol{\beta}$ is called the *intercept*. Many statistical inference procedures for

² This section has been taken from Wikipedia https://en.wikipedia.org/wiki/Linear_regression, last accessed Sept-18-2021.

linear models require an intercept to be present, so it is often included even if theoretical considerations suggest that its value should be zero.

- Sometimes one of the regressors can be a non-linear function of another regressor or of the data, as in polynomial regression and segmented regression. The model remains *linear* as long as it is linear in the parameter vector β (see Eq. 12.4).
- β is a p -dimensional parameter vector. Its elements are also called *effects*, or *regression coefficients*. Statistical estimation and inference in linear regression focuses on β .
- ε_i is called the *residuals*, *error term*, *disturbance term*, or *noise*. This variable captures all other factors which influence the dependent variable y_i other than the regressors x_i . The relationship between the error term and the regressors, for example whether they are correlated, is a crucial step in formulating a linear regression model, as it will determine the method to use for estimation.
- If $p = 1$ in Eq. 12.11, we have a *simple linear regression*, corresponding to Eq. 11.4. If $i > 1$ we talk about *multilinear regression* or *multiple linear regression* (see Eq. 12.6).

b) Examples: One-way ANOVA

One-way ANOVA (Cell Means Model) This example demonstrates a one-way analysis of variance (ANOVA) with 3 groups and 7 observations. The given data set has the first three observations belonging to the first group, the following two observations belong to the second group, and the final two observations are from the third group. If the model to be fit is just the mean of each group, then the model is

$$y_{ij} = \mu_i + \epsilon_{ij}, \quad i = 1, 2, 3 \quad (12.14)$$

which can be written as

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mu_1 \\ \mu_2 \\ \mu_3 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \\ \epsilon_6 \\ \epsilon_7 \end{bmatrix} \quad (12.15)$$

It should be emphasized that in this model μ_i represents the mean of the i th group.

This type of coding of categorical or ordinal variables, where the presence/absence of each category is indicated by 0/1-s, is called “dummy coding” or “one-hot-encoding” of variables.

One-way ANOVA (offset from reference group) The ANOVA model could be equivalently written as each group parameter τ_i being an offset from some overall reference. Typically this reference point is taken to be one of the groups under consideration. This makes sense in the context of comparing multiple treatment groups to a control group, and the control group is considered the “reference”. In this example, group 1 was chosen to be the reference group. As such the model to be fit is:

$$y_{ij} = \mu + \tau_i + \epsilon_{ij}, \quad i = 1, 2, 3 \quad (12.16)$$

with the constraint that τ_1 is zero.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mu \\ \tau_2 \\ \tau_3 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \\ \epsilon_6 \\ \epsilon_7 \end{bmatrix} \quad (12.17)$$

In this model μ is the mean of the reference group and τ_i is the difference from group i to the reference group. τ_1 and is not included in the matrix because its difference from the reference group (itself) is necessarily zero.

12.3 Linear Regression Analysis with Python

12.3.1 Example 1: Line Fit with Confidence Intervals

For univariate distributions, the 95%-confidence intervals based on the standard deviation indicate the interval that we expect to contain 95% of the data, and the confidence intervals based on the standard error of the mean indicates the interval that contains the true mean with 95% probability. We also have these two types of confidence intervals (one for the data, and one for the fitted parameters) for line fits, and they are shown in Fig. 12.3.

The corresponding equation is Eq. 11.4, and the formula syntax is given by Eq. 12.7.



Code: ISP_fitLine.py³ Linear regression fit, with the output shown in Fig. 12.3.

³ [ISP2e>/12_LinearModels/fitLine/ISP_fitLine.py](https://ISP2e.com/12_LinearModels/fitLine/ISP_fitLine.py).

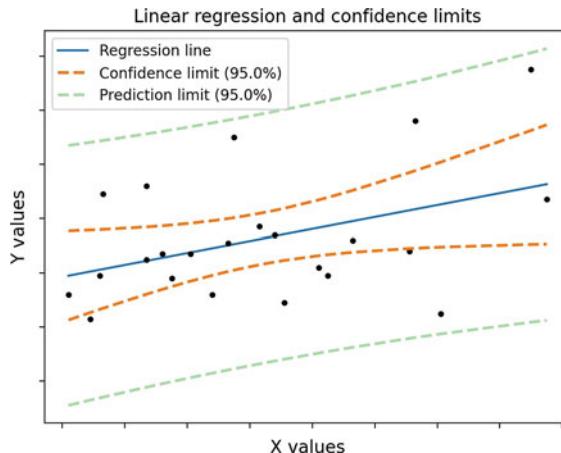


Fig. 12.3 Regression, with confidence intervals for the mean as well as for the predicted data. The orange dotted line shows the confidence interval for the mean; and the green dotted line the confidence interval for predicted data. The corresponding code can be found in `ISP_fitLine.py`

12.3.2 Example 2: Noisy Quadratic Polynomial

To see how different models can be used to evaluate a given set of data, let us look at a simple example: fitting a noisy, slightly quadratic curve. Let us start with the algorithms implemented in `numpy`, and fit a linear, quadratic, and a cubic curve to the data.

```
In [1]: import numpy as np
....: import matplotlib.pyplot as plt

In [2]: ''' Generate a noisy, slightly quadratic dataset '''
....: x = np.arange(100)
....: y = 150 + 3*x + 0.03*x**2 + 5*np.random.randn(len(x))
....:

In [3]: # Create the Design Matrices for the linear,
        quadratic,
        # and cubic fit
....: M1 = column_stack((np.ones_like(x), x))
....: M2 = column_stack((np.ones_like(x), x, x**2))
....: M3 = column_stack((np.ones_like(x), x, x**2, x**3))
....:
....: # an equivalent alternative solution with statsmodels
      # would be
....: # M1 = sm.add_constant(x)
....:

In [4]: # Solve the equations
....: p1 = np.linalg.lstsq(M1, y)
....: p2 = np.linalg.lstsq(M2, y)
....: p3 = np.linalg.lstsq(M3, y)
....:
```

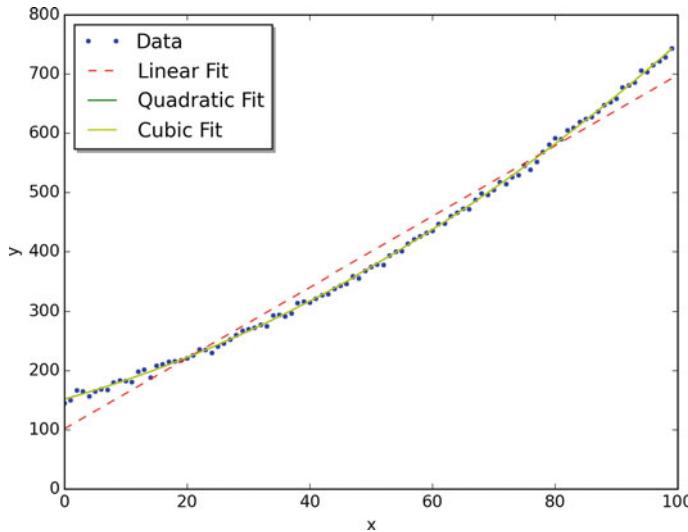


Fig. 12.4 A noisy, slightly quadratic data set, with a linear, a quadratic, and a cubic fit superposed. The quadratic and the cubic lines are almost exactly the same

```
In [5]: np.set_printoptions(precision=3)

In [6]: print(f'Coefficients from the linear fit: {p1[0]}')
Coefficients from the linear fit:
[ 100.42    5.98]

In [7]: print(f'Coefficients from the quadratic fit: {p2[0]}')
Coefficients from the quadratic fit:
[ 1.48e+02   3.10e+00   2.91e-02]

In [8]: print(f'Coefficients from the cubic fit: {p3[0]}')
Coefficients from the cubic fit:
[ 1.47e+02   3.12e+00   2.84e-02   4.81e-06]
```

With this simple analytical solution we can obtain the fitted coefficients (β_i in Eq.12.11) for a linear, a quadratic, and a cubic model. And as we see in Fig.12.4, the quadratic and the cubic fit are both very good and essentially undistinguishable.

If we want to find out which is the “better” fit, we can use the tools provided by *statsmodels* to again fit the model. Using *statsmodels* we obtain not only the best-fit parameters, but a wealth of additional information about the model:

```
In [9]: '''Solution with the tools from statsmodels'''
....: import statsmodels.api as sm
....:
....: Res1 = sm.OLS(y, M1).fit()
```

```
....: Res2 = sm.OLS(y, M2).fit()
....: Res3 = sm.OLS(y, M3).fit()
```

```
In [10]: print(Res1.summary2())
```

```
Results: Ordinary least squares
=====
Model: OLS Adj. R-squared: 0.983
Dependent Variable: y AIC: 909.6344
Date: 2015-06-27 13:50 BIC: 914.8447
No. Observations: 100 Log-Likelihood: -452.82
Df Model: 1 F-statistic: 5818.
Df Residuals: 98 Prob (F-statistic): 4.46e-89
R-squared: 0.983 Scale: 512.18
-----
      Coef. Std.Err. t P>|t| [0.025 0.975]
-----
const 100.4163 4.4925 22.3519 0.0000 91.5010 109.3316
x1     5.9802 0.0784 76.2769 0.0000 5.8246 6.1358
-----
Omnibus: 10.925 Durbin-Watson: 0.131
Prob(Omnibus): 0.004 Jarque-Bera (JB): 6.718
Skew: 0.476 Prob(JB): 0.035
Kurtosis: 2.160 Condition No.: 114
=====
```

```
In [11]: print(f'The AIC-value is {Res1.aic:4.1f} for the
linear fit,\n' +\
....: f'{Res2.aic:4.1f} for the quadratic fit, and \n' +\
....: f'{Res3.aic:4.1f} for the cubic fit')

The AIC-value is 909.8 for the linear fit,
578.7 for the quadratic fit, and
580.2 for the cubic fit
```

In the next section we will explain the meaning of all these parameters in detail. Here I just want to point out the *AIC*-value, the *Akaike Information Criterion*, which can be used to assess the quality of the model: the lower the AIC-value, the better the model. We see that the quadratic model has the lowest AIC-value and therefore is the best model: it provides the same quality of fit as the cubic model but uses fewer parameters to achieve that quality.

Before we move to the next example, let me show how the formula language can be used to perform the same fits, but without having to manually generate the design matrices, and how to extract, e.g., the model parameters, standard errors, and confidence intervals. Note that the use of *pandas* DataFrames allows Python to add information about the individual parameters.

```
In [14]: '''Formula-based modeling'''
....: import pandas as pd
....: import statsmodels.formula.api as smf
....:
....: # Turn the data into a pandas DataFrame so that we
....: # can address them in the formulas with their name
....: df = pd.DataFrame({'x':x, 'y':y})
....:
```

```

....: # Fit the models, and show the results
....: Res1F = smf.ols('y~x', df).fit()
....: Res2F = smf.ols('y ~ x+I(x**2)', df).fit()
....: Res3F = smf.ols('y ~ x+I(x**2)+I(x**3)', df).fit()

In [15]: Res2F.params # show e.g. parameters for quadratic
         fit
Out[15]:
Intercept      148.022539
x              3.043490
I(x ** 2)     0.029454
dtype: float64

In [16]: Res2F.bse    # standard errors
Out[16]:
Intercept    1.473074
x            0.068770
I(x ** 2)   0.000672
dtype: float64

In [17]: Res2F.conf_int()    # 95%-CI
Out[17]:
          0           1
Intercept 145.098896 150.946182
x          2.907001  3.179978
I(x ** 2)  0.028119  0.030788

```

 **Code:** ISP_modelImplementations.py⁴ Three ways how to solve a linear regression model in *Python*.

12.4 Model Results of Linear Regression Models

The output of linear regression models, such as the one in Fig. 12.4, can at first be daunting. Since understanding this type of output is a worthwhile step towards more complex models, I will in the following present a simple example, and explain the output step-by-step. We will use Python to explore measures of fits for linear regression: the coefficient of determination (R^2), hypothesis tests (F, T, Omnibus), and other measures.⁵

⁴ <ISP2e>/12_LinearModels/modelImplementations/ISP_modelImplementations.py.

⁵ The following is based on the blog of Connor Johnson (<http://connor-johnson.com/2014/02/18/linear-regression-with-python/>), with author's permission.

12.4.1 Example: Tobacco and Alcohol in the UK

First we will look at a small data set from the DASL library (<https://dasl.datadescription.com/datafile/tobacco-and-alcohol>), regarding the correlation between tobacco and alcohol purchases in different regions of the United Kingdom. The interesting feature of this data set is that Northern Ireland is reported as an outlier. Notwithstanding, we will use this data set to describe two tools for calculating a linear regression. We will alternatively use the *statsmodels* and *sklearn* modules for calculating the linear regression, while using *pandas* for data management, and *matplotlib* for plotting. To begin, we will import the modules, get the data into Python, and have a look at them (Fig. 12.5):

```
In [1]: import numpy as np
....: import pandas as pd
....: import matplotlib as mpl
....: import matplotlib.pyplot as plt
....: import statsmodels.formula.api as sm
....: from sklearn.linear_model import LinearRegression
....: from scipy import stats
....:

In [2]: data_str = '''Region Alcohol Tobacco
....: North 6.47 4.03
....: Yorkshire 6.13 3.76
....: Northeast 6.19 3.77
....: East_Midlands 4.89 3.34
....: West_Midlands 5.63 3.47
....: East_Anglia 4.52 2.92
....: Southeast 5.89 3.20
....: Southwest 4.79 2.71
....: Wales 5.27 3.53
....: Scotland 6.08 4.51
....: Northern_Ireland 4.02 4.56'''
....:
....: from io import StringIO
....: df = pd.read_csv(StringIO(data_str),
....:                  delim_whitespace=True)
....:

In [3]: # Plot the data
....: df.plot('Tobacco', 'Alcohol', style='o')
....: plt.ylabel('Alcohol')
....: plt.title('Sales in Several UK Regions')
....: plt.show()
```

Fitting the model, leaving the outlier (which is the last data point) for the moment away is then very easy:

```
In [4]: result = sm.ols('Alcohol ~ Tobacco', df[:-1]).fit()
....: print(result.summary())
....:
```

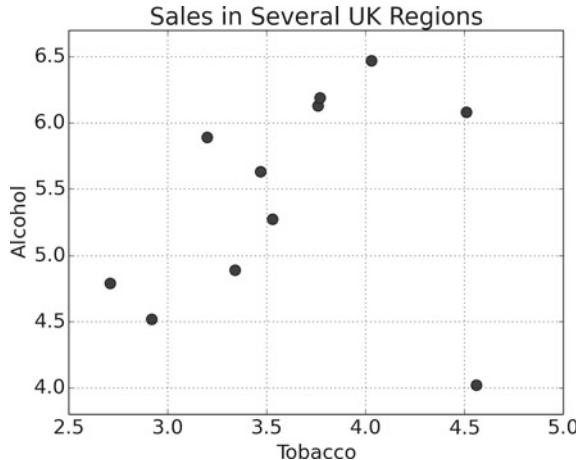


Fig. 12.5 Sales of Alcohol versus Tobacco in the UK. We notice that there seems to be a linear trend, and one outlier, which corresponds to Northern Ireland

Note that using the `formula.api` module from `statsmodels`, an intercept is automatically added. This gives us

OLS Regression Results						
Dep. Variable:	Alcohol	R-squared:	0.615			
Model:	OLS	Adj. R-squared:	0.567			
Method:	Least Squares	F-statistic:	12.78			
Date:	Sun, 27 Apr 2014	Prob (F-statistic):	0.00723			
Time:	13:19:51	Log-Likelihood:	-4.9998			
No. Observations:	10	AIC:	14.00			
Df Residuals:	8	BIC:	14.60			
Df Model:	1					
=====						
	coef	std err	t	P> t	[95.0% Conf. Int.]	
Intercept	2.0412	1.001	2.038	0.076	-0.268	4.350
Tobacco	1.0059	0.281	3.576	0.007	0.357	1.655
=====						
Omnibus:		2.542	Durbin-Watson:			1.975
Prob(Omnibus):		0.281	Jarque-Bera (JB):			0.904
Skew:		-0.014	Prob(JB):			0.636
Kurtosis:		1.527	Cond. No.			27.2
=====						

And now we have a very nice table of numbers—which at first looks fairly daunting. To explain what the individual numbers mean, I will go through and explain each one.

Most of the values listed in the summary are available via the `result` object. For instance, the R^2 value can be obtained by `result.rsquared`. If you are using *IPython*, you may type `result.` and hit the TAB key, to see a list of all possible attributes for the `result`-object.

12.4.2 Model Characteristics

a) Degrees of Freedom

The OLS Regression Results are structured into three blocks. The first block contains the characteristics of the model. The left column of the first table is mostly self-explanatory. The degrees of freedom (Df) of the model are the number of predictors, or explanatory, variables. The Df of the residuals is the number of observations minus the degrees of freedom of the model, minus one (for the offset).

The Sum-of Squares variables SS_{xx} have already been defined above, in Sect. 11.3. n is the number of observations, and k is the number of regression parameters. For example, if you fit a straight line, $k = 2$. And as above (Sect. 11.3) \hat{y}_i will indicate the fitted model values, and \bar{y} the mean. In addition to these, the following variables will be used:

- $DF_{\text{mod}} = k - 1$ is the (*Corrected*) *Model Degrees of Freedom*. (The “-1” comes from the fact that we are only interested in the correlation, not in the absolute offset of the data.)
- $DF_{\text{res}} = n - k$ is the *Residuals Degrees of Freedom*
- $DF_{\text{tot}} = n - 1$ is the (*Corrected*) *Total Degrees of Freedom*. The Horizontal line regression is the null hypothesis model.

For multiple regression models with intercept, $DF_{\text{mod}} + DF_{\text{res}} = DF_{\text{tot}}$.

- $MS_{\text{mod}} = SS_{\text{mod}}/DF_{\text{mod}}$: *Model Mean of Squares*
- $MS_{\text{res}} = SS_{\text{res}}/DF_{\text{res}}$: *Residuals Mean of Squares*. MS_{res} is an unbiased estimate for σ^2 for multiple regression models.
- $MS_{\text{tot}} = SS_{\text{tot}}/DF_{\text{tot}}$: *Total Mean of Squares*, which is the sample variance of the y-variable.

b) The R^2 Value

As we have already seen in Sect. 11.3, the R^2 value indicates the proportion of variation in the y-variable that is due to variation in the x-variables. For simple linear regression, the R^2 value is the square of the sample correlation r_{xy} . For multiple linear regression with intercept (which includes simple linear regression), the R^2 value is defined as

$$R^2 = \frac{SS_{\text{mod}}}{SS_{\text{tot}}} \quad (12.18)$$

c) \bar{R}^2 —The Adjusted R^2 Value

For assessing the quality of models, many researchers prefer the *adjusted R^2 value*, commonly indicated with the bar above the \bar{R} , which is penalized for having a large number of parameters in the model.

Here is the logic behind the definition of \bar{R}^2 : R^2 is defined as $R^2 = 1 - SS_{\text{res}}/SS_{\text{tot}}$ or $1 - R^2 = SS_{\text{res}}/SS_{\text{tot}}$. To take into account the number of regression parameters p , define the *adjusted R-squared* value as

$$1 - \bar{R}^2 = \frac{\text{ResidualVariance}}{\text{TotalVariance}} \quad (12.19)$$

where (*Sample*) *Residual Variance* is estimated by $SS_{\text{res}}/DF_{\text{res}} = SS_{\text{res}}/(n - k)$, and (*Sample*) *Total Variance* is estimated by $SS_{\text{tot}}/DF_{\text{tot}} = SS_{\text{tot}}/(n - 1)$. Thus,

$$\begin{aligned} 1 - \bar{R}^2 &= \frac{SS_{\text{res}}/(n - k)}{SS_{\text{tot}}/(n - 1)} \\ &= \frac{SS_{\text{res}}}{SS_{\text{tot}}} \frac{n - 1}{n - k} \end{aligned} \quad (12.20)$$

so

$$\begin{aligned} \bar{R}^2 &= 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}} \frac{n - 1}{n - k} \\ &= 1 - (1 - R^2) \frac{n - 1}{n - k} \end{aligned} \quad (12.21)$$

d) The F-test

For a multiple regression model with intercept,

$$\begin{aligned} Y_j &= \alpha + \beta_1 X_{1j} + \dots + \beta_n X_{nj} + \epsilon_i \\ &= \alpha + \sum_{i=1}^n \beta_i X_{ij} + \epsilon_j \\ &= E(Y_j|X) + \epsilon_j \end{aligned} \quad (12.22)$$

In the last line $E(Y_j|X)$ indicates the “expected value for Y, given X”.

We want to test the following null hypothesis and alternative hypothesis:

$H_0: \beta_1 = \beta_2 = \dots = \beta_n = 0$

$H_1: \beta_j \neq 0$, for at least one value of j

This test is known as the *overall F-test for regression*.

Remember, if t_1, t_2, \dots, t_m are independent, $N(0, \sigma^2)$ random variables, then $\sum_{i=1}^m \frac{t_i^2}{\sigma^2}$ is a χ^2 (chi-squared) random variable with m degrees of freedom. It can be shown that if H_0 is true and the residuals are unbiased, homoscedastic (i.e., all function values have the same variance), independent, and normal (see Sect. 12.5), then:

1. SS_{res}/σ^2 has a χ^2 distribution with DF_{res} degrees of freedom.
2. SS_{mod}/σ^2 has a χ^2 distribution with DF_{mod} degrees of freedom.
3. SS_{res} and SS_{mod} are independent random variables.

If u is a χ^2 random variable with n degrees of freedom, v is a χ^2 random variable with m degrees of freedom, and u and v are independent, then $F = \frac{u/n}{v/m}$ has an F distribution with (n, m) degrees of freedom.

If H_0 is true,

$$\begin{aligned} F &= \frac{(SS_{\text{mod}}/\sigma^2)/DF_{\text{mod}}}{(SS_{\text{res}}/\sigma^2)/DF_{\text{res}}} \\ &= \frac{SS_{\text{mod}}/DF_{\text{mod}}}{SS_{\text{res}}/DF_{\text{res}}} \\ &= \frac{MS_{\text{mod}}}{MS_{\text{res}}}, \end{aligned} \tag{12.23}$$

has an F distribution with $(DF_{\text{mod}}, DF_{\text{res}})$ degrees of freedom, and is independent of σ .

We can test this directly in Python with

```
In [5]: N = result.nobs
....: k = result.df_model+1
....: dfm, dfe = k-1, N - k
....: F = result.mse_model / result.mse_resid
....: p = 1.0 - stats.f.cdf(F, dfm, dfe)
....: print(f'F-statistic: {F:.3f}, p-value: {p:.5f}')
....:
F-statistic: 12.785, p-value: 0.00723
```

which corresponds to the values in the model summary above.

Here, `stats.f.cdf(F, m, n)` returns the cumulative sum of the F-distribution with shape parameters $m = k-1 = 1$, and $n = N - k = 8$, up to the F-statistic F . Subtracting this quantity from one, we obtain the probability in the tail, which represents the probability of observing F-statistics more extreme than the one observed.

e) Log-Likelihood Function

A very common approach in statistics is the idea of *maximum likelihood estimation*. The basic idea is quite different from the *OLS (least square)* approach: in the least square approach the model is constant, and the errors of the response are variable; in contrast, in the maximum likelihood approach, the data response values are regarded as constant, and the likelihood of the model is maximised. (The concept of maximum likelihood estimation is very well explained in Duda et al. 2004.)

For the classical linear regression model (with normal errors) we have

$$\epsilon = y_i - \sum_{k=1}^n \beta_k x_{ik} = y_i - \hat{y}_i \in N(0, \sigma) \tag{12.24}$$

so the probability density is given by

$$p(\epsilon_i) = \Phi\left(\frac{y_i - \hat{y}_i}{\sigma}\right) \quad (12.25)$$

where $\Phi(z)$ is the standard normal probability distribution function. The probability of independent samples is the product of the individual probabilities

$$\Pi_{total} = \prod_{i=1}^n p(\epsilon_i) \quad (12.26)$$

The *Log Likelihood function* is defined as

$$\begin{aligned} \ln(\mathcal{L}) &= \ln(\Pi_{total}) \\ &= \ln \left[\prod_{i=1}^n \frac{1}{\sigma \sqrt{2\pi}} \exp\left(-\frac{(y_i - \hat{y}_i)^2}{2\sigma^2}\right) \right] \\ &= \sum_{i=1}^n \left[\ln\left(\frac{1}{\sigma \sqrt{2\pi}}\right) - \left(\frac{(y_i - \hat{y}_i)^2}{2\sigma^2}\right) \right] \end{aligned}$$

It can be shown that the maximum likelihood estimator of σ^2 is

$$E(\sigma^2) = \frac{SS_{\text{res}}}{n} \quad (12.27)$$

We can calculate this in Python as follows:

```
In [6]: N = result.nobs
....: SSR = result.ssr
....: sigma2 = SSR / N
....: L = (1.0/np.sqrt(2*np.pi*sigma2))**N * np.exp(-SSR
    / (2*sigma2))
....: print('ln(L) = ', np.log( L ))
....:
ln(L) = -4.9998
```

which again matches the model summary.

f) Information Content of Statistical Models—AIC and BIC

To judge the quality of a model, one should first visually inspect the residuals. In addition, one can also use a number of numerical criteria to assess the quality of a statistical model. These criteria represent various approaches for balancing model accuracy with parsimony.

Above we have already encountered the *adjusted R²* value, which—in contrast to the R² value—decreases if there are too many regressors in the model.

Other commonly encountered criteria are the *Akaike Information Criterion (AIC)* and the *Schwartz* or *Bayesian Information Criterion (BIC)*, which are based on the log-likelihood described in the previous section. Both measures introduce a penalty for model complexity, but the AIC penalizes complexity less severely than the BIC. The *Akaike Information Criterion AIC* is given by

$$AIC = 2 * k - 2 * \ln(\mathcal{L}) \quad (12.28)$$

and the *Schwartz* or *Bayesian Information Criterion BIC* by

$$BIC = k * \ln(N) - 2 * \ln(\mathcal{L}). \quad (12.29)$$

Here, N is the number of observations, k is the number of parameters, and \mathcal{L} is the likelihood. We have two parameters in this example, the slope and intercept. The AIC is a relative estimate of information loss between different models. The BIC was initially proposed using a Bayesian argument and does not relate to ideas of information. Both measures are only used when trying to decide between different models. So, if we have one regression for alcohol sales based on cigarette sales, and another model for alcohol consumption that incorporated cigarette sales and lighter sales, then we should choose the model with the lower AIC or BIC value.

12.4.3 Model Coefficients and Their Interpretation

The second block in the model summary on Sect. 12.4.2 contains the model coefficients and their interpretation.

a) Coefficients

The *coefficients* or weights of the linear regression are contained in `result.params` and returned as a *pandas Series* object, since we used a *pandas DataFrame* as input. This is nice because the coefficients are named for convenience.

```
In [7]: result.params
Out[7]:
Intercept    2.0412
Tobacco      1.0059
dtype: float64
```

We can obtain this directly by computing

$$\beta = (\mathbf{X}^T \cdot \mathbf{X})^{-1} \mathbf{X}^T \cdot \mathbf{y}. \quad (12.30)$$

Here, \mathbf{X} is the design matrix, i.e., the matrix of predictor variables as columns, with an extra column of ones for the constant term; \mathbf{y} is the column vector of the response variable, and β is the column vector of coefficients corresponding to the columns of \mathbf{X} . In Python:

```
In [8]: df['Ones'] = np.ones( len(df) )
...: Y = df.Alcohol[:-1]
...: X = df[['Tobacco', 'Ones']][:-1]
...:
```

Note: the “-1” in the indices excludes the last data point, i.e., the outlier Northern Ireland.

b) Standard Error

To obtain the *standard errors of the coefficients* we will calculate the *covariance-variance matrix*, also called the *covariance matrix*, for the estimated coefficients β of the predictor variables using

$$C = \text{cov}(\beta) = \sigma^2(\mathbf{X}^T \cdot \mathbf{X})^{-1}. \quad (12.31)$$

Here, σ^2 is the variance, or the mean-squared-error of the residuals. The standard errors are the square roots of the elements on the main diagonal of this covariance matrix. We can perform the operation above and calculate the element-wise square root using the following Python code,

```
In [9]: X = df.Tobacco[:-1]
...:
...: # add a column of ones for the constant intercept
...: term
...: X = column_stack( (np.ones_like(X), X) )
...:
...: # perform the matrix multiplication,
...: # and then take the inverse
...: C = np.linalg.inv(X.T @ X)
...:
...: # multiply by the mean squared error of the residual
...: C *= result.mse_resid
...:
...: # take the square root
...: SE = np.sqrt(C)
...:
...: print(SE)
...:
[[ 1.00136021      nan]
 [       nan  0.28132158]]
```

c) t-statistic

We use the t-test to test the null hypothesis that the coefficient of a given predictor variable is zero, implying that a given predictor has no appreciable effect on the response variable. The alternative hypothesis is that the predictor does contribute to the response. In testing we set some threshold, $\alpha = 0.05$ or 0.001 , calculate the corresponding t-value “T”, and if $\Pr(T \geq |t|) < \alpha$ then we reject the null hypothesis at our threshold α , otherwise we fail to reject the null hypothesis. The t-test generally allows us to evaluate the importance of different predictors, *assuming that the*

residuals of the model are normally distributed about zero. If the residuals do not behave in this manner, then that suggests that there is some non-linearity between the variables, and that their t-tests should not be used to assess the importance of individual predictors. Furthermore, it might be best to try to modify the model so that the residuals do tend the cluster normally about zero.

The t-statistic is given by the ratio of the coefficient (or factor) of the predictor variable of interest and its corresponding standard error. If β is the vector of coefficients or factors of our predictor variables, and SE is our standard error, then the t-statistic is given by,

$$t_i = \beta_i / SE_{i,i} \quad (12.32)$$

So, for the first factor, corresponding to the slope in our example, we have the following code

```
In [10]: i = 1
....: beta = result.params[i]
....: se = SE[i,i]
....: t = beta / se
....: print(f't = {t:.3f}')
....:
t = 3.575
```

Once we have a t-statistic, we can calculate the probability of observing a statistic at least as extreme as what we have already observed, given our assumptions about the normality of our errors by using the code

```
In [11]: N = result.nobs
....: k = result.df_model + 1
....: dof = N - k
....: p_onesided = stats.t(dof).sf(t)
....: p = p_onesided * 2.0
....: print(f'p = {p:.3f}')
....:
p = 0.007
```

Here, dof are the degrees of freedom, which should be eight: the number of observations, N , minus the number of parameters, which is two. The CDF is the cumulative sum of the PDF. We are interested in the area under the right hand tail, beyond our t-statistic, t , so we calculate the survival-function for that statistic. Then we multiply this tail probability by two to obtain a two-tailed probability.

d) Confidence Interval

The confidence interval is built using the standard error, the p-value from our t-test, and a critical value from a t-test having $N - k$ degrees of freedom, where k is the number of observations and P is the number of model parameters, i.e., the number

of predictor variables. The confidence interval is the range of values in which we would expect to find the parameter of interest, based on what we have observed. You will note that we have a confidence interval for the predictor variable coefficient and for the constant term. A smaller confidence interval suggests that we are confident about the value of the estimated coefficient or constant term. A larger confidence interval suggests that there is more uncertainty or variance in the estimated term. Again, let me reiterate that hypothesis testing is only one perspective. Furthermore, it is a perspective that was developed in the late nineteenth and early twentieth centuries when data sets were generally smaller and more expensive to gather, and data scientists were using books of logarithm tables for arithmetic.

The confidence interval is given by,

$$CI = \beta_i \pm z * SE_{i,i} \quad (12.33)$$

Here, β_i is one of the estimated coefficients, z is a *critical-value*, which is the t-statistic required to obtain a probability less than the alpha significance level, and $SE_{i,i}$ is the standard error. The critical value is calculated using the inverse of the cumulative distribution function. In code, the confidence interval using a t-distribution looks like

```
In [12]: i = 0
...
...: # the estimated coefficient, and its variance
...: beta, c = result.params[i], SE[i,i]
...
...: # critical value of the t-statistic
...: N = result.nobs
...: P = result.df_model
...: dof = N - P - 1
...: z = stats.t(dof).ppf(0.975)
...
...: # the confidence interval
...: print(beta - z * c, beta + z * c)
...
-0.2679 4.3504
```

12.4.4 Analysis of Residuals

The third block in the model summary on p. 243 contains the parameters that characterize the residuals. If those clearly deviate from a normal distribution, then the model most likely has missed an essential element of the data.

The `OLS` command from `statsmodels.formula.api` provides some additional information about the residuals of the model: Omnibus, Skewness, Kurtosis, Durbin-Watson, Jarque-Bera, and the Condition number. In the following we will briefly describe these parameters.

a) Skewness and Kurtosis

Skew and kurtosis refer to the shape of a distribution (see Sect. 6.1.3). *Skewness* is a measure of the asymmetry of a distribution. And *kurtosis* is a measure of its curvature, specifically how pointed the curve is, and is for normally distributed data approximately 3. These values are defined by

$$S = \frac{\hat{\mu}_3}{\hat{\sigma}^3} = \frac{\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^3}{\left(\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2 \right)^{3/2}}, \quad (12.34a)$$

$$K = \frac{\hat{\mu}_4}{\hat{\sigma}^4} = \frac{\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^4}{\left(\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2 \right)^2} \quad (12.34b)$$

As you see, the $\hat{\mu}_3$ and $\hat{\mu}_4$ are the third and fourth central moments of a distribution.

The *excess kurtosis* is defined as $K - 3$, to ensure that its value for a normal distribution is equal to zero.

One possible Python implementation would be

```
In [13]: d = Y - result.fittedvalues
....:
....: S = np.mean( d**3.0 ) / np.mean( d**2.0 )** (3.0/2.0)
....: # equivalent to:
....: # S = stats.skew(result.resid, bias=True)
....:
....: K = np.mean( d**4.0 ) / np.mean( d**2.0 )** (4.0/2.0)
....: # equivalent to:
....: # K = stats.kurtosis(result.resid, fisher=False,
....: # bias=True)
....: print(f'Skewness: {S:.3f}, Kurtosis: {K:.3f}')
....:
Skewness: -0.014, Kurtosis: 1.527
```

b) Omnibus Test

The *Omnibus test* uses skewness and kurtosis to test the null hypothesis that a distribution is normal. In this case, we are looking at the distribution of the residuals. If we obtain a very small value for $P(\text{Omnibus})$, then the residuals are not normally distributed about zero, and we should maybe look at our model more closely. The *statsmodels OLS* function uses the `stats.normaltest()` function:

```
In [14]: (K2, p) = stats.normaltest(result.resid)
....: print(f'Omnibus: {K2:.3f}, p = {p:.3f}')
....:
Omnibus: 2.542, p = 0.281
```

Thus, if either the skewness or kurtosis suggests non-normality, this test should pick it up.

c) Durbin-Watson

The *Durbin-Watson test* is used to detect the presence of autocorrelation (a relationship between values separated from each other by a given time lag) in the residuals. Here the lag is one:

$$DW = \frac{\sum_{i=2}^N ((y_i - \hat{y}_i) - (y_{i-1} - \hat{y}_{i-1}))^2}{\sum_{i=1}^N (y_i - \hat{y}_i)^2} \quad (12.35)$$

```
In [15]: DW = np.sum( np.diff( result.resid.values )**2.0 ) \
...:           / result.ssr
...: print(f'Durbin-Watson: {DW:.3f}')
...:
Durbin-Watson: 1.975
```

d) Jarque-Bera Test

The *Jarque-Bera test* is another test that considers skewness (S) and kurtosis (K). The null hypothesis is that the distribution is normal, so that both the skewness and excess kurtosis equal zero. Unfortunately, with small samples the Jarque-Bera test is prone to rejecting the null hypothesis — that the distribution is normal — when it is in fact true.

$$JB = \frac{N}{6} \left(S^2 + \frac{1}{4}(K - 3)^2 \right) \quad (12.36)$$

Calculating the Jarque-Bera statistic using the χ^2 distribution with two degrees of freedom we have

```
In [16]: JB = (N/6.0) * (S**2.0 + (1.0/4.0)*( K - 3.0 )**2.0)
...: p = stats.chi2(2).sf(JB)
...: print(f'JB-statistic: {JB:.3f}, p-value: {p:.3f}')
...:
JB-statistic: 0.904, p-value: 0.636
```

e) Condition Number

The *condition number* measures the sensitivity of a function's output to its input. When two predictor variables are highly correlated, which is called *multicollinearity*, the coefficients or factors of those predictor variables can fluctuate erratically for small changes in the data or the model. Ideally, similar models should be similar,

i.e., have approximately equal coefficients. Multicollinearity can cause numerical matrix inversion to crap out or produce inaccurate results (see Kaplan 2009). One approach to this problem in regression is the technique of *ridge regression*, which is available in the Python package *sklearn*.

We calculate the condition number by taking the eigenvalues of the product of the predictor variables (including the constant vector of ones) and then taking the square root of the ratio of the largest eigenvalue to the smallest eigenvalue. If the condition number is greater than thirty, then the regression may have multicollinearity.

```
In [17]: EV = np.linalg.eig( X * X.T )
....: print(EV)
....:
(array([ 0.1841, 136.5153]),
 array([[[-0.9633, -0.2683],
       [ 0.2683, -0.9633]]]))
```

Note that $X.T * X$ should be $(P + 1) \times (P + 1)$, where P is the number of degrees of freedom of the model (the number of predictors) and the $+1$ represents the addition of the constant vector of ones for the intercept term. In our case, the product should be a 2×2 matrix, so we will have two eigenvalues. Then our condition number is given by

```
In [18]: CN = np.sqrt( EV[0].max() / EV[0].min() )
....: print(f'Condition No.: {CN:.3f}')
....:
```

Condition No.: 27.229

Our condition number is just below 30 (weak!), so we can sort of sleep okay.

12.4.5 Comparison to Model With Outlier

Now that we have seen an example of linear regression with a reasonable degree of linearity, compare that (and the corresponding model result on p. 243) with an example of one with a significant outlier. In practice, outliers should be understood before they are discarded, because they might turn out to be very important. They might signify a new trend or some possibly catastrophic event.

```
In [19]: X = df[['Tobacco', 'Ones']]
....: Y = df.Alcohol
....: result = sm.OLS( Y, X ).fit()
....: result.summary()
....:
```

```

OLS Regression Results
=====
Dep. Variable:      Alcohol   R-squared:       0.050
Model:              OLS        Adj. R-squared:  -0.056
Method:             Least Squares F-statistic:    0.4735
Date:               Sun, 27 Apr 2014 Prob (F-statistic): 0.509
Time:               12:58:27   Log-Likelihood:  -12.317
No. Observations:  11        AIC:            28.63
Df Residuals:      9         BIC:            29.43
Df Model:          1
=====
      coef    std err      t      P>|t|   [95.0% Conf. Int.]
-----
Intercept     4.3512    1.607    2.708    0.024      0.717    7.986
Tobacco       0.3019    0.439    0.688    0.509     -0.691    1.295
=====
Omnibus:            3.123   Durbin-Watson:  1.655
Prob(Omnibus):      0.210   Jarque-Bera (JB): 1.397
Skew:              -0.873   Prob(JB):       0.497
Kurtosis:           3.022   Cond. No.      25.5
=====
```

12.4.6 Regression Using Sklearn

scikit-learn is arguably the most advanced open source machine learning package available (<http://scikit-learn.org>). It provides simple and efficient tools for data mining and data analysis, covering supervised as well as unsupervised learning.

It provides tools for

- **Classification** Identifying to which category a new observation belongs.
- **Regression** Predicting a continuous value for a new example.
- **Clustering** Automatic grouping of similar objects into sets.
- **Dimensionality reduction** Reducing the number of random variables to consider.
- **Model selection** Comparing, validating and choosing parameters and models.
- **Preprocessing** Feature extraction and normalization.

Here we use it for the simple case of a regression analysis.

In order to use *sklearn*, we need to input our data in the form of vertical vectors.

```
In [20]: data = df[['Alcohol', 'Tobacco']].values
...: X, Y = np.c_[data[:, 0]], np.c_[data[:, 1]]
```

Next, we create the regression objects and fit the data to them. We will consider i) a clean set (which will fit a linear regression better), consisting of the data for all of the regions except Northern Ireland; and ii) an original set consisting of the original data.

```
In [21]: cln = LinearRegression()
...: org = LinearRegression()
...:
...: cln.fit(X[:-1], Y[:-1])
```

```

....: org.fit(X, Y)
....:
....: clean_score     = '{0:.3f}'.format(cln.score(X[:-1],
....:                                              Y[:-1]))
....: original_score = '{0:.3f}'.format(org.score(X, Y))
....:

```

Then we produce a scatter plot of the regions, with all of the regions plotted as empty blue circles, except for Northern Ireland, which is depicted as a red star (Fig. 12.6).

```

In [22]: mpl.rcParams['font.size']=16
....:
....: labelStart = 'All other regions, $R^2$ = '
....: plt.plot(df.Tobacco[:-1], df.Alcohol[:-1], 'bo',
....:           markersize=10, label = labelStart+clean_score)
....:
....: plt.hold(True)
....: labelStart = 'N. Ireland, outlier, $R^2$ = '
....: plt.plot(df.Tobacco[-1:], df.Alcohol[-1:], 'r*',
....:           ms=20, lw=10, label = labelStart+original_score)
....:

```

The next part generates a set of points from 2.5 to 4.85 and then predicts the response of those points using the linear regression object trained on the clean and original sets, respectively.

```

In [23]: test = np.c_[np.arange(2.5, 4.85, 0.1)]
....: plt.plot(test, cln.predict(test), 'k')
....: plt.plot(test, org.predict(test), 'k--')
....:

```

Finally, we limit and label the axes, add a title, overlay a grid, place the legend at the bottom, and then save the figure.

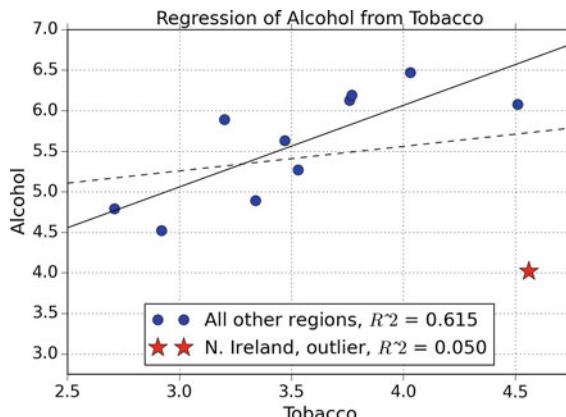


Fig. 12.6 Regression fits, with outlier included/excluded

```
In [24]: xlabel('Tobacco') ; xlim(2.5,4.75)
...: ylabel('Alcohol') ; ylim(2.75,7.0)
...: title('Regression of Alcohol from Tobacco')
...: grid()
...: legend(loc='lower center')
...: plt.show()
...:
```

12.4.7 Conclusion

Before you do anything, visualize your data. If your data is highly dimensional, then at least examine a few slices using box plots. At the end of the day, use your own judgement about a model based on your knowledge of your domain. Statistical tests should guide your reasoning, but they should not dominate it. In most cases, your data will not align itself with the assumptions made by most of the available tests. A very interesting, openly accessible article on classical hypothesis testing has been written by Nuzzo (2014). A more intuitive – but also more mathematical – approach to hypothesis testing is Bayesian analysis (see Chap. 14).

12.5 Assumptions and Interpretations of Linear Regression

12.5.1 Assumptions

Standard linear regression models with standard estimation techniques make a number of assumptions about the predictor variables, the response variables and their relationship. Numerous extensions have been developed that allow each of these assumptions to be relaxed, and in some cases eliminated entirely. Generally these extensions make the estimation procedure more complex and time-consuming, and may also require more data in order to get an accurate model.

The following are the major assumptions made by standard linear regression models with standard estimation techniques (e.g. ordinary least squares):

1. *Linear relationship*: There should be a linear relationship in the data!
2. *Homoscedasticity*: Equality of variances
3. *Independence*: No autocorrelation in the residuals
4. *Normality of residuals*
5. *Little or no collinearity*: The predictors should be independent

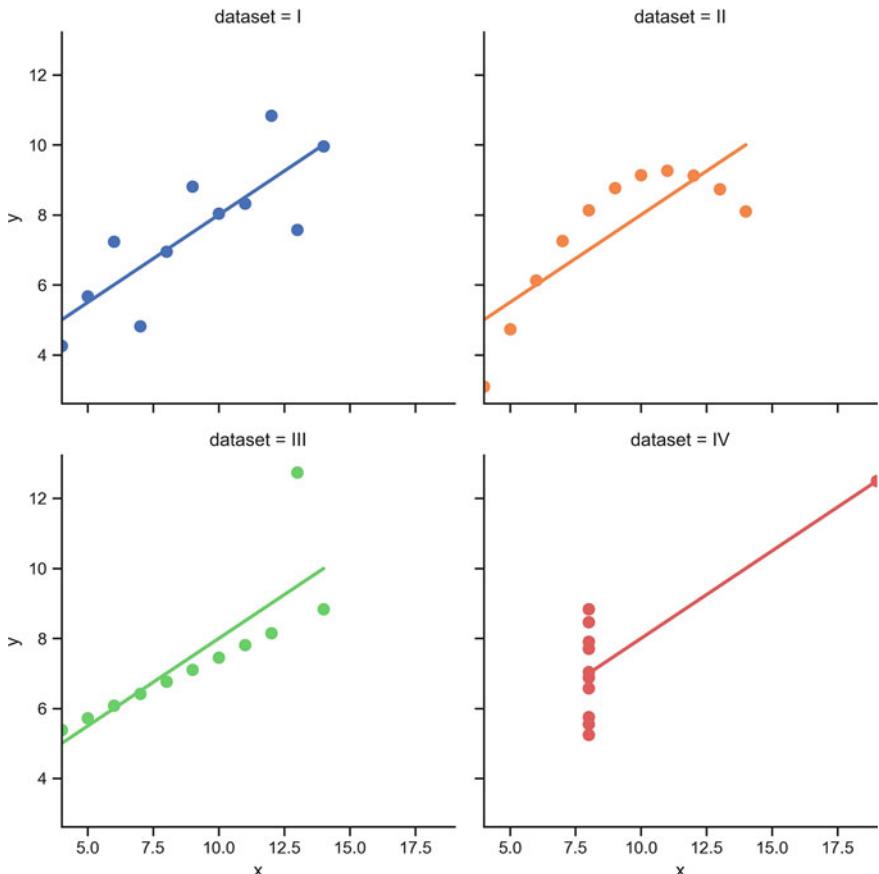


Fig. 12.7 The sets in the “Anscombe’s quartet” have the same linear regression line but are themselves very different

Let me discuss each of them in some more detail:

- **Linear Relationship:** Figure 12.7 (*Anscombe’s quartet*) shows how a linear fit can be meaningless if the wrong model is chosen, or if some of the assumptions are not met. For a meaningful fit, the response variable should be a linear combination of the parameters (regression coefficients) and the predictor variables. Note that this assumption is much less restrictive than it may at first seem. For example, if we square a variable and use the squares as one of the predictors, we can fit a polynomial curve to the data with linear regression (see Sect. 12.2.2). This makes linear regression an extremely powerful tool.
- **Constant variance** (aka *homoscedasticity*). This means that different response variables have the same variance in their errors, regardless of the values of the predictor variables. In practice this assumption is often invalid if the response

variables can vary over a wide scale. For example, a given person whose income is predicted to be \$100,000 may easily have an actual income of \$80,000 or \$120,000 (a standard deviation of around \$20,000), while another person with a predicted income of \$10,000 is unlikely to have the same \$20,000 standard deviation, which would imply their actual income would vary anywhere between -\$10,000 and \$30,000. (In fact, as this shows, in many cases - often the same cases where the assumption of normally distributed errors fails - the variance or standard deviation should be predicted to be proportional to the mean, rather than constant.) Simple linear regression estimation methods give less precise parameter estimates and misleading inferential quantities such as standard errors when substantial heteroscedasticity is present.

A scatterplot of the residuals is often a good way to check: if the data are not homoscedastic, the residuals systematically narrow or widen with increasing predictor values.

- **Independence of errors.** This assumes that the errors of the response variables should be uncorrelated with each other. This is often not the case in time-series: for example, when the temperature is high today, it will most likely be at least warm tomorrow. Relationships between adjacent data can be seen with the auto-correlation function, and quantified with the *Durbin-Watson test*.
- **Normality** When the model describes the data well, the residuals should be normally distributed (see Sect. 7.1.2). If the residuals systematically change with the value of the predictors, the model has often missed an important aspect of the data.
- **Lack of multicollinearity in the predictors.** For standard least squares estimation methods, the design matrix \mathbf{X} must have full column rank p ; otherwise, we have a condition known as *multicollinearity* in the predictor variables. This problem is analyzed clearly and in detail by (Kaplan, 2009). It can be triggered by having two or more correlated predictor variables. For example, with rising income the number of children typically decreases, while the time spent at a golf-court increases. If we use both variables as predictors, the results of a linear regression will lead to spurious solutions, which can depend e.g. on the sequence in which the predictors are used in the model.

An assumption which is mentioned less frequently is that the independent variables are known exactly, and that all the variability comes from the residuals. Look for example at Fig. 12.8: when we fit $y(x)$ (orange line), the x -values are fixed, and the residuals in the y -values (dashed lines) are minimized. In contrast, when we fit $x(y)$ (green line), the y -values are kept fixed, and the dotted residuals are minimized. The two resulting fits have different inclinations!

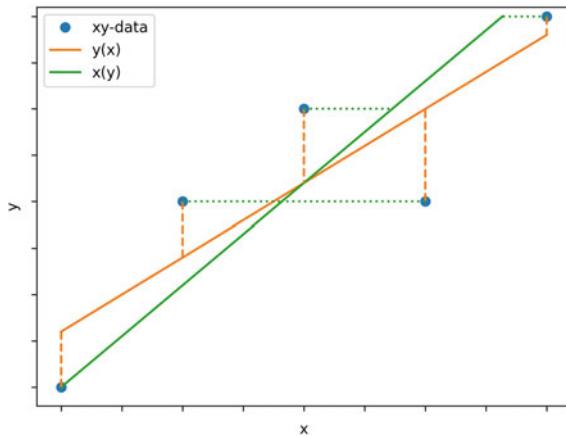


Fig. 12.8 With linear regression we assume that the predictors are known exactly, and all the variability comes from the dependent variable. As a result, fitting a line $y(x)$ leads to a different result than fitting $x(y)$

 **python**TM
Code: `ISP_anscombe.py`⁶ Code for the generation of Anscombe's Quartet.

Beyond these assumptions, several other statistical properties of the data strongly influence the performance of different estimation methods:

- The statistical relationship between the error terms and the regressors plays an important role in determining whether an estimation procedure has desirable sampling properties such as being unbiased and consistent.
- The arrangement, or probability distribution of the predictor variables in \mathbf{X} has a major influence on the precision of estimates of β . Sampling and design of experiments are highly-developed subfields of statistics that provide guidance for collecting data in such a way to achieve a precise estimate of β .

12.5.2 Interpreting Multilinear Regression Models

A fitted linear regression model can be used to identify the relationship between a single predictor variable x_j and the response variable y when all the other predictor variables in the model are “held fixed” (Eq. 12.11). Specifically, the interpretation of β_j is the expected change in y for a one-unit change in x_j when the other covariates

⁶ [<ISP2e>/l2_LinearModels/anscombe/ISP_anscombe.py](https://ISP2e/l2_LinearModels/anscombe/ISP_anscombe.py)

are held fixed, that is, the expected value of the partial derivative of y with respect to x_j . This is sometimes called the *unique effect* of x_j on y . In contrast, the *marginal effect* of x_j on y can be assessed using a correlation coefficient or simple linear regression model relating x_j to y ; this effect is the total derivative of y with respect to x_j .

Care must be taken when interpreting regression results, as some of the regressors may not allow for marginal changes (such as dummy variables, or the intercept term), while others cannot be held fixed. (Recall the example from the polynomial fit in Sect. 12.2.2: it would be impossible to “hold t_j fixed” and at the same time change the value of t_i^2 .)

It is possible that the unique effect can be nearly zero even when the marginal effect is large. This may imply that some other covariate captures all the information in x_j so that once that variable is in the model, there is no contribution of x_j to the variation in y . Conversely, the unique effect of x_j can be large while its marginal effect is nearly zero. This would happen if the other covariates explained a great deal of the variation of y , but they mainly explain variation in a way that is complementary to what is captured by x_j . In this case, including the other variables in the model reduces the part of the variability of y that is unrelated to x_j , thereby strengthening the apparent relationship with x_j .

The meaning of the expression “held fixed” may depend on how the values of the predictor variables arise. If the experimenter directly sets the values of the predictor variables according to a study design, the comparisons of interest may literally correspond to comparisons among units whose predictor variables have been “held fixed” by the experimenter. Alternatively, the expression “held fixed” can refer to a selection that takes place in the context of data analysis. In this case, we “hold a variable fixed” by restricting our attention to the subsets of the data that happen to have a common value for the given predictor variable. This is the only interpretation of “held fixed” that can be used in an observational study.

The notion of a “unique effect” is appealing when studying a complex system where multiple interrelated components influence the response variable. In some cases, it can literally be interpreted as the causal effect of an intervention that is linked to the value of a predictor variable. However, it has been argued that in many cases multiple regression analysis fails to clarify the relationships between the predictor variables and the response variable when the predictors are correlated with each other and are not assigned following a study design (Kaplan, 2009).



Code: `ISP_simpleModels.py`⁷ shows an example.

⁷ [<ISP2e2/12_LinearModels/simpleModels/ISP_simpleModels.py>](https://ISP2e2/12_LinearModels/simpleModels/ISP_simpleModels.py).

12.6 Bootstrapping

Another type of modeling is *bootstrapping*. Sometimes we have data describing a distribution, but do not know what type of distribution it is. So what can we do if we want to find out, e.g., confidence values for the mean?

The answer is bootstrapping. Bootstrapping is a scheme of *resampling*, i.e., taking additional samples repeatedly from the initial sample, to provide estimates of its variability. In a case where the distribution of the initial sample is unknown, bootstrapping is of special help in that it provides information about the distribution.

The application of bootstrapping in Python is much facilitated by the package *scikits.bootstrap* by Constantine Evans (<https://github.com/cgevans/scikits-bootstrap>).



Code: `ISP_bootstrapDemo.py`⁸ Example of bootstrapping the confidence interval for the mean of a sample distribution.

12.7 Exercises

1. Peak Observations

Correlation

First read in the data for the average yearly temperature at the Sonnblick, Austria's highest meteorological observatory, from the file `data/data_others/avgtemp.xls`. Then calculate the pearson and spearman correlation, and kendall's tau, for the temperature verses year.

Regression

For the same data, calculate the yearly increase in temperature, assuming a linear increase with time. Is this increase significant?

Normality Check

For the data from the regression model, check if the model is OK by testing if the residuals are normally distributed (e.g., by using the Kolmogorov-Smirnov test).

2. Climate Crisis

- Read in the CO2-levels recorded on Mauna Loa in Hawaii, from https://www.esrl.noaa.gov/gmd/webdata/ccgg/trends/co2/co2_mm_mlo.txt, and remove the seasonal oscillations with the `statsmodels` function `seasonal_decompose`. (See Sect. 11.7.1)

⁸ <ISP2e>/12_LinearModels/bootstrapDemo/ISP_bootstrapDemo.py.

- Fit a linear, a quadratic, and a cubic curve to the data, using the *patsy* formula language.
- Plot the data, and superpose them with the resulting fits.
- Use the resulting AIC-values to find out which of these three curves provides the best-fit to the data.

Chapter 13

Generalized Linear Models



Data can be discrete for different reasons. One is that they were acquired in a discrete way (e.g., levels in a questionnaire). Another one is that the paradigm only gives discrete results (e.g., rolling dice). For the analysis of such data, we can build on the tools for the analysis of *ranked data* that have already been covered in the previous chapters. Extending this analysis to statistical models of ranked data requires the introduction of *Generalized Linear Models (GLMs)*. This chapter shows how to implement *logistic regression*, one frequently used application of GLMs, with the tools provided by *Python*.

13.1 Comparing and Modeling Ranked Data

Ordinal data have clear rankings, e.g., “none—little—some—much—very much”. However they are not continuous. For the analysis of such *rank ordered data* we can use the rank order methods described in Chap. 8:

Two groups When comparing three or more rank ordered groups, we can use the *Mann-Whitney test* (Sect. 8.2.4)

Three or more groups When comparing two rank ordered groups, we can use the *Kruskal-Wallis test* (Sect. 8.3.3)

A *hypothesis test* provides a value for the probability of a hypothesis. And *linear regression modeling* allows to make predictions and give confidence intervals for output variables that depend linearly on given inputs. But a large class of problems exceeds these requirements. For example, suppose we want to calculate the probability that a patient survives an operation, based on the amount of anesthetic he/she receives, and we want to find out how much anesthetic we can give the patient so that the chance of survival is at least 95%.

The answer to this question involves statistical modeling and the tool of *logistic regression*. If more than two ordinal (i.e., naturally ranked) levels are involved, the so-called *ordinal logistic regression* is used.

To cover such questions *Generalized Linear Models (GLMs)* have been introduced, which extend the technique of linear regression to a wide range of other problems. A general coverage of GLMs is beyond the goals of this book, and I would like to refer to the excellent book by Annette Dobson Dobson and Barnett (2018). While Dobson and Barnett only give solutions in *R* and *Stata*, I have developed *Python* solutions for almost all examples in that book (<https://github.com/thomas-haslwanter/dobson>).

In the following chapter I want to cover one commonly used case, *logistic regression*, and its extension to *ordinal logistic regression*. The *Python* solutions presented should allow the readers to solve similar problems on their own and should give a brief insight into Generalized Linear Models.

13.2 Elements of GLMs

Here only the general principles of *Generalized Linear Model (GLM)* will be described. For details please refer to the excellent book by Dobson and Barnett (2018).

A GLM consists of three elements:

1. A probability distribution from the *exponential family*.
2. A linear predictor $\eta = \mathbf{X} \cdot \beta$.
3. A link function g such that $E(Y) = \mu = g^{-1}(\eta)$.

13.2.1 Exponential Family of Distributions

The exponential family is a set of probability distributions of a certain form, specified below. This special form is chosen for mathematical convenience, on account of some useful algebraic properties, as well as for generality, as exponential families are in a sense very natural sets of distributions to consider. The exponential families include many of the most common distributions, including the normal, exponential, chi-squared, Bernoulli, Poisson distribution and many others. (A common distribution that is not from the exponential family is the t-distribution.)

In mathematical terms, a distribution from the exponential family has the general form

$$f_X(x|\theta) = h(x)g(\theta)\exp[\eta(\theta) \cdot T(x)], \quad (13.1)$$

where $T(x)$, $h(x)$, $g(\theta)$, $\eta(\theta)$, and $A(\theta)$ are known functions (g is different from the “link function” in the previous paragraph). While Eq. 13.1 is dauntingly abstract, it provides the theoretical basis for a common consistent treatment of a large number of different statistical models.

13.2.2 Linear Predictor and Link Function

The linear predictor for GLM is the same as the one used for *linear models*. The resulting terminology is unfortunately fairly confusing:

General Linear Models are models of the form $y = \mathbf{X} \cdot \boldsymbol{\beta} + \epsilon$, where ϵ is normally distributed (see Chap. 12).

Generalized Linear Models encompass a much wider class of models, including all distributions from the exponential family *and* a link function. The *linear predictor* $\eta = \mathbf{X} \cdot \boldsymbol{\beta}$ is now “only” an element of the distribution function any more, which provides the flexibility of GLMs.

The *link function* is an arbitrary function, with the only requirements that it is continuous and invertible.

13.3 GLM 1: Logistic Regression

So far we have been dealing with linear models, where a linear change on the input leads to a corresponding linear change on the output:

$$y = m * x + b + \epsilon . \quad (13.2)$$

However, for many applications this model is not suitable. Suppose we want to calculate the probability that a patient survives an operation, based on the amount of anesthetic he/she receives. This probability is bounded on both ends since it has to be a value between 0 and 1.

We can achieve such a bounded relationship, though, if we do not use the output of Eq. 13.2 directly, but wrap it by another function. Here we achieve this with the frequently used *logistic function*

$$p(y) = \frac{1}{1 + e^{\beta y + \alpha}} . \quad (13.3)$$

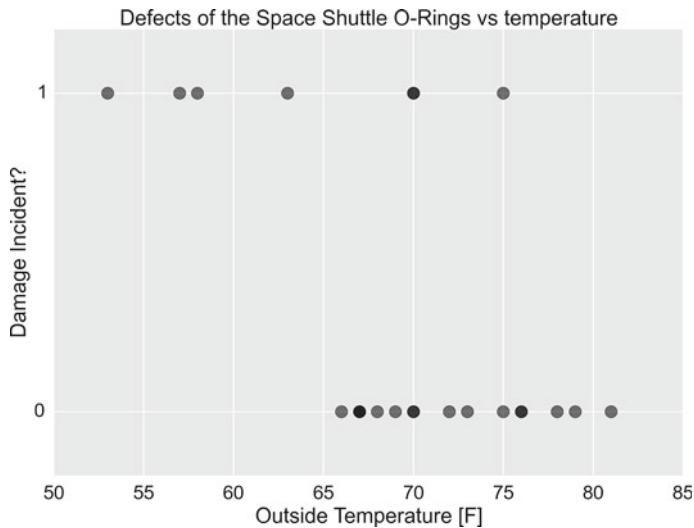


Fig. 13.1 Failure of O-rings during space shuttle launches, as a function of temperature

Example: The Challenger Disaster

A good example for logistic regression is the simulation of the probability of an O-ring failure as a function of temperature for space shuttle launches. Here we analyze it with a logistic regression model, whereas in the next chapter we will look at it with the tools of Bayesian modeling.

On January 28, 1986, the twenty-fifth flight of the U.S. space shuttle program ended in disaster when one of the rocket boosters of the Shuttle Challenger exploded shortly after lift-off, killing all seven crew members. The presidential commission on the accident concluded that it was caused by the failure of an O-ring in a field joint on the rocket booster, and that this failure was due to a faulty design that made the O-ring unacceptably sensitive to a number of factors including outside temperature. Of the previous 24 flights, data were available on failures of O-rings on 23, (one was lost at sea), and these data were discussed on the evening preceding the Challenger launch, but unfortunately only the data corresponding to the 7 flights on which there was a damage incident were considered important and these were thought to show no obvious trend (top row points in Fig. 13.1). However, the full set of data indicates a trend to O-ring failure with lower temperatures. The full data set is shown below:

To simulate the probability of the O-rings failing, we can use the *logistic function* (Eq. 13.3):

With a given p-value, the *binomial distribution* (Sect. 6.2.2) determines the probability-mass-function for a given number n of shuttle launches. This tells us how likely it is to have 0, 1, 2,... failures during those n launches.

Listing 13.1: L13_1_logitShort.py

```
# Import standard packages
import numpy as np
import os
import pandas as pd

# additional packages
from statsmodels.formula.api import glm
from statsmodels.genmod.families import Binomial

# Get the data
inFile = '..\data\challenger_data.csv'
challenger_data = np.genfromtxt(inFile, skip_header=1,
                                usecols=[1, 2], missing_values='NA',
                                delimiter=',')
# Eliminate NaNs
challenger_data = challenger_data[~np.isnan(challenger_data[:, 1])]

# Create a dataframe, with suitable columns for the fit
df = pd.DataFrame()
df['temp'] = np.unique(challenger_data[:,0])
df['failed'] = 0
df['ok'] = 0
df['total'] = 0
df.index = df.temp.values

# Count the number of starts and failures
for ii in range(challenger_data.shape[0]):
    curTemp = challenger_data[ii,0]
    curVal = challenger_data[ii,1]
    # the following lines find the current temperature in df, and
    # add one to the counts in 'total', and in 'failed' or 'ok'
    df.loc[curTemp, 'total'] += 1
    if curVal == 1:
        df.loc[curTemp, 'failed'] += 1
    else:
        df.loc[curTemp, 'ok'] += 1

# fit the model

# --- >>> START stats <<< ---
model = glm('ok + failed ~ temp', data=df, family=Binomial()).fit()
# --- >>> STOP stats <<< ---

print(model.summary())
```



Code: ISP_logisticRegression.py¹ shows the full code for Fig. 13.2.

¹ <ISP2e>/13_LogisticRegression/LogisticRegression/ISP_logisticRegression.py.

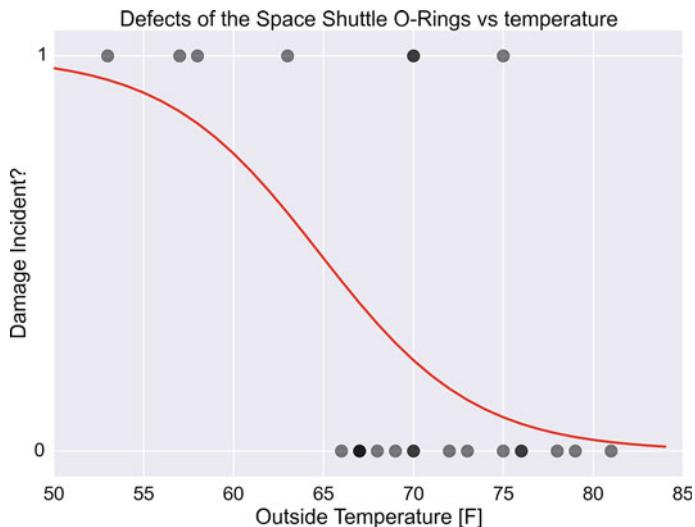


Fig. 13.2 Probability for O-ring failure (red line)

To summarize, we have three elements in our model:

1. a probability distribution, which determines the probability of the outcome for a given trial (here the binomial distribution);
2. a linear model that relates the co-variates (here the temperature) to the variates (the failure/success of an O-ring); and
3. a *link function* that wraps the linear model to produce the parameter for the probability distribution (here the logistic function).

13.4 GLM 2: Ordinal Logistic Regression

13.4.1 Model

Section 13.3 has shown one example of a GLM, logistic regression. In this section I want to show how further generalization, from a *yes/no* decision to a decision for one-of-many groups ($Group_1/Group_2/Group_3$), leads into the area of numerical optimization.

²The *logistic ordinal regression* model, also known as the proportional odds model, was introduced in the early 80s by (McCullagh 1980; McCullagh and Nelder 1989) and is a generalized linear model specially tailored for the case of predicting

²This section has been taken with permission from Fabian Pedregosa's blog on ordinal logistic regression, <http://fa.bianp.net/blog/2013/logistic-ordinal-regression/>.

ordinal variables, that is, variables that are discrete (as in classification) but which can be ordered (as in regression). It can be seen as an extension of the logistic regression model described above to the ordinal setting (Fig. 13.3).

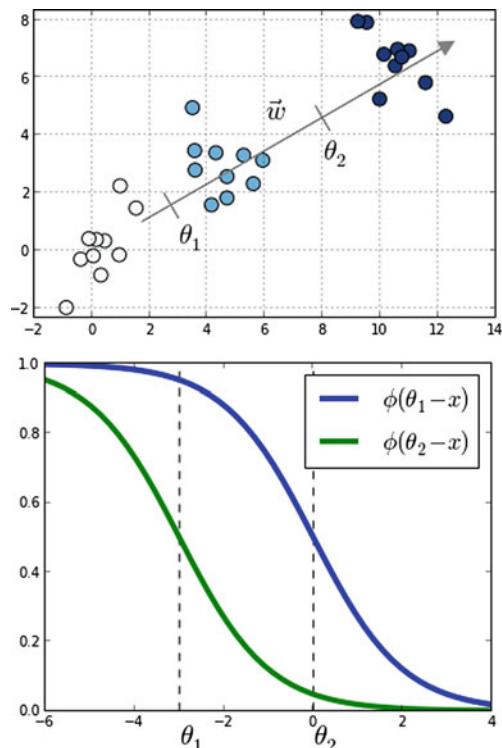
$$P(y \leq j | X_i) = \phi(\theta_j - w^T X_i) = \frac{1}{1 + \exp(w^T X_i - \theta_j)}, \quad (13.4)$$

where w and θ are vectors to be estimated from the data, and ϕ is the logistic function defined as $\phi(t) = \frac{1}{1 + \exp(-t)}$.

Compared to multiclass logistic regression, we have added the constraint that the hyperplanes that separate the different classes are *parallel* for all classes, that is, the vector w is common across classes. To decide to which class will X_i be predicted we make use of the vector of thresholds θ . If there are K different classes, θ is a non-decreasing vector (that is, $\theta_1 \leq \theta_2 \leq \dots \leq \theta_{K-1}$) of size $K - 1$. We will then assign the class j if the prediction $w^T X$ (recall that it is a linear model) lies in the interval $[\theta_{j-1}, \theta_j]$. In order to keep the same definition for extremal classes, we define $\theta_0 = -\infty$ and $\theta_K = +\infty$.

The intuition is that we are seeking a vector w such that $X \cdot w$ produces a set of values that are well separated into the different classes by the different thresholds θ_i . We choose a logistic function to model the probability $P(y \leq j | X_i)$, but other

Fig. 13.3 Toy example with three classes denoted in different colors. Also shown are the vector of coefficients w and the thresholds θ_0 and θ_1 . (Figure from Fabian Pedregosa, with permission)



choices are also possible. In the proportional hazards model McCullagh (1980) the probability is modeled as

$$-\log(1 - P(y \leq j | X_i)) = \exp(\theta_j - w^T \cdot X_i). \quad (13.5)$$

Other link functions are possible, where the link function satisfies
 $link(P(y \leq j | X_i)) = \theta_j - w^T \cdot X_i$.

Under this framework, the logistic ordinal regression model has a logistic link function, and the proportional hazards model has a log-log link function.

The logistic ordinal regression model is also known as the *proportional odds model* because the ratio of corresponding odds for two different samples X_1 and X_2 is $\exp(w^T \cdot (X_1 - X_2))$ and so does not depend on the class j but only on the difference between the samples X_1 and X_2 .

13.4.2 Optimization

Model estimation can be posed as an optimization problem. Here, we minimize the loss function for the model, defined as minus the log-likelihood:

$$\mathcal{L}(w, \theta) = - \sum_{i=1}^n \log(\phi(\theta_{y_i} - w^T \cdot X_i) - \phi(\theta_{y_i-1} - w^T \cdot X_i)). \quad (13.6)$$

In this sum all terms are convex on w , thus the loss function is convex over w . In the solution we will use the function `fmin_slsqp` in `scipy.optimize` to optimize \mathcal{L} under the constraint that θ is a non-decreasing vector.

Using the formula $\log(\phi(t))' = (1 - \phi(t))$, we can compute the gradient of the loss function as

$$\begin{aligned} \nabla_w \mathcal{L}(w, \theta) &= \sum_{i=1}^n X_i (1 - \phi(\theta_{y_i} - w^T \cdot X_i) - \phi(\theta_{y_i-1} - w^T \cdot X_i)) \\ \nabla_\theta \mathcal{L}(w, \theta) &= \sum_{i=1}^n e_{y_i} \left(1 - \phi(\theta_{y_i} - w^T \cdot X_i) - \frac{1}{1 - \exp(\theta_{y_i-1} - \theta_{y_i})} \right) \\ &\quad + e_{y_i-1} \left(1 - \phi(\theta_{y_i-1} - w^T \cdot X_i) - \frac{1}{1 - \exp(-(\theta_{y_i-1} - \theta_{y_i}))} \right), \end{aligned}$$

where e_i is the i -th canonical vector.

13.4.3 Performance

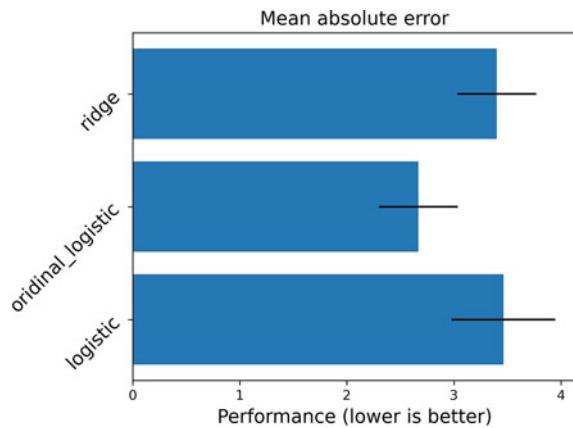
Fabian Pedregosa has compared the prediction accuracy of this model in the sense of mean absolute error on the Boston house-prices a data set commonly used in machine learning. To have an ordinal variable, he rounded the values to the closest integer, which resulted in a problem of size $506 * 13$ with 46 different target values. Although not a huge increase in accuracy, this model did give better results on this particular data set (Fig. 13.4):

Here, ordinal logistic regression is the best-performing model, followed by a Linear Regression model and a One-versus-All Logistic regression model as implemented in scikit-learn.



Code: `ISP_ordinalLogisticRegression.py`³ corresponding code by Fabian Pedregosa. It implements a *Python* version of the algorithm described above, using *scipy*'s `optimize.fmin_slsqp` function. This takes as arguments the loss function, the gradient denoted before and a function that is > 0 when the inequalities on θ are satisfied.

Fig. 13.4 Figure from Fabian Pedregosa, with permission



³ [ISP2e>/13_LogisticRegression/OrdinalLogisticRegression/ISP_ordinalLogisticRegression.py](https://ISP2e.readthedocs.io/en/latest/_modules/ISP_ordinalLogisticRegression/ISP_ordinalLogisticRegression.py).

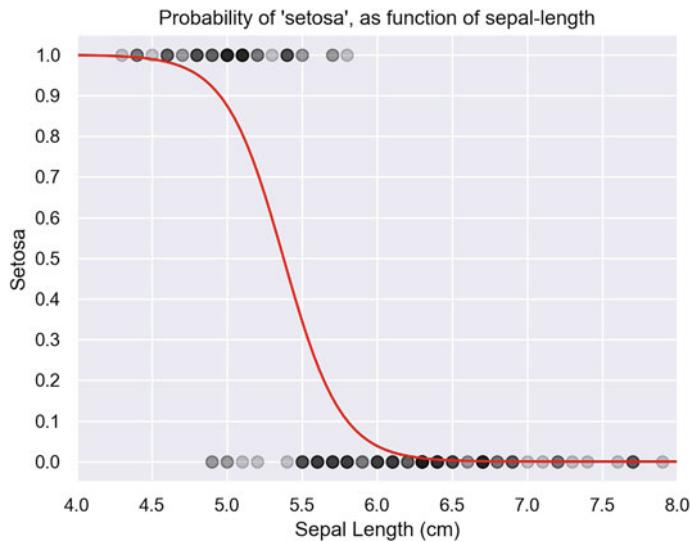


Fig. 13.5 Chance of having a flower of the species *iris setosa*, as a function of the sepal length

13.5 Exercises

1. Finding the Right Flower

- Take the *iris-data* (see Sect. 11.2.1), and plot for which petal_length the measured flowers are of the species *iris setosa* (see Fig. 13.5).
- Fit a logistic regression curve to the data, and superpose it on the plot.
- Find the largest sepal length where you still have a 10% chance of having a *iris setosa*.

Chapter 14

Bayesian Statistics



Calculating probabilities is only one part of statistics. Another is the interpretation of them—and the consequences that come with different interpretations.

So far we have restricted ourselves to the *frequentist interpretation*, which interprets p as the *frequency of an occurrence*: if an outcome of an experiment has the probability p , it means that if that experiment is repeated N times (where N is a large number), then we observe this specific outcome $N * p$ times. Or in other words: given a certain model, we look at the likelihood to find the observed set of data.

The *Bayesian interpretation* of p is quite different and interprets p as our *belief of the likelihood* of a certain outcome. Here we take the observed data as fixed and look at the likelihood to find certain model parameters. For some events, this makes a lot more sense. For example, a presidential election is a one-time event, and we will never have a large number N repetitions.

14.1 Bayesian Versus Frequentist Interpretation

14.1.1 Bayes' Theorem

In addition to this difference in interpretation, the Bayesian approach has another advantage: it lets us bring in *prior knowledge* into the calculation of the probability p , through the application of *Bayes' Theorem*:

In its most common form, it is

$$P(A|B) = \frac{P(B|A) P(A)}{P(B)}. \quad (14.1)$$

In the Bayesian interpretation, probability measures a degree of belief. Bayes' theorem then links the degree of belief in a proposition before and after accounting for evidence. For example, suppose it is believed with 50% certainty that a coin is twice as likely to land heads than tails. If the coin is flipped a number of times and the outcomes observed, that degree of belief may rise, fall or remain the same depending on the results.

John Maynard Keynes, a great economist and thinker, said “When the facts change, I change my mind. What do you do, sir?” This quote reflects the way a Bayesian updates his or her beliefs after seeing evidence.

For proposition A and evidence B,

- $P(A)$, the *prior probability*, is the initial degree of belief in A.
- $P(A|B)$, the *posterior probability*, is the degree of belief having accounted for B.
It can be read as “*the probability of A, given that B is the case*”.
- the quotient $P(B|A)/P(B)$ represents the support B provides for A.

If the number of available data points is large, the difference in interpretation does typically not change the result significantly. If the number of data points is small, however, the possibility to bring in external knowledge may lead to a significantly improved estimation of p .

14.1.2 Bayesian Example

Suppose a man told you he had a nice conversation with someone on the train. Not knowing anything about this conversation, the probability that he was speaking to a woman is 50% (assuming the speaker was as likely to strike up a conversation with a man as with a woman). Now suppose he also told you that his conversational partner had long hair. It is now more likely he was speaking to a woman, since women are more likely to have long hair than men. Bayes' theorem can be used to calculate the probability that the person was a woman.

To see how this is done, let W represent the event that the conversation was held with a woman and L denote the event that the conversation was held with a long-haired person. It can be assumed that women constitute half the population for this example. So, not knowing anything else, the probability that W occurs is $P(W) = 0.5$.

Suppose it is also known that 75% of women have long hair, which we denote as $P(L|W) = 0.75$ (read: the probability of event L given event W is 0.75, meaning that the probability of a person having long hair (event “L”), given that we already know that the person is a woman (“event W”) is 75%). Likewise, suppose it is known that 15% of men have long hair, or $P(L|M) = 0.15$, where M is the complementary event of W , i.e., the event that the conversation was held with a man (assuming that every human is either a man or a woman).

Our goal is to calculate the probability that the conversation was held with a woman, given the fact that the person had long hair, or, in our notation, $P(W|L)$. Using the formula for Bayes' theorem, we have

$$P(W|L) = \frac{P(L|W)P(W)}{P(L)} = \frac{P(L|W)P(W)}{P(L|W)P(W) + P(L|M)P(M)},$$

where we have used the *law of total probability* to expand $P(L)$. The numeric answer can be obtained by substituting the above values into this formula (the algebraic multiplication is annotated using “*”). This yields

$$P(W|L) = \frac{0.75 * 0.50}{0.75 * 0.50 + 0.15 * 0.50} = \frac{5}{6} \approx 0.83,$$

i.e., the probability that the conversation was held with a woman, given that the person had long hair, is about 83%.

Another way to do this calculation is as follows. Initially, it is equally likely that the conversation is held with a woman as with a man, so the prior odds are 1:1. The respective chances that a man and a woman have long hair are 15% and 75%. It is 5 times more likely that a woman has long hair than that a man has long hair. We say that the likelihood ratio or Bayes factor is 5:1. Bayes' theorem in odds form, also known as *Bayes' rule*, tells us that the posterior odds that the person was a woman is also 5:1 (the prior odds, 1:1, times the likelihood ratio, 5:1). In a formula:

$$\frac{P(W|L)}{P(M|L)} = \frac{P(W)}{P(M)} \cdot \frac{P(L|W)}{P(L|M)}.$$

14.2 The Bayesian Approach in the Age of Computers

Bayes' theorem was named after the Reverend Thomas Bayes (1701–1761), who studied how to compute a distribution for the probability parameter of a binomial distribution. So it has been around for a long time. The reason Bayes' theorem has become so popular in statistics in recent years is the cheap availability of massive computational power. This allows the empirical calculation of posterior probabilities, one-by-one, for each new piece of evidence. This, combined with statistical approaches like Markov-Chain-Monte-Carlo simulations, has allowed radically new statistical analysis procedures and has led to what may be called “statistical trench warfare” between the followers of the different philosophies. If you do not believe me, check the corresponding discussions on the WWW.

For more information on that topic, check out (in order of rising complexity)

- Wikipedia, which has some nice explanations under “*Bayes ...*”
- Bayesian Methods for Hackers
[C. Davidson Pilon, ‘Practical introduction to pyMC’](#), see Sect. ‘Web Resources’ in p. 323.
- The PyMC User Guide (<http://pymc-devs.github.io/pymc/>): PyMC is a very powerful Python package which makes the application of MCMC techniques very simple.
- *Pattern Classification* does not avoid the mathematics, but uses it in a practical manner to help you gain a deeper understanding of the most important machine learning techniques (Duda et al. 2004).
- *Pattern Recognition and Machine Learning*, a comprehensive, but often quite technical book by Christopher Bishop (2007).

14.3 Example: Markov-Chain-Monte-Carlo Simulation

In the following we will re-analyze the data from the Challenger disaster already used in the previous chapter, but this time with a Markov-Chain-Monte-Carlo (MCMC) simulation. (This chapter is an excerpt of the excellent ‘Bayesian Methods for Hackers’ (Pilon 2015, with permission from the author).

The data are again from the Challenger disaster (see Sect. 13.3). To perform the simulation, we are going to use *PyMC*, a Python module that implements Bayesian statistical models and fitting algorithms, including MCMC simulations (<http://pymc-devs.github.io/pymc/>). Its flexibility and extensibility make it applicable to a large suite of problems. Along with core sampling functionality, PyMC includes methods for summarizing output, plotting, goodness-of-fit and convergence diagnostics.

PyMC provides functionalities to make Bayesian analysis as painless as possible. Here is a short list of some of its features:

- Fits Bayesian statistical models with Markov chain Monte Carlo and other algorithms.
- Includes a large suite of well-documented statistical distributions.
- Includes a module for modeling Gaussian processes.
- Creates summaries including tables and plots.
- Traces can be saved to the disk as plain text, Python pickles, SQLite or MySQL database, or HDF5 archives.
- Extensible: easily incorporates custom step methods and unusual probability distributions.
- MCMC loops can be embedded in larger programs, and results can be analyzed with the full power of Python.

To simulate the probability of the O-rings failing, we need a function that goes from one to zero. We again apply the logistic function:

$$p(t) = \frac{1}{1 + e^{\beta t + \alpha}}.$$

In this model, the variable β that describes how quickly the function changes from 1 to 0, and α indicates the location of this change.

Using the Python package PyMC, a Monte Carlo simulation of this model can be done remarkably easily:

```
# --- Perform the MCMC-simulations ---
temperature = challenger_data[:, 0]
D = challenger_data[:, 1] # defect or not?

# Define the prior distributions for alpha and beta
# 'value' sets the start parameter for the simulation
# The second parameter for the normal distributions is the
# "precision", i.e. the inverse of the standard deviation
beta = pm.Normal("beta", 0, 0.001, value=0)
alpha = pm.Normal("alpha", 0, 0.001, value=0)

# Define the model-function for the temperature
@pm.deterministic
def p(t=temperature, alpha=alpha, beta=beta):
    return 1.0 / (1. + np.exp(beta * t + alpha))

# connect the probabilities in `p` with our observations
# through a Bernoulli random variable.
observed = pm.Bernoulli("bernoulli_obs", p, value=D,
                        observed=True)

# Combine the values to a model
model = pm.Model([observed, beta, alpha])

# Perform the simulations
map_ = pm.MAP(model)
map_.fit()
mcmc = pm.MCMC(model)
mcmc.sample(120000, 100000, 2)
```

From this simulation, we obtain not only our best estimate for α and β but also information about our uncertainty about these values (Fig. 14.1).

The probability curve for an O-ring failure thus looks as follows (Fig. 14.2).

One advantage of the MCMC simulation is that it also provides confidence intervals for the probability (Fig. 14.3):

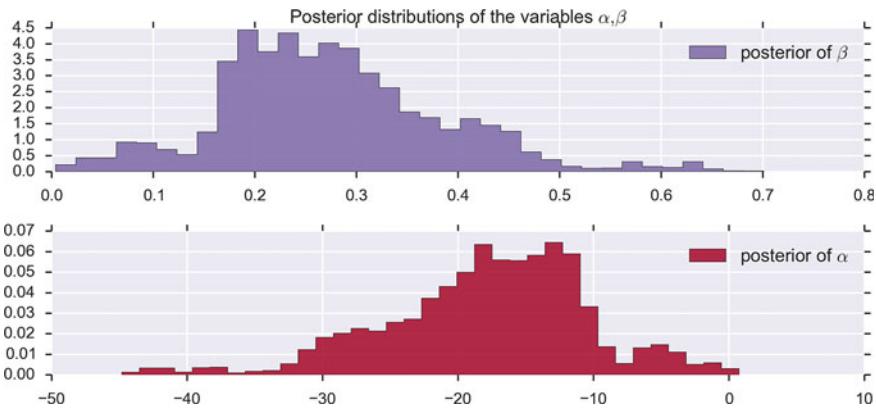


Fig. 14.1 Probabilities for alpha and beta, from the MCMC simulations

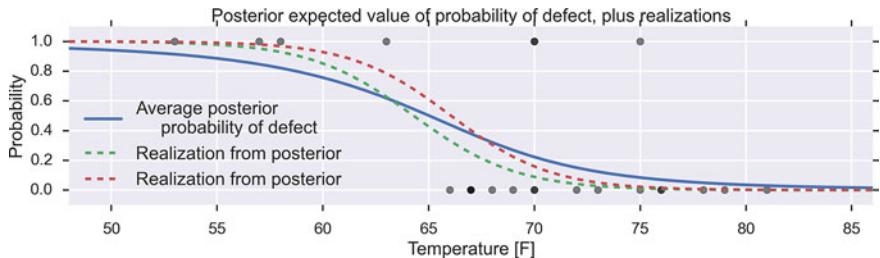


Fig. 14.2 Probability for an O-ring failure, as a function of temperature

On the day of the Challenger disaster, the outside temperature was 31 degrees Fahrenheit. The posterior distribution of a defect occurring, given this temperature, almost guaranteed that the Challenger was going to be subject to defective O-rings.

 **Code:** `ISP_bayesianStats.py`¹ Full implementation of the MCMC simulation.

14.4 Summing Up

The Bayesian approach offers a natural framework to deal with parameter and model uncertainty and has become very popular, especially in areas like machine learning. However, it is computationally much more intensive and therefore typically implemented with the help of existing tools like *PyMC* or *scikit-learn*.

¹ ISP2e/14_Bayesian/bayesianStats/ISP_bayesianStats.py.

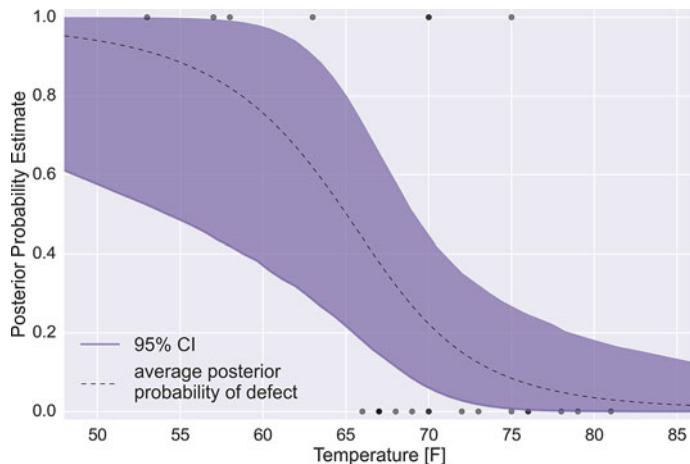


Fig. 14.3 95% Confidence Intervals for the Probability for an O-ring failure

This also shows one of the advantages of free and open languages like *Python*: they provide a way to build on the existing work of the scientific community and only require your enthusiasm and dedication. Therefore I would like to finish this book by thanking the Python community for the incredible amount of work that they have put into the development of core Python and Python packages. And I hope that this book allows you to share some of my enthusiasm for this vibrant language.

Appendix A

Useful Programming Tools

This chapter contains information on how to refine your programming skills. It will help you to get most value out of the time you invest in programming and ensures that your programs can be re-used and adapted later on.

A.1 Debugger

A *debugger* is a tool that lets you interrupt the program execution at chosen locations in your program or if an error occurs. This can be tremendously helpful for finding errors and solving problems. On the downside, additional files need to be loaded in order to *run the debugger*, versus simply *executing the code*. Take care that you distinguish between these two modes, since the debugging mode can be significantly slower.

The debugger that comes with Python, `pdb`, is a bit cumbersome to use. Instead you should learn to use the debugger of one of the IDEs presented in Sect. 2.3.5.

Tip

Take the time to get to know the debugger in the IDE that you are using. This will save you a lot of time later on when developing programs.

The file `<ISP2e>/App_DebugDemo/ISP_debug_demo.py` shows an example that triggers an error. The first Exercise in this Section uses that code sample.

A.2 Test Tools

Testing is arguably the most underestimated aspect of professional coding. But testing is indispensable for the generation of trustworthy programs. Different frameworks can be used for testing source code. A commonly used framework are *function-based unit tests* or short “*unittests*”. Function-based tests subscribe to the *xUnit* testing philosophy. One advantage of *unittests* is that they are already incorporated in the core Python packages (<https://docs.python.org/3/library/unittest.html>). Another popular framework is *nose*, which extends *unittests* (<https://nose.readthedocs.io/>).

But probably the easiest way to start with testing in Python is *pytest* (<https://pytest.org>). Thereby it is worth noting that *unittests* and *nose* test suites can be also run by *pytest*. The *pytest* framework makes it easy to write small tests yet scales to support complex functional testing for applications and libraries.

To show the principle behind testing, let me give an example of a very simple test in the file `test_sample.py`:

```
# content of test_sample.py
def inc(x):
    """The function to be tested. Increments inputs by 1."""
    return x + 1

def test_answer():
    """The test to check that the function 'inc' provides the
       correct result."""
    assert inc(4) == 5
```

The code containing the function and the testing can also be split into two separate files. To demonstrate the effects of a coding error I generate a second file `functions.py`, containing

```
def inc(x):
    """ Increments inputs by 1. Is correct. """
    return x + 1

def dec(x):
    """ Is supposed to decrement inputs by 1.
        Contains a mistake """
    return x - 2
```

and a third file called `test_function.py`, that tests the functions in `functions.py` with

```
import functions as fcn

def test_inc():
    """ The test to check that the function 'inc' provides the
        correct result. """
    assert fcn.inc(4)==5
```

```
def test_dec():
    """ The test to check that the function 'dec' provides the
    correct result. """
    assert fcn.dec(4)==3
```

With the three files (`test_sample.py`, `functions.py`, `test_functions.py`) in one folder, we can open that folder in a command-line terminal and simply type `pytest`. This results on my computer in

```
===== test session starts =====
platform win32 -- Python 3.7.6, pytest-5.3.2, py-1.8.0, pluggy-0.13.1
rootdir: D:\Users\thomas\Data\CloudStation\Books\sapy\testing
plugins: hypothesis-4.53.3
collected 3 items

test_functions.py .F
test_sample.py .

===== FAILURES =====
_____
test_dec _____
```

`def test_dec():
 """ The test to check that the function 'dec' provides the
 correct result. """
> assert fcn.dec(4)==3
E assert 2 == 3
E + where 2 = <function dec at 0x000001A2881E04C8>(4)
E + where <function dec at 0x000001A2881E04C8> = fcn.dec

test_functions.py:13: AssertionError
===== 1 failed, 2 passed in 0.06s =====`

`pytest` goes through all files in the folder, tries to recognize the files to be tested by their name, and runs the corresponding tests. The number of “.” in the output after the test-name indicate how many tests were run successfully in that module (one successful test in `test_functions.py`, and one in `test_sample`). And “F” indicates tests that failed (one in `test_functions.py`). The final message `1 failed, 2 passed` summarizes the test results.

A.3 Code Versioning with *git*

A.3.1 Overview

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

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 introduction 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/>.

A source of confusion can be the difference between “git” and “github”:

git is a version control program and is well integrated in most Python IDEs.

github is a website (<https://github.com/>) frequently used to share code and is now owned by Microsoft. It is the place where the source code for the majority of open source Python packages is hosted. While one can also download source code from there, it is more efficient to use *git* for this task.

Under Windows *tortoisesgit* (<https://tortoisegit.org/>) provides a very useful Windows shell interface for *git*. (Note that *git* and *tortoisesgit* have to be installed separately!) An overview of GUI-clients for *git* for the different platforms can be found under

<https://git-scm.com/downloads/guis>.

A.3.2 Installation and Interfaces

git can be downloaded for free from <https://git-scm.com/>. There one also finds very good documentation and also help getting started with *git*. Once it is installed it can be run in different ways:

- It can be run from a graphical user interface (GUI). Numerous GUIs exist (see <https://git-scm.com/downloads/guis>). Note that *git* has to be installed separate from the GUI interfaces.
- It can be run from the command-line. If you use the standard command-line tool, you first have to ensure that `git.exe` is part of the system path.
- *git for windows* <https://gitforwindows.org/> also comes with a *git Bash*, which provides a Bash emulation used to run *git* from the command line.¹ Unix and Linux users should feel right at home, as the Bash emulation behaves just like the `git` command in Linux and Unix environments.

¹ *Bash* is a Unix shell and command language.

A.3.3 Examples

a) TortoiseGit

In order to clone a repository, e.g.,

<https://github.com/thomas-haslwanger/statsintro-python-2e>

in *tortoisegit* from *github* to your computer, you simply have to right-click on the folder 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!

b) Command-line

You can (i) start a new git repository, (ii) add the file `test.txt` to this repository, and (iii) commit this file with the following command sequence:

```
git init  
git add test.txt  
git commit -a -m 'This is the first commit'
```

The options for `git commit` specify “to commit all staged files” (`-a`), with the message “This is the first commit” (`-m "This is the first commit"`).

Cloning an existing repository is even simpler: to obtain for example a copy of the repository that goes with this book, simply go to the directory where you want to have it and type

```
git clone https://github.com/thomas-haslwanger/statsintro-python-2e.git
```

A.4 Graphical User Interfaces (GUIs)

Many applications become a lot more accessible to the user if they provide a GUI. There are two reasons for that: (1) Many people are scared of the command-line, for the simple reason that they know neither how to get help, nor what to type next. (2) GUIs reduce the number of choices, making it more obvious to the user what can and/or should be done next.

One can start to program a GUI by using a GUI framework, such as *tkinter*, *Wx*, or *Qt* directly. However, this approach should be restricted to programmers who already have experience in object-oriented programming and in user interface design. The second option is to use a package that simplifies the use of GUIs. While *PySimpleGUI* is a rather young package, it offers a good balance between usability and power.

The most common tasks for the user interface are

- selecting an existing file for data input,
- selecting a new (or existing) file for data output, and
- selecting a directory.

The following section shows how to complete these tasks with *PySimpleGUI*. For other applications, check out <https://github.com/PySimpleGUI/PySimpleGUI>.

A.4.1 PySimpleGUI—Examples

a) Selecting an Existing File

This can be done with

```
import PySimpleGUI as sg

layout = [[sg.Text('Filename')], [sg.Input(), sg.FileBrowse()], [sg.OK(), sg.Cancel()]]

window = sg.Window('Get filename example', layout)

event, values = window.Read()
```

The parameter `event` is '`OK`' or '`Cancel`'. And `values` is a Python dictionary containing the filename selected and the filename in the text window (which is usually the same as the file selected) at the time the "OK" or "Cancel" button was pushed.

Directory, file name, and extension can be extracted with the standard Python package `pathlib`:

```
import pathlib
file_name = values[0]
path = pathlib.Path(file_name)

directory = path.parent    # e.g. WindowsPath('C:/Users/thomas
                           /Python')
full_file = path.name      # e.g. 'test.py'
extension = path.suffix    # e.g. '.py'
```

If instead of the interface in Fig. A.1 one prefers a file-browser for file selection (see Fig. A.2), the following command can be used:

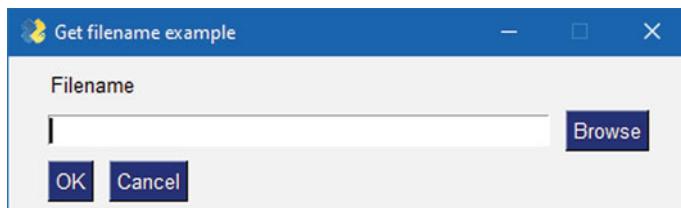


Fig. A.1 File Dialog from PySimpleGUI

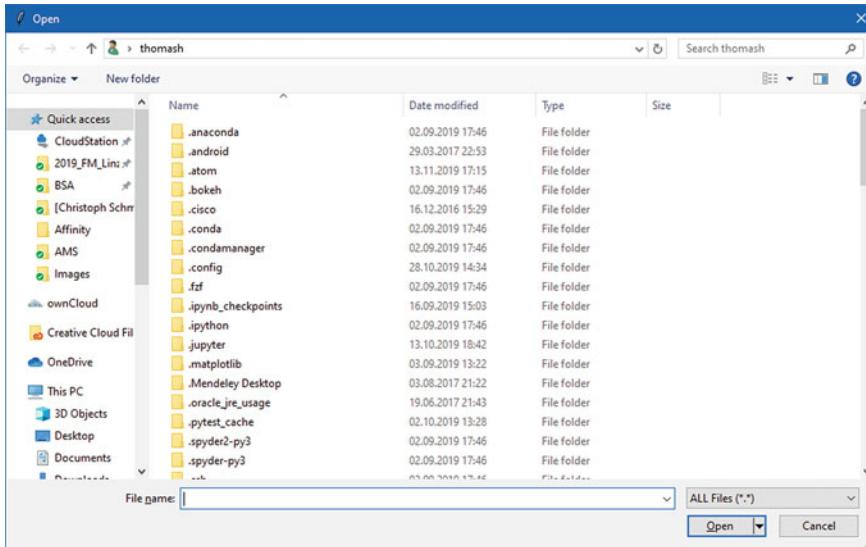


Fig. A.2 PySimpleGUI File Browser for file selection

```
import PySimpleGUI as sg
filename = sg.popup_get_file(' ', no_window=True)
```

b) Selecting an Output File

... is exactly the same procedure as above for selecting an input file, only with `FileBrowse` replaced by `FileSaveAs`.

c) Selecting a Directory

... is again exactly the same procedure as above, only with `FileBrowse` replaced by `FolderBrowse` (and to be nice to the user, you should probably also replace '`Filename`' with '`Foldername`').

d) Embedding Matplotlib

Interfaces with PySimpleGUI can also incorporate Matplotlib figures (Fig. A.3). (See also the corresponding Exercise for this chapter.)

A.4.2 PyQtGraph

PyQtGraph (<http://pyqtgraph.org/>) is a pure Python graphics and GUI library built on *PyQt5/PySide* and *numpy*. It is intended for use in mathematics, scientific, and engineering applications. Despite being written entirely in Python, the library is very fast due to its heavy leverage of *numpy* for number crunching and *Qt's Graphics View Framework* for fast display. This makes it very useful for the real-time display of signals.

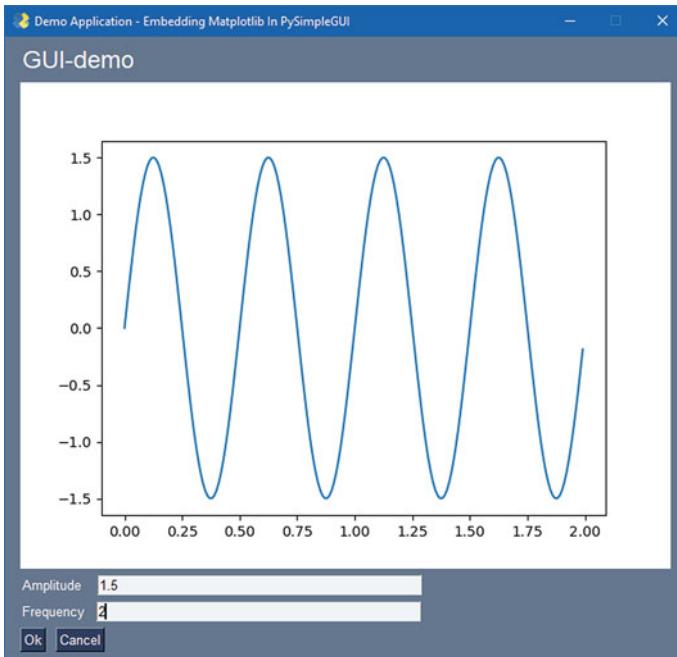


Fig. A.3 Embedding Matplotlib in PySimpleGUI. See also Exercise 2 below

The following command gives a quick introduction to the capabilities of *PyQtGraph*:

```
from pyqtgraph import examples
examples.run()
```

For example, the different line- and scatter-plots available in *PyQtGraph* are shown in Fig. A.4.

A.4.3 *Tips for User Interface*

Development of a GUI should proceed systematically: first gather information about the user and think about the requirements; then design and implement the GUI; and afterwards, have a real user test it before the final release.

- Before you start, think about the people who will use the interface: what do they really need, what do they know, and what are the work processes in which it will be used?
 - Is this program used on a daily basis? Or is it just once-in-a while?
 - Who is using the program? Is it you (or some other expert in the field)? Or is it someone who has no experience in the application?

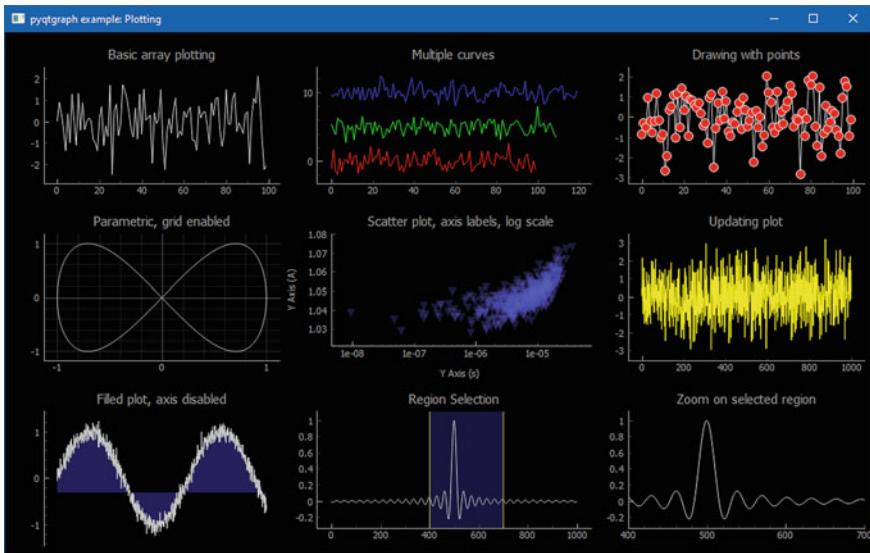


Fig. A.4 An overview of line- and scatter-plots in *PyQtGraph*

- Next, spend some time designing it using simple, quick tools. A pen and paper, or paper and scissors, are excellent starting points!
 - If the interface is for experts, keyboard-shortcuts should be used to accelerate frequently used tasks; if it is intended for novices, buttons are preferable.
 - If the interface is for daily use, it should be optimized for efficiency, e.g., by implementing keyboard shortcuts for frequently used tasks; if it is used just once in a while, the interface should be as self-explanatory as possible, and buttons are preferable to keyboard shortcuts.
 - When designing the GUI it is particularly important that it fits the expectations of the user, so it helps to make it similar to user interfaces that people already know.
- If the user interface is not intended for yourself, it usually pays off to have a user quickly test it. I am amazed again and again that there may be a word that they do not understand or an element they do not see.

A.5 Exercises

1. **Using a Debugger** Open the code-quantlet [`<ISP2e>/App_DebugDemo/ISP_debug_demo.py`](#) in an editor/debugger of your choice and proceed to the first error. At the point of error, check where you are in the main program, and where and in which function the error occurs. Check the

local and global variables at the point of error and try to execute a few python commands—using these variables—in the debugger.

2. **Construction of a GUI (hard)** create a graphical user interface (GUI, see last chapter) for this a function that draws a sine-wave for two seconds, where “Amplitude” and “Frequency” can be entered in the GUI, as shown in Fig. A.3.

Appendix B

Solutions

Problems of Chap. 2

Data Input

Listing B.1: S2_translatation.py

```
""" Solution to Exercise 'Translation', Chapter 'Python' """

# author: Thomas Haslwanter
# date: June-2022

# Import the required packages
import numpy as np
import matplotlib.pyplot as plt

# Define the original points
p_0 = [0,0]
p_1 = [2,1]

# Combine them to an array
array = np.array([p_0, p_1])
print(array)

# Translate the array
translated = array + [3,1]
print(translated)

# Plot the data
plt.plot(array[:,0], array[:,1], label='original')
plt.plot(translated[:,0], translated[:,1], label='translated')

# Format and show the plot
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.show()
```

Listing B.2: S2_rotation.py

```
""" Solution to Exercise 'Rotation', Chapter 'Python' """

# author: Thomas Haslwanter
# date: June-2022

# Import the required packages
import numpy as np
import matplotlib.pyplot as plt


def rotate_me(in_vector:np.ndarray, alpha:float) -> np.ndarray:
    """Function that rotates a vector in 2 dimensions

    Parameters
    -----
    in_vector : vector (2,) or array (:,2)
        vector(s) to be rotated
    alpha : rotation angle [deg]

    Returns
    -----
    rotated_vector : vector (2,) or array (:,2)
        rotated vector

    Examples
    -----
    perpendicular = rotate_me([1,2], 90)

    """
    alpha_rad = np.deg2rad(alpha)
    R = np.array([[np.cos(alpha_rad), -np.sin(alpha_rad)],
                  [np.sin(alpha_rad), np.cos(alpha_rad)]])
    return R @ in_vector


if __name__ == '__main__':
    vector = [2,1]
    # Draw a green line from [0,0] to [2,1]
    plt.plot([0,vector[0]], [0, vector[1]], 'g',
              label='original')

    # Coordinate system
    plt.hlines(0, -2, 2, linestyles='dashed')
    plt.vlines(0, -2, 2, linestyles='dashed')

    # Make sure that the x/y dimensions are equally drawn
    cur_axis = plt.gca()
    cur_axis.set_aspect('equal')

    # Rotate the vector
    rotated = rotate_me(vector, 25)
    plt.plot([0, rotated[0]], [0 ,rotated[1]],
              label='rotated',
              color='r',
              linewidth=3)

    plt.legend()
    plt.show()
```

Listing B.3: S2_taylor.py

```
""" Solution to Exercise 'Taylor', Chapter 'Python' """

# author: Thomas Haslwanter
# date: June-2022

# Import the required packages
import numpy as np
import matplotlib.pyplot as plt
from typing import Tuple


def approximate(angle:np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
    """Function that calculates a second order approximation
    to sine and cosine

    Parameters
    -----
    angle : angle [deg]

    Returns
    -----
    approx_sine : approximated sine
    approx_cosine : approximated cosine

    Examples
    -----
    alpha = 0.1
    sin_ax, cos_ax = approximate(alpha)

    Notes
    -----
    Input can also be a single float

    """
    sin_approx = angle
    cos_approx = 1 - angle**2/2

    return (sin_approx, cos_approx)


if __name__ == '__main__':
    limit = 50          # [deg]
    step_size = 0.1      # [deg]

    # Calculate the data
    theta_deg = np.arange(-limit, limit, step_size)
    theta = np.deg2rad(theta_deg)
    sin_approx, cos_approx = approximate(theta)

    # Plot the data
    plt.plot(theta_deg, np.column_stack((np.sin(theta),
                                         np.cos(theta))), label='exact')
    plt.plot(theta_deg,
              np.column_stack((sin_approx, cos_approx)),
              linestyle='dashed',
              label='approximated')
    plt.legend()
    plt.xlabel('Angle [deg]')
    plt.title('sine and cosine')
    out_file = 'approximations.png'
```

```

plt.savefig(out_file, dpi=200)
print(f'Resulting image saved to {out_file}')

plt.show()

```

Listing B.4: S2_pandas.py

```

""" Solution to Exercise "First Steps with pandas":
Generate a sine and cosine wave using pandas' DataFrames,
and write them to an out-file.
"""

# author: Thomas Haslwanter, date: June-2022

import numpy as np
import pandas as pd

# Set the parameters
rate = 10
dt = 1/rate
freq = 1.5

# Derived quantities
omega = 2*np.pi*freq

# Generate the data
t = np.arange(0,10,dt)
y = np.sin(omega*t)
z = np.cos(omega*t)

# Assemble them in a DataFrame
df = pd.DataFrame({'Time':t, 'YVals':y, 'ZVals':z})

# Show the top 5 values
print(df.head())

# Save lines 10-15 of the y- and z-values to an outfile
outfile = 'out.txt'
df[10:16][['YVals', 'ZVals']].to_csv(outfile)
print('Data written to {}'.format(outfile))
input('Done')

```

Problems of Chap. 3

Reading in Data

Listing B.5: S3_data_gen.py

```

""" Generation of data for Chapter 'Data Input'

It shows how to generate
- formmated text-strings
- CSV files           -> 'data.csv'
- otherwise formmated TXT-files   -> 'data_tab.txt',
                                    'data_modified.txt'
- Excel files          -> 'data.xls'
- Matlab files         -> 'data.mat'
- Binary data          -> 'data.raw'

Requires the package "xlwt"
"""

```

```
# author: Thomas Haslwanter, date: June-2022

# Import the required packages
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

def save_txt(df: pd.DataFrame, out_file='data.csv') -> None:
    """ Save data to ASCII-format

    Parameters
    -----
    df : Input data
    out_file : Output file; two other ASCII-files with same
               stem are also generated
    """

    # Saves the data to CSV-format, which means by default
    # - Separated by a comma
    # - With a column name
    # - With a running index on the left side
    df.to_csv(out_file)

    # Always let the user know when you generate a new file!!
    # If you use Python >3.7, you can use the "format-strings"
    print(f'Data have been saved in CSV-format to {out_file}')

    # For earlier versions of Python you have to use
    # print(f'CSV-Data have been saved to {out_file}')

    # Simple file, tab-separated, no header, no index
    simple_file = out_file.replace('.csv', '_tab.txt')
    df.to_csv(simple_file, sep='\t', header=False, index=False)
    print(f'Data have been saved to {simple_file}')

    # Show how to add a file-header to an existing text file
    with open(out_file, 'r') as original:
        text = original.read()

    modified_file = out_file.replace('.csv', '_modified.txt')
    with open(modified_file, 'w') as modified:
        modified.write('This file was generated on Sept 19\n')
        modified.write(f'Sampling rate: {rate} [Hz]\n')
        modified.write(text)
    print(f'A file header has been added to {out_file}, '+
          f'and the new file saved as {modified_file}')


def save_xls(df: pd.DataFrame, out_file='data.xls') -> None:
    """ Save data to MS Excel-format

    Parameters
    -----
    df : Input data
    out_file : Name of output file
    """

    # Save data to Excel-format
    df.to_excel(out_file, index=False)
    print(f'MS-Excel data have been saved to {out_file}')
```

```

def save_matlab(t: np.ndarray, data: np.ndarray,
                 out_file='data.mat') -> None:
    """ Save data to Matlab-format

    Parameters
    -----
    t : Time-values [sec]
    data : sine-wave
    out_file : Name of output file
    """

    # Saving data to Matlab-format requires "scipy.io", ...
    from scipy.io import savemat

    # ... and we have to put the data into a Python-dictionary
    # For this example I add an information-text and format
    # the data as matrix
    data_mat = np.column_stack( (t, data) )
    data_dict = {'data': data_mat,
                 'info':'These are demo-data, showing a sine-wave'}

    savemat(out_file, data_dict)
    print(f'Matlab data have been saved to {out_file}')


def generate_binary(out_file='data.raw') -> None:
    """Generate binary data, with an ASCII-header with
    256 byte, and three columns of data.

    Parameters
    -----
    out_file : name of outfile
    """

    # To generate a more interesting signal I produce a "chirp"
    from scipy.signal import chirp

    # Set the parameters
    length_header = 256      # byte

    # Generate a dummy header text
    txt = """This is the header.

    It has a length of 'length_header' byte. After the text,
    it is padded with whitespaces."""
    out_txt = txt + ' '*(length_header-len(txt))

    # Write it to the out_file
    fh = open(out_file, 'wb')
    fh.write(out_txt.encode())

    # Generate some data
    t = np.arange(0,20, 0.1)
    x = np.sin(t)
    y = chirp(t, 3, np.max(t), 0.01)
    data = np.column_stack((t, np.sin(t), y*t))

    # Also write them to a file
    fh.write(data.tobytes())
    fh.close()
    print(f'Binary data have been written to {out_file}')

```

```

if __name__ == '__main__':
    # Set the parameters for a sine wave
    rate = 50
    freq = 2
    duration = 4
    amp = 5
    noise_amp = 0.8

    # Calculate the sine-values
    delta = np.deg2rad(15)
    dt = 1/rate
    omega = 2*np.pi*freq

    t = np.arange(0, duration, dt)
    data = amp * np.sin(omega*t + delta) +
           noise_amp * np.random.randn(len(t))

    # Put them in a pandas-DataFrame, for easier text output
    df = pd.DataFrame({'t':t, 'values':data})

    # Show how to generate a formatted string in Python
    print(f'The first time-sample is {t[0]:5.3f}, '+
          f'and the first data-value is {data[0]:5.3f}\n')

    # Generate the output files
    save_txt(df)
    save_xls(df)
    save_matlab(t, data)
    generate_binary()

```

Modifying Text Files: Imaginary Numbers

Listing B.6: S3_data_read.py

```

""" Solution to Ex. 'Reading in Data', chapter 'Data Input' """
# author: Thomas Haslwanter, date: June-2022

# Import the standard packages
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# ----- data.csv -----
# The name "df" indicates a Pandas-DataFrame
in_file = 'data.csv'
df = pd.read_csv(in_file, index_col=0)
print(df.head())
print(df.tail())

# ----- data_tab.txt -----
in_file = 'data_tab.txt'
df = pd.read_csv(in_file, sep='\t', header=None)
print(df.head())
print(df.tail())

# ----- data_modified.txt -----
in_file = 'data_modified.txt'
df = pd.read_csv(in_file, sep=',', header=2, index_col=0)
df.plot('t', 'values')
plt.show()

```

```
# ----- data.xls -----
in_file = 'data.xls'
df = pd.read_excel(in_file)
print(df.head())

# ----- data.mat -----
from scipy.io import loadmat
in_file = 'data.mat'
data_dict = loadmat(in_file)
print(data_dict['info'])
data = data_dict['data']
plt.plot(data[:,0], data[:,1])
plt.show()
```

Mixed Inputs

Listing B.7: S3_imaginary.py

```
""" Solution to Exercise 'Modifying Text Files',
chapter 'Data Input' """

# author: Thomas Haslwanter, date: June-2022

# Import the required packages
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import os

# Set the required parameters
data_dir = '../Misc'
file_name = 'imaginary.txt'
in_file = os.path.join(data_dir, file_name)
out_file = 'imaginary_out.txt'

# Get the data
df = pd.read_csv(in_file, delim_whitespace=True)

# Add radius and angle as new columns
df['Radius'] = np.sqrt(df.Real**2 + df.Imaginary**2)
df['Angle [rad]'] = np.arctan2(df.Imaginary, df.Real)

# Make sure all columns are floats, and write to the new file
df = df.astype('float')
df.to_csv(out_file, sep='\t', index=None, float_format='%.3f')
print(f'Modified data saved to {out_file}')
```

Binary Data

Listing B.8: S3_read_binary.py

```
"""Solution to Exercise 'Binary Data', chapter 'Data Input'
"""Read in binary data, with an 256 byte ASCII-header.

# author: Thomas Haslwanter, date: June-2022

# Import the required packages
import numpy as np
import matplotlib.pyplot as plt
import array
```

```
# Set the parameters
data_file = 'data.raw'
length_header = 256      # byte

# Approach #1: -----
# Read and show the file header
fh = open(data_file, 'rb')
txt = fh.read(length_header).decode()
print(txt)

# Read all the binary data
bin_data = fh.read()

# Interpret them as 'double', and reshape them to an ndarray
double = 8   # byte
num_cols = 3
num_data = int(len(bin_data)/(num_cols*double))  # must be int
values = array.array('d', bin_data)
mat = np.reshape(values, (num_data,-1))

# Assign the three columns to the variables (t,x,y)
t,x,y = mat.T

# Plot the data
fig, axs = plt.subplots(2,1)
axs[0].plot(t, x)
axs[1].plot(t, y)
axs[0].set_title('Retrieved Data')
plt.show()

# Approach #2: -----
# Define a structured array
dt = np.dtype([('t', 'd'), ('x', 'd'), ('y', 'd')])

# Position the file pointer after the header
fh.seek(256)

# Read the data in as a structured array
structured_array = np.fromfile(fh, dtype=dt)

# Convert that array to a numpy-ndarray
mat = np.array(structured_array.tolist())

# Plot the data again
fig, axs = plt.subplots(2,1)
axs[0].plot(t, x)
axs[1].plot(t, y)
axs[0].set_title('Retrieved Data, Version 2')
plt.show()
```

Problems of Chap. 4

Displaying Data

Listing B.9: S4_display.py

```
""" Solution for Exercise 'Data Display'
Read in weight-data recorded from newborns, and analyze the
data based on the gender of the baby. """

# author: Thomas Haslwanter, date: June-2022

import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
import pandas as pd
import seaborn as sns
import os

def getData():
    """ Read in data from a text-file, and return them as
        labelled DataFrame
    """

    # Set directory and infile
    dataDir = '.'
    inFile = 'babyboom.dat.txt'

    # Read and label the data
    os.chdir(dataDir)
    data = pd.read_csv(inFile, header=None,
                       delim_whitespace=True,
                       names= ['TOB', 'sex', 'Weight', 'Minutes'])

    # Eliminate "Minutes", since this is redundant
    df = data[['Minutes', 'sex', 'Weight']]

    return(df)

def showData(df):
    """ Graphical data display """

    # Show the data: first all of them ....
    plt.plot(df.Weight, 'o')

    plt.title('All data')
    plt.xlabel('Subject-Nr')
    plt.ylabel('Weight [g]')
    plt.show()

    # To make the plots easier to read, replace "1/2"
    # with "female/male"
    df.sex = df.sex.replace([1,2], ['female', 'male'])

    # ... then show the grouped plots
    df.boxplot(by='sex')
    plt.show()

    # Display statistical information numerically
    grouped = df.groupby('sex')
    print(grouped.describe())
```

```
# This is a bit fancier: scatter plots, with labels and
# individual symbols
symbols = ['o', 'D']
colors = ['r', 'b']

fig = plt.figure()
ax = fig.add_subplot(111)

# "enumerate" provides a counter, and variables can be
# assigned names in one step if the "for"-loop uses a tuple
# as input for each loop:
for (ii, (sex, group)) in enumerate(grouped):
    ax.plot(group['Weight'], marker = symbols[ii],
            linewidth=0, color=colors[ii], label=sex)

ax.legend()
ax.set_ylabel('Weight [g]')
plt.show()

# Fancy finish: a kde-plot
df.Weight = np.double(df.Weight) # kdeplot requires doubles

sns.kdeplot(grouped.get_group('male').Weight,
             color='b', label='male')
sns.kdeplot(grouped.get_group('female').Weight,
             color='r', label='female')

plt.xlabel('Weight [g]')
plt.ylabel('PDF(Weight)')
plt.show()

def isNormal(data, dataType):
    """ Check if the data are normally distributed """
    alpha = 0.05
    (k2, pVal) = stats.normaltest(data)
    if pVal < alpha:
        print(f'{dataType} are NOT normally distributed.')
    else:
        print(f'{dataType} are normally distributed.')

def checkNormality(df):
    """ Check selected data vlaues for normality """

grouped = df.groupby('sex')

# Run the check for male and female groups
isNormal(grouped.get_group('male').Weight, 'male')
isNormal(grouped.get_group('female').Weight, 'female')

if __name__ == '__main__':
    """Main Program"""

df = getData()
showData(df)
checkNormality(df)

# Wait for an input before exiting
input('Done - Hit any key to continue')
```

Problems of Chap. 6

Sample Standard Deviation

Listing B.10: S6_sd.py

```
""" Solution to Exercise "Sample Standard Deviation" """
# author: Thomas Haslwanter, date: June-2022

import numpy as np

x = np.linspace(1, 10, 10)
std = np.std(x, ddof=1)
print('The standard deviation of the numbers from 1 to 10 ' +
      f'is {std:4.2f}')
```

Normal Distribution

Listing B.11: S6_normDist.py

```
""" Solution to Exercise "Normal Distribution" """
# author: Thomas Haslwanter, date: June-2022

import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
import pandas as pd
import seaborn as sns

# Generate a PDF with a mean of 5 and a standard deviation of 3
nd = stats.norm(5,3)

# Generate 1000 data from this distribution
data = nd.rvs(1000)

# Standard error
se = np.std(data, ddof=1)/np.sqrt(1000)
print('The standard error is {}'.format(se))

# Histogram
plt.hist(data)
plt.show()

# 95% confidence interval
print('95% Confidence interval: ' +
      f'{nd.ppf(0.025):4.2f} - {nd.ppf(0.975):4.2f}')

# SD for hip implants
nd = stats.norm()
numSDs = nd.isf(0.0005)
tolerance = 1/numSDs
print('The required SD to fulfill both ' +
      f'requirements = {tolerance:6.4f} mm')
```

Other Continuous Distributions

Listing B.12: S6_continuous.py

```
"""Solution for Exercise "Continuous Distribution Functions" """

# author: Thomas Haslwanter, date: June-2022

import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

# T-distribution -----
# Enter the data
x = [52, 70, 65, 85, 62, 83, 59]
""" Note that "x" is a Python "list", not an array!
Arrays come with the numpy package, and have to contain all
elements of the same type.
Lists can mix different types, e.g. "x = [1, 'a', 2]"
"""

# Generate the t-distribution: note that the degrees of freedom
# is the length of the data minus 1.
# In Python, the length of an object x is given by "len(x)"
num = len(x)
dof = num - 1
mean = np.mean(x)
alpha = 0.01

td = stats.t(dof, loc=mean, scale=stats.sem(x))
ci = td.interval(1-alpha)
# This is equivalent to:
# ci = td.ppf([alpha/2, 1-alpha/2])

print(f'mean_weight = {mean:.1f} kg, 99%CI = [{ci[0]:.1f}, {ci[1]:.1f}] kg')

# Chi2-distribution, with 3 DOF -----
# Define the normal distribution
nd = stats.norm()

# Generate three sets of random variates from this distribution
num_data = 1000
data = np.random.randn(num_data, 3)

# Show histogram of the sum of the squares of these random data
plt.hist(np.sum(data**2, axis=1), bins=100, density=True)

# Superpose it with the exact chi-square distribution
x = np.arange(0, 18, 0.1)
chi2 = stats.chi2(df=3)
pdf = chi2.pdf(x)
plt.plot(x, pdf, lw=3)
plt.xlabel('x')
plt.ylabel('Probability(x)')
plt.show()

# F-distribution -----
# Enter the data
femurs_1 = [32.0, 32.5, 31.5, 32.1, 31.8]
femurs_2 = [33.2, 33.3, 33.8, 33.5, 34.0]

# Do the calculations
fval = np.var(femurs_1, ddof=1) / np.var(femurs_2, ddof=1)
```

```

fd = stats.distributions.f(len(femurs_1),len(femurs_2))
pval = fd.cdf(fval)

# Show the results
print(f'The p-value of the F-distribution = {pval:.3f}.')
if pval>0.025 and pval<0.975:
    print('The precisions of the two machines are equal.')
else:
    print('The precisions of the two machines are NOT equal.')

# Uniform distribution -----
ud = stats.uniform(0, 1)
data = ud.rvs(1000)
plt.plot(data, '.')
plt.title('Uniform Distribution')
for ci in [0.95, 0.999]:
    print(f'The {ci*100:.1f}-% confidence interval is {np.float16
(ud.interval(ci))}')

```

Discrete Distributions

Listing B.13: S6_discrete.py

```

"""Solution for Exercise "Continuous Distribution Functions" """

# author: Thomas Haslwanter, date: June-2022

from scipy import stats

# Binomial distribution -----
# Generate the distribution
p = 0.37
n = 15
bd = stats.binom(n, p)

# Select the interesting numbers, and calculate the
# "Probability Mass Function" (PMF)
x = [3,6,10]
y = bd.pmf(x)

# To print the result, we use the "zip" function to generate
# pairs of numbers
for num, solution in zip(x,y):
    print(f'The chance of finding {num} students with blue ' +
          f'eyes is {solution*100:4.1f}%.')

# Poisson distribution -----
# Generate the distribution.
prob = 62/(365/7)
pd = stats.poisson(prob)

# Select interesting numbers, calculate PMF, and print results
x = [0,2,5]
y = pd.pmf(x)*100
for num, solution in zip(x,y):
    print(f'The chance of haveing {num} fatal accidents in one week ' +
          f'is {solution:4.1f}%.')

# The last line just makes sure that the program does not
# close, when it is run from the commandline.
input('Done! Thanks for using programs by thomas.')

```

Problems of Chap. 8

Comparing One or Two Groups

Listing B.14: S8_twoGroups.py

```
""" Solution for Exercise 'Comparing Groups' """

# author: Thomas Haslwanter, date: June-2022

import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
import os


def oneGroup() -> None:
    """ Test of mean value of a single set of data """

    print('Single group of data =====')

    # First get the data
    data = np.array([5260, 5470, 5640, 6180, 6390,
                    6515, 6805, 7515, 7515, 8230, 8770],
                   dtype=float)
    checkValue = 7725    # value to compare the data to

    # (1) Normality test
    # We don't need the first parameter, so we just assign
    # the output to the dummy variable "_"
    (_, p) = stats.normaltest(data)
    if p > 0.05:
        print(f'Data are distributed normally, p = {p:5.3f}')


    # (2) Do the onesample t-test
    t, prob = stats.ttest_1samp(data, checkValue)
    if prob < 0.05:
        print('With the one-sample t-test, ' +
              f'{checkValue:4.2f} is significantly different ' +
              f'from the mean (p={prob:5.3f}).')
    else:
        print('No difference from reference value ' +
              'with onesample t-test.')

    # (3) This implementation of the Wilcoxon test checks for
    # the "difference" of one vector of data from zero
    (_,p) = stats.wilcoxon(data-checkValue, correction=True)
    if p < 0.05:
        print(f'With the Wilcoxon test, {checkValue:4.2f} ' +
              'is significantly different from ' +
              f'the mean (p={p:5.3f}).')
    else:
        print('No difference from reference value with ' +
              'Wilcoxon rank sum test.')


def twoGroups() -> None:
    """Compare the mean of two groups"""
```

```

print('Two groups of data =====')

# Enter the data
data1 = [76., 101., 66., 72., 88., 82., 79., 73., 76., 85.,
         75., 64., 76., 81., 86.]
data2 = [64., 65., 56., 62., 59., 76., 66., 82., 91., 57.,
         92., 80., 82., 67., 54.]


# (1) Normality test
print('\n Normality test -----')

# To do the test for both data-sets, make a tuple
# with "(..., ...)", add a counter with "enumerate", and
# and iterate over the set:
for ii, data in enumerate((data1, data2)):
    _, pval = stats.normaltest(data)
    if pval > 0.05:
        print(f'Dataset # {ii} is normally distributed')


# (2) T-test of independent samples
print('\n T-test of independent samples -----')

# Do the t-test for independent samples
t, pval = stats.ttest_ind(data1, data2)
if pval < 0.05:
    print('With the T-test, data1 and data2 are ' +
          f'significantly different (p = {pval:5.3f})')
else:
    print('No difference between data1 and data2 ' +
          'with T-test.')

# (3) Mann-Whitney test -----
print('\n Mann-Whitney test -----')
u, pval = stats.mannwhitneyu(data1, data2,
                             alternative='two-sided')
if pval < 0.05:
    print('With Mann-Whitney test, data1 and data2 are' +
          f' significantly different(p = {pval:5.3f})')
else:
    print('No difference between data1 and data2 ' +
          'with Mann-Whitney test.')

if __name__ == '__main__':
    oneGroup()
    twoGroups()

```

Comparing Multiple Groups

Listing B.15: S8_multiGroups.py

```

""" Solution for Exercise "Comparing Multiple Groups" """

# author: Thomas Haslwanter, date: June-2022

# Load the required modules -----
# Standard modules
import numpy as np
import matplotlib.pyplot as plt

```

```
from scipy import stats
import pandas as pd

# Modules for data-analysis
from statsmodels.formula.api import ols
from statsmodels.stats.anova import anova_lm
from statsmodels.stats import multicomp

# Module for working with Excel-files
import xlrd

def get_ANOVA_data() -> pd.DataFrame:
    """Get the data for the ANOVA

    Returns
    ------
    df : DataFrame with 'group' and 'weight'
    """

    # First we have to get the Excel-data into Python. This can
    # be done e.g. with the package "xlrd".
    # You have to make sure that you select a valid location on
    # your computer!
    inFile = r'..\..\data\Table 6.6 Plant experiment.xls'
    book = xlrd.open_workbook(inFile)
    # We assume that the data are in the first sheet. This
    # avoids the language problem "Tabelle/Sheet"
    sheet = book.sheet_by_index(0)

    # Select the columns and rows that you want:
    # The "treatment" information is in column "E",
    #           i.e. you have to skip the first 4 columns
    # The "weight" information is in column "F",
    #           i.e. you have to skip the first 5 columns
    treatment = sheet.col_values(4)
    weight = sheet.col_values(5)

    # The data start in line 4, so you have to skip the first 3
    # I use a "pandas" DataFrame, so that I can assign names
    # to the variables.
    df = pd.DataFrame({'group':treatment[3:],
                       'weight':weight[3:]})

    return df

def do_levene(data: pd.DataFrame) ->None:
    """Perform a Levene-test on the data

    Parameters
    ------
    data : DataFrame with 'group' and 'weight'
    """

    print('Levene: -----')
    # Group the data
    grouped = data.groupby('group')

    # Extract the values into a list
    data = []
    for group in (grouped.groups.keys()):
        print(group)
        data.append(grouped.get_group(group).weight)
```

```

# Do the Levene-test on those values
_, p = stats.levene(*data)
if p > 0.05:
    print('The variances are equal, you can do an ANOVA.')
else:
    print('The variances are NOT equal, you should ' +
        'proceed with a Kruskal-Wallis test.')

def do_ANOVA(data: pd.DataFrame) ->None:
    """Perform an ANOVA on the data

    Parameters
    -----
    data : DataFrame with 'group' and 'weight'
    """

    print('ANOVA: -----')

    # First, I fit a statistical "ordinary least square (ols)"
    # -model to the data, using the formula language from
    # "patsy". The formula
    #   'weight ~ C(group)'
    # says:
    #   "weight" is a function of the categorical value "group"
    # and the data are taken from the DataFrame "data", which
    # contains "weight" and "group"
    model = ols('weight ~ C(group)', data).fit()
    # "anova_lm" (where "lm" stands for "linear model")
    # extracts the ANOVA-parameters from the fitted model.
    anovaResults = anova_lm(model)
    print(anovaResults)

    if anovaResults['PR(>F)'][0] < 0.05:
        print('One of the groups is different.')

def compare_many(data: pd.DataFrame) -> None:
    """Multiple comparisons: Which one is different?

    Parameters
    -----
    data : DataFrame with 'group' and 'weight'
    """

    print('\n MultComp: -----')

    # An ANOVA is a hypothesis test, and only answers the
    # question: "Are all the groups from the same distribution?"
    # It does NOT tell you which one is different!
    # Since we now compare many different groups to each other,
    # we have to adjust the p-values to make sure that we don't
    # get a Type I error. For this, we use the statsmodels
    # module "multicomp"
    mc = multicomp.MultiComparison(data['weight'],
                                    data['group'])

    # There are many ways to do multiple comparisons. Here, we
    # choose "Tukeys Honest Significant Difference" test. The
    # first element of the output ("0") is a table containing
    # the results
    print(mc.tukeyhsd().summary())

```

```

# Show the group names
print(mc.groupsunique)

# Generate a print -----
res2 = mc.tukeyhsd()      # Get the data

simple = False
if simple:
    # You can do the plot with a one-liner, but then this
    # does not - yet - look that great
    res2.plot_simultaneous()
else:
    # Or you can do it the hard way, i.e. by hand:

    # Plot values and errorbars
    xvals = np.arange(3)
    plt.plot(xvals, res2.meandiffs, 'o')
    errors = np.ravel(np.diff(res2.confint)/2)
    plt.errorbar(xvals, res2.meandiffs, yerr=errors,
                  fmt='o')

    # Set the x-limits
    xlim = -0.5, 2.5
    # The "*xlim" passes the elements of the variable
    # "xlim" elementwise into the function "hlines"
    plt.hlines(0, *xlim)
    plt.xlim(*xlim)

    # Plot labels (this is a bit tricky):
    # First, "np.array(mc.groupsunique)" makes an array
    # with the names of the groups; and then,
    # "np.column_stack(res2[1][0])" puts the correct
    # groups together
    pair_labels = \
        mc.groupsunique[
            np.column_stack(res2._multicomp.pairindices)]
    labels = ['\n'.join(label) for label in pair_labels]
    plt.xticks(xvals, labels)

    plt.title('Multiple Comparison of Means - Tukey HSD, ' +
              'FWER=0.05' +
              '\n Pairwise Mean Differences')

plt.show()

def KruskalWallis(data: pd.DataFrame) -> None:
    """Non-parametric comparison between the groups

    Parameters
    -----
    data : DataFrame with 'group' and 'weight'
    """

    print('\n Kruskal-Wallis test -----')

    # First, I get the values from the dataframe
    g_a = data['weight'][data['group']=='TreatmentA']
    g_b = data['weight'][data['group']=='TreatmentB']
    g_c = data['weight'][data['group']=='Control']

```

```

# Note: this could also be accomplished with the "groupby"
# function from pandas
groups = pd.groupby(data, 'group')
g_a = groups.get_group('TreatmentA').values[:,1]
g_c = groups.get_group('Control').values[:,1]
g_b = groups.get_group('TreatmentB').values[:,1]

# Then do the Kruskal-Wallis test
h, p = stats.kruskal(g_c, g_a, g_b)
print(f'Result from Kruskal-Wallis test: p = {p:.3f}')


if __name__ == '__main__':
    data = get_ANOVA_data()
    do_levene(data)
    do_ANOVA(data)
    compare_many(data)
    KruskalWallis(data)

    print('\nThanks for using programs by Thomas!\n' +
          'Hit any key to finish')

```

Problems of Chap. 9

A Lady Drinking Tea

Listing B.16: S9_fisherExact.py

```

""" Solution for Exercise 'Categorical Data'
'A Lady Tasting Tea'

# author: Thomas Haslwanter, date: June-2022

from scipy import stats
obs = [[3,1], [1,3]]
_, p = stats.fisher_exact(obs, alternative='greater')

#obs2 = [[4,0], [0,4]]
#stats.fisher_exact(obs2, alternative='greater')

print('\n--- A Lady Tasting Tea (Fisher Exact Test) ---')
print('The chance that the lady selects 3 or more cups ' +
      f'correctly by chance is {p:.3f}')

```

Chi2 Contingency Test

Listing B.17: S9_chi2Contingency.py

```

""" Solution for Exercise "Categorical Data":
Chi2-test with frequency tables

# author: Thomas Haslwanter, date: June-2022

from scipy import stats

obs = [[36,14], [30,25]]
chi2, p, dof, expected = stats.chi2_contingency(obs)

print('--- Contingency Test ---')

```

```

if p < 0.05:
    print(f'p={p:.4f}: the drug affects heart rate.')
else:
    print(f'p={p:.4f}: the drug does NOT affect heart rate.')

obs2 = [[36,14], [29,26]]
chi2, p, dof, expected = stats.chi2_contingency(obs2)
chi2, p2, dof, expected = stats.chi2_contingency(obs2,
    correction=False)

print('If response in 1 non-treated person were different,\n'+
    f' we would get p={p:.4f} with Yates correction,'+
    f' and p={p2:.4f} without.')

```

Chi2 Oneway Test

Listing B.18: S9_chi2OneWay.py

```

""" Solution for Exercise 'Categorical Data' """

# author: Thomas Haslwanter, date: June-2022

from scipy import stats

# Chi2-oneway-test
obs = [4,6,14,10,16]
_, p = stats.chisquare(obs)

print('\n--- Chi2-oneway ---')
if p < 0.05:
    print('The difference in opinion between the different ' +
        f'age groups is significant (p={p:.4f})')
else:
    print('The difference in opinion between the different ' +
        f'age groups is NOT significant (p={p:.4f})')

print(f'DOF={len(obs)-1:3d}')

```

McNemar Test

Listing B.19: S9_mcNemar.py

```

""" Solution for Exercise 'Categorical Data'
McNemar's Test
"""

# author: Thomas Haslwanter, date: June-2022

from scipy import stats
from statsmodels.sandbox.stats.runs import mcnemar

obs = [[19,1], [6, 14]]
obs2 = [[20,0], [6, 14]]

_, p = mcnemar(obs)
_, p2 = mcnemar(obs2)

print('\n--- McNemar Test ---')
if p < 0.05:
    print('The results from the neurologist are ' +
        'significantly different ' +
        f'from the questionnaire (p={p:.3f}).')

```

```

else:
    print('The results from the neurologist are' +
          ' NOT significantly different ' +
          f'from the questionnaire (p={p:5.3f}).')

if (p<0.05 == p2<0.05):
    print('The results would NOT change if the ' +
          ' expert had diagnosed all "sane" people correctly.')
else:
    print('The results would change if the ' +
          ' expert had diagnosed all "sane" people correctly.')

```

Problems of Chap. 12

Correlation

Peak Observations

Listing B.20: S12_peakObservations.py

```

""" Solution for Exercises 'Peak Observations' in Chapter 12
Requires the package 'xlrd' to be installed, with

`pip install xlrd`

"""

# author: Thomas Haslwanter, date: June-2022

# Import the required libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy import stats
import seaborn as sns
import statsmodels.formula.api as sm

def getModelData(show: bool=True) ->None:
    """ Get the data from an Excel-file

    Parameters
    -----
    show : boolean flag, controlling the display
    """

    # First, define the in-file and get the data
    in_file = '..\..\data\AvgTemp.xls'

    # When the data are neatly organized, they can be read in
    # directly with the pandas-functions:
    # with "ExcelFile" you open the file ...
    xls = pd.ExcelFile(in_file)

    # ... and with "parse" you get the data from the file,
    # from the specified Excel-sheet
    data = xls.parse('Tabelle1')

```

```
if show:
    data.plot('year', 'AvgTmp')
    plt.xlabel('Year')
    plt.ylabel('Average Temperature')
    plt.show()

return data

def correlation(data):
    """ Exercise Peak observations - Correlation -----
        Calculate and show the different correlation coefficients """
    """

    pearson = data['year'].corr(data['AvgTmp'],
                                 method = 'pearson')
    spearman = data['year'].corr(data['AvgTmp'],
                                 method = 'spearman')
    tau = data['year'].corr(data['AvgTmp'],
                           method = 'kendall')

    print(f'Pearson correlation coefficient: {pearson:4.3f}')
    print(f'Spearman correlation coefficient: {spearman:4.3f}')
    print(f'Kendall tau: {tau:4.3f}')


def normality_check(data):
    """ Exercise Peak observations - Normality Check """

    # Fit the model
    model = sm.ols('AvgTmp ~ year', data)
    results = model.fit()

    # Normality check -----
    res_data = results.resid      # Get the values for the residuals

    # QQ-plot, for a visual check
    stats.probplot(res_data, plot=plt)
    plt.show()

    # Normality test, for a quantitative check:
    _, pVal = stats.normaltest(res_data)
    if pVal < 0.05:
        print('WARNING: The data are not normally distributed ' +
              f'(p = {pVal})')
    else:
        print('Data are normally distributed.')

def regression(data):
    """ Exercise Peak observations - Regression """

    # Regression -----
    # For "ordinary least square" models, you can do the model
    # with the formula-approach from statsmodels:
    # offsets are automatically included in the model
    model = sm.ols('AvgTmp ~ year', data)
    results = model.fit()
    print(results.summary())

    # Visually, the confidence intervals can be shown using seaborn
    sns.lmplot('year', 'AvgTmp', data)
    plt.show()
```

```

# Is the inclination significant?
ci = results.conf_int()

# This line is a bit tricky: if both are above or both below
# zero the product is positive:
# we look at the coefficient that describes the correlation
# with "year"
if np.prod(ci.loc['year'])>0:
    print('The slope is significant')

if __name__=='__main__':
    data = getModelData()

    correlation(data)
    regression(data)
    normality_check(data)

```

Climate Crisis

Listing B.21: S12_climateCrisis.py

```

""" Solution to Exercise 'Climate Crisis'
of the chapter 'Linear Regresison Models' """

# author: Thomas Haslwanter
# date: June-2022

# Import standard packages
import numpy as np
import matplotlib.pyplot as plt
import os
import pandas as pd
import seaborn as sns
""" Time Series Analysis of global CO2-levels """

# modules from 'statsmodels'
from statsmodels.tsa.seasonal import seasonal_decompose
import statsmodels.formula.api as smf

def get_CO2_data() -> pd.DataFrame:
    """Read in data, and return them as a pandas DataFrame

    Returns
    ------
    df : time stamped CO2-levels at Mauna Loa, Hawaii
    """

    # Get the data, display a few values, and show the data
    url = 'https://www.esrl.noaa.gov/gmd/webdata/ccgg/trends/co2/co2_mm_mlo.txt'
    df = pd.read_csv(url,
                     skiprows=53,
                     delim_whitespace=True,
                     names = ['year', 'month', 'time', 'co2',
                             'deseasoned', 'nr_days', 'std_days',
                             'uncertainty'])


```

```
## Show CO2-levels as a function of time
#df.plot('time', 'co2')
#plt.show()

return df


def decompose(df: pd.DataFrame) -> np.array:
    """ Make a seasonal decomposition of the input data

    Parameters
    -----
    df : time stamped CO2-levels at Mauna Loa, Hawaii

    Returns
    -----
    decomposed : trend, seasonal, and residual data
    """

    # Seasonal decomposition
    result_add = seasonal_decompose(df['co2'],
        model='additive',
        period=12,
        extrapolate_trend='freq')

    ## Show the decomposed data
    #result_add.plot()
    #plt.show()

    return result_add.trend


def find_best_fit(df: pd.DataFrame) -> None:
    """ Take the trend-data from the CO2 measurements,
        and find the best fit

    Parameters
    -----
    df : 'year' in years (decimal), and 'co2' trend of the CO2-data
    """

    # Fit the models, and show the results
    linear = smf.ols('co2 ~ year', df).fit()
    quadratic = smf.ols('co2 ~ year+I(year**2)', df).fit()
    cubic = smf.ols('co2 ~ year+I(year**2)+I(year**3)',
                     df).fit()

    df['linear'] = linear.predict()
    df['quadratic'] = quadratic.predict()
    df['cubic'] = cubic.predict()

    # Show the data
    df.plot('year', ['co2', 'linear', 'quadratic', 'cubic'])

    # Select the best fit
    aics = [linear.aic, quadratic.aic, cubic.aic]
    index = np.argmin(aics)

    print(f'The best fit is of the order {index+1}.')

    plt.show()
    return
```

```

if __name__ == '__main__':
    data = get_CO2_data()
    trend = decompose(data)

    time_co2 = pd.concat({'year': data.time, 'co2': trend},
                         axis=1)
    find_best_fit(time_co2)

```

Problems of Chap. 13

Finding the Right Flower

Listing B.22: S13_logisticRegression.py

```

""" Solution to Exercise 'Logistic Regression' of the chapter 'GLM' """

# author: Thomas Haslwanter, date: June-2022

# Import standard packages
import numpy as np
import matplotlib.pyplot as plt
import os
import pandas as pd
import seaborn as sns

# additional packages
from statsmodels.formula.api import glm
from statsmodels.genmod.families import Binomial

sns.set_context('notebook')

def prepare_fit(in_data: np.ndarray) -> pd.DataFrame:
    """ Use the sepal-length as index, and count occurences
        of 'setosa' and 'other species' for each length

    Parameters
    -----
    in_data : the iris data

    Returns
    -----
    df : 'num_setosa' and 'num_others', for each sepal length
    """

    # Create a dataframe, with suitable columns for the fit
    df = pd.DataFrame()
    df['length'] = np.unique(in_data.sepal_length)
    df['num_setosa'] = 0
    df['num_others'] = 0
    df.index = df.length      # make the 'length' the index

    # Count number of 'setosa' and 'others', for each length
    for cur_length in df.length:
        df.loc[cur_length, 'num_setosa'] = \
            ((iris.sepal_length == cur_length) &
             (iris.species == 'setosa')).sum()

    df.loc[cur_length, 'num_others'] = \
        ((iris.sepal_length == cur_length) &
         (iris.species != 'setosa')).sum()

```

```
# Just to check the total number
df['total'] = df.num_setosa + df.num_others

return df


def logistic(x: np.ndarray, beta:float,
             alpha:float=0) -> np.ndarray:
    """ Logistic Function """
    return 1.0 / (1.0 + np.exp(np.dot(beta, x) + alpha))



def show_results(iris_data: np.ndarray, model) -> None:
    """ Show the original data, and the resulting logit-fit

    Parameters
    -----
    iris_data : input data
    model : model results
        (statsmodels.genmod.generalized_linear_model.GLM)

    """

    sepal_length = iris_data.sepal_length

    # First plot the original data
    plt.figure()
    sns.set_style('darkgrid')
    np.set_printoptions(precision=3, suppress=True)

    plt.scatter(iris.sepal_length, iris.species=='setosa',
                s=50, color="k", alpha=0.2)
    plt.yticks(np.linspace(0, 1, 11))
    plt.ylabel("Setosa")
    plt.xlabel("Sepal Length (cm)")
    plt.title("Probability of 'setosa', " +
              "as function of sepal-length")
    plt.tight_layout

    # Plot the fit
    x = np.linspace(4, 8, 100)
    alpha = model.params[0]
    beta = model.params[1]
    y = logistic(x, beta, alpha)

    plt.plot(x, y, 'r')
    plt.xlim([4, 8])

    out_file = 'setosa.jpg'
    plt.savefig(out_file, dpi=200)
    plt.show()
    print(f'Results save to {out_file}')

    return(x, y)


def find_probabilities(x: np.arange, px: np.arange,
                      lengths: list) -> None:
    """ Find the probability that flower is a 'setosa'

    Parameters
    -----
```

```
-----
x : sepal length
px: corresponding probability of being 'setosa'
lengths : values of interest
"""

# find the closest x-value
for length in lengths:
    index = np.max(np.where(x<length))
    print(f'For a length of {length:4.2f} cm, the ' +
          f'probability of being a "setosa" is {px[index]:5.3f}')

# Find maximum length of having at least a 10% chance of
# being a 'setosa'
chance = 10/100 # [%]

max_index = np.max(np.where(px>chance))
print('The maximum length where you still have a ' +
      f'{chance*100:3.0f}% chance of being a "setosa" is ' +
      f'{x[max_index]:3.1f}cm.')

if __name__ == '__main__':
    # Get the data
    iris = sns.load_dataset('iris')

    # Count occurrences of 'setosa' and 'others',
    # for each length
    f_fit = prepare_fit(iris)

    # fit the model
    # --- >>> START stats <<< ---
    model = glm('num_others + num_setosa ~ length',
                data=df_fit, family=Binomial()).fit()
    # --- >>> STOP stats <<< ---

    print(model.summary())
    (x, px) = show_results(iris, model)
    find_probabilities(x, px, [5, 6])
```

Appendix C

Equations for Confidence Intervals

Standard Deviation of Normal Distribution

Given a set of normally distributed data, the question asked most frequently is: “How well do we know the true mean value of the underlying population?” The answer to this question is provided by Eq. 6.14, and the scale is given by the standard error of the mean.

Another question that can be asked is “How well do we know the variability of the data?”. Thereby the variability is typically indicated by the standard deviation. The two-tailed confidence interval for the standard deviation is given by

$$\sigma \sqrt{\frac{\chi^2_{dof;\alpha/2}}{dof}} \leq s \leq \sigma \sqrt{\frac{\chi^2_{dof;1-\alpha/2}}{dof}}, \quad (\text{C.1})$$

where $dof = \text{len}(data) - 1$ is the degrees of freedom of this dataset.

From this the uncertainty of the estimated standard deviation can be determined:

$$s \sqrt{\frac{dof}{\chi^2_{dof;1-\alpha/2}}} \leq \sigma \leq s \sqrt{\frac{dof}{\chi^2_{dof;\alpha/2}}}, \quad (\text{C.2})$$

Binomial Distribution

General Case

When there are n_S successes in n trials, the best estimate for the probability p of a binomial distribution is given by

$$\hat{p} = n_S/n. \quad (\text{C.3})$$

The confidence intervals of p are given by

$$p_lower = 1 - beta_{inv}(n - n_S + 1, n_S, 1 - \alpha/2) \quad (C.4)$$

$$p_upper = 1 - beta_{inv}(n - n_S, n_S + 1, \alpha/2)$$

where beta_{inv} is the PPF-method of the inverse beta function.

Large Datasets

For large datasets ($p * n > 10$), the binomial distribution can be well approximated by a normal distribution. In that approximation, the confidence intervals of p are

$$CI_p = \hat{p} \pm \mathcal{N}.ppf(1 - \alpha/2) * \sqrt{\frac{\hat{p}(1 - \hat{p})}{n}}, \quad (C.5)$$

where \mathcal{N} is the standard normal distribution.

Poisson Distribution

General Case

Uncommon events in populations, such as the occurrence of specific diseases, are usually modelled using a Poisson distribution. Thereby the incidence rate is estimated as the number of events observed divided by the time at risk of event during the observation period.

The confidence interval of the expected rate can be calculated exactly with a χ^2 -distribution (Ulm 1990):

$$\lambda_{lower} = \frac{\chi^2_{2n,\alpha/2}}{2} \quad (C.6)$$

$$\lambda_{upper} = \frac{\chi^2_{2(n+1),1-\alpha/2}}{2} \quad (C.7)$$

where n is the observed number of events, and λ_{lower} and λ_{upper} are the lower and upper confidence limits for λ , respectively. $\chi^2_{v,\alpha}$ is the chi-square quantile for upper tail probability on v degrees of freedom.

Large Datasets

When the results of n samples from a Poisson distribution are X_1, X_2, \dots, X_n , then for large n the α -% confidence interval can be estimated with the *Wald interval*, which uses the asymptotic normality of the test statistic:

$$conf_interval = \bar{X} \pm \mathcal{N}.ppf(\alpha/2) * \sqrt{\frac{\bar{X}}{n}}. \quad (C.8)$$

Appendix D

Web Ressources

anaconda	https://www.anaconda.com/products/individual
astroem and murray: feedback systems	http://www.cds.caltech.edu/~murray/amwiki Free online book on control systems
bokeh	https://bokeh.org/
git	https://git-scm.com/
github	https://github.com/
gohlke	http://www.lfd.uci.edu/~gohlke/pythonlibs/
Introduction to PyMC	http://camdavidsonpilon.github.io/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers/
ipython	http://ipython.org/
jupyter	https://jupyter.org/
matplotlib	https://matplotlib.org/
numpy	https://numpy.org/devdocs/
pandas	https://pandas.pydata.org/docs/
pep8 = python style guide	https://pep8.org/.
pingouin	https://pingouin-stats.org/
plotly	https://plot.ly/
pypi	https://pypi.org/
pyqtgraph	http://pyqtgraph.org/
pysimplegui	https://github.com/PySimpleGUI/PySimpleGUI
pytest	https://docs.pytest.org
python	https://www.python.org/
python tutorial	https://docs.python.org/3/tutorial/
real python	https://realpython.com/

regular expresseions - cheatsheet

<https://www.debuggex.com/cheatsheet/regex/python>

regular expressions

<http://www.regular-expressions.info/>

sample-size

<https://sample-size.net>

scikit-learn

<https://scikit-learn.org/>

scikit-posthocs

<https://github.com/maximtrp/scikit-posthocs>

scipy

<https://www.scipy.org/>

scipy lecture notes

<https://scipy-lectures.org/>

seaborn

<https://seaborn.pydata.org/>

statsintro-2e

<https://github.com/thomas-haslwanger/statsintro-python-2e>

spyder

<https://www.spyder-ide.org/>

stackoverflow

<https://stackoverflow.com/>

statsmodels

<https://www.statsmodels.org/>

tortoisegit

<https://github.com/TortoiseGit/TortoiseGit>

wing

<http://www.wingware.com/>

winpython

<https://winpython.github.io/>

Glossary

analysis of variance Also called ANOVA. Comparison of the variability within groups to the variability between groups. Used when comparing measurements from three or more groups.

auto-correlation The correlation of a signal with a shifted copy of itself. Used to find hidden patterns in signals.

Bayes' Theorem Describes the probability of an event, based on prior knowledge of conditions that might be related to the event. For example, if the risk of developing health problems is known to increase with age, Bayes' theorem allows the risk to an individual of a known age to be assessed more accurately (by conditioning it on their age) than simply assuming that the individual is typical of the population as a whole. With Bayesian probability interpretation, the theorem expresses how a degree of belief, expressed as a probability, should rationally change to account for the availability of related evidence. Bayesian inference is fundamental to Bayesian statistics.

bias Systematic deviation of a sample statistic from the corresponding population statistic. Often caused by poor selection of subjects.

blocking To reduce the variability of a variable that cannot be randomized by fixating it.

box-plot A common visualisation of the distribution of data, expressed by a box with a line inside that box, and whiskers at the top and bottom. The box indicates the first and third quartile, and the line the median value of the data sample. The whiskers can indicate either the range of the data, or the most extreme value within $1.5 * \text{the inner-quartile-range}$.

case control study A type of observational study in which two existing groups differing in outcome are identified and compared on the basis of some supposed causal attribute. (“First treat, then select.”)

categorical data Data that can take on one of a limited, and usually fixed, number of possible values, with no natural order. (If a “mean value” makes no sense.).

centiles Also called “percentiles”. The $\alpha\%$ -centile is the value of the statistic that is larger than $\alpha\%$ of the sample/population. For example, the median is the 50% percentile.

co-factors Uncontrolled parameters of a study.

cohort study A type of observational study, where you first select the patients, and then follow their development. For instance, in medicine a cohort study starts with an analysis of risk factors. Then the study follows a group of people who do not have the disease. Finally, correlations are used to determine the absolute risk of subject contraction. (“First select, then treat.”)

confidence interval For parameters: Interval estimate of a population parameter, which contains the true value of the parameter with a defined percent-probability (e.g., 95%-CI). For data: interval estimate that contains 95% of the data.

correlation Any departure of two or more random variables from independence.

covariate Refers to a variable that is possibly predictive of the outcome being studied, and can be a factor or a co-factor.

crossover study A longitudinal study in which all subjects receive a sequence of different treatments.

cumulative distribution function The probability to find a random variable with a value lower than the given one.

density Function of a continuous parameter.

degrees of freedom The number of degrees of freedom is the number of values in the final calculation of a statistic that are free to vary.

density A continuous function that describes the relative likelihood for a random variable to take on a given value. E.g.: *kernel-density* estimation (KDE), or *probability-density* function (PDF).

design matrix The data matrix \mathbf{X} in the regression model $y = \mathbf{X} \cdot \boldsymbol{\beta} + \epsilon$.

distribution A function which assigns a probability to each measurable subset of the possible outcomes of a random experiment.

experimental study Study where the selection of the subjects as well as the conditions of the study are under the control of the investigator.

factor Also called *treatment* or *independent variable*, is an explanatory variable manipulated by the experimenter. A controlled parameter of a study.

function A Python object that accepts input data, executes commands and calculations with them, and can return one or more return objects.

generalized linear model Generalization of linear regression by allowing the linear model to be related to the response variable via a link function. For example, while a linear model could be $y = m * x + b$, a GLM could be $y = \frac{1}{1+exp(-m*x+b)}$.

hypothesis test A method of statistical inference used for testing a statistical hypothesis.

kurtosis Measure of the peakedness of a distribution, relative to the normal distribution. Is approximately 3 for normally distributed data. Deviations from 3 are called *excess kurtosis*.

linear regression Modeling a scalar variable (*dependent variable*) using a linear predictor function. The unknown model parameters are estimated from the data. Simple linear regression: $y = k * x + d$.

Multiple linear regression: $y = k_1 * x_1 + k_2 * x_2 + \dots + k_n * x_n + d$.

location Parameter that shifts the mean of a probability distribution.

logistic regression Also called *logit regression*. The probabilities describing the possible outcomes of a single trial are modeled as a function of the explanatory (predictor) variables, using a logistic function: $f(x) = \frac{L}{1+e^{-k(x-x_0)}}$.

Markov Chain Stochastic model of a process where the probability of each state only depends on the previous state.

maximum likelihood For a fixed set of data and an underlying statistical model, the method of maximum likelihood selects the set of values of the model parameters that maximizes the likelihood function. Intuitively, this maximizes the “agreement” of the selected model with the observed data, and for discrete random variables it indeed maximizes the probability of the observed data under the resulting distribution.

median value The value separating the higher half of the data sample from the lower half.

minimization Closely related to *randomization*. Thereby one takes whichever treatment has the smallest number of subjects, and allocates this treatment with a probability greater than 0.5 to the next patient.

mode value The highest value in a discrete or continuous probability distribution.

module A file containing Python variables and function definitions.

Monte Carlo Simulation Repeated simulation of the behavior of some parameter based on repeated sampling of a random variable.

numerical data Data that can be expressed by a (continuous or discrete) number.

observational study Study where the assignment of subjects into a treated group versus a control group is outside the control of the investigator.

paired test Two data sets are *paired* when the following one-to-one relationship exists between values in the two data sets: (1) Each data set has the same number of data points. (2) Each data point in one data set is related to one, and only one, data point in the other data set.

percentile Also called *centile*. Value that is larger or equal than $p^*100\%$ of the data, where $0 < p < 1$.

population Includes all of the elements from a set of data.

post-hoc analysis Analyzing the data for patterns that were not specified in advance. For example, after an ANOVA has established that at least one of the groups tested does *not* come from the same population, a *post-hoc test* can investigate *which* groups are different from each other.

power analysis Calculation of the minimum sample size required so that one can be reasonably likely to detect an effect of a given size.

power Same as *sensitivity*. Denoted by $1 - \beta$, with β the probability of type II errors.

primary outcome measure The most important outcome measure of a study.

probability density function (PDF) A continuous function whose value at a give point provides a relative likelihood that the value of the corresponding random variable would be close to that point. The probability to find the value of the random variable in a given interval is the integral of the probability density function over that interval.

probability mass function (PMF) A discrete function which defines the probability to obtain a given number of events in an experiment or observation.

prospective study A prospective study watches for outcomes, such as the development of a disease, during the study period and relates this to other factors such as suspected risk or protection factor(s).

package A folder containing one or more Python modules and an “.ini”-file.

quantile Value that is larger or equal than $p * 100\%$ of the data, where $0 < p \leq 1$.

quartile Value that is larger or equal than 25% / 50% / 75% of the data (first/second/third quartile). The 2nd quartile is equivalent to the median.

randomization A method to eliminate bias from research results, by dividing a homogeneous group of subjects into a *control group* (which receives no treatment) and a *treatment group*.

ranked data Numbered data where the number corresponds to the rank of the data, i.e., the number in the sequence of the sorted data, and not to a continuous value.

regression Prediction of the value of one variable from the value(s) of one or more other variables.

regular expression A sequence of characters that define a search pattern, mainly for use in pattern matching with strings. Available for Unix, and for *Python*, *Perl*, *Java*, *C++*, and many other programming languages.

residual Difference between the observed value and the estimated function value.

retrospective study A retrospective study looks backwards and examines exposures to suspected risk or protection factors in relation to an outcome that is established at the start of the study.

ROC-curve Short for *receiver-operator-characteristic* curve. A graph showing the true-positive rate versus the false-positive rate. The ROC-curve is a tool to determine the optimal value to distinguish between two groups.

sample One or more observations from a population.

scale Parameter that controls the variance of a probability distribution.

sensitivity Proportion of actual positives which are correctly identified as such (e.g., the percentage of sick people who are correctly identified as having the condition).

shape parameter Parameters beyond *location* and *scale* which control the shape of a probability distribution. Rarely used.

significance level Denoted by α , is the probability of the study rejecting the null hypothesis, given that the null hypothesis was assumed to be true. For a *significant difference*, typically a significance level of $\alpha = 5\%$ is used.

skewness Measure of the asymmetry of a distribution.

specificity Proportion of actual negatives which are correctly identified as such (e.g., the percentage of healthy people who are correctly identified as not having the condition).

standard deviation Square root of variance.

standard error Often short for *standard error of the mean*. Square root of the variance of a statistic.

treatment Same as *factor*.

type I error A type I error occurs when one rejects the null hypothesis when it is true. The probability of a type I error is the level of significance of the hypothesis test, and is commonly denoted by α .

type II error A type II error occurs when one rejects the alternative hypothesis (fails to reject the null hypothesis) when the alternative hypothesis is true. Therefore it is dependent on an alternative hypothesis. The probability of a type II error is commonly denoted by β .

unpaired test Test with two sets of independent data.

variable A symbol which acts as a placeholder for a quantity. For example, the variable *weight* might represent the weight of a subject. Compare to the meaning of *variate* below.

variance Measure of how far a set of numbers is spread out. Mathematically, it is the expected value of the squared deviation from the mean: $Var(X) = E[(X - \mu)^2]$. The variance of a sample gives an estimate of the population variance that is biased by a factor of $\frac{n-1}{n}$. The best unbiased estimate of the population variance is therefore given by $s^2 = \frac{1}{n-1} \sum_{i=1}^n (y_i - \bar{y})^2$ which is called *(unbiased) sample variance*.

variate A specific representation of a variable. For example, the variates [65.3, 73.2] could be specific instances of the variable *weight*.

Bibliography

- Altman, D. G. (1999). *Practical statistics for medical research*. London: Chapman & Hall/CRC.
- Andrade, C. (2015). The primary outcome measure and its importance in clinical trials. *76*, e1320–e1323.
- Bishop, C. (2007). *Pattern recognition and machine learning*. Berlin: Springer.
- Box, J. F. (1978). *R. A. Fisher: The life of a scientist*. New York: Wiley.
- Button, K. S., Ioannidis, J. P. A., Mokrysz, C., Nosek, B. A., Flint, J., Robinson, E. S. J., & Munafò, M. R. (2013). Power failure: Why small sample size undermines the reliability of neuroscience. *14*, 365–376.
- Chatfield, C., & Xing, H. (2019). *The analysis of time series* (7th ed.). London: Chapman and Hall/CRC.
- Chow, S-C., Shao, J., & Wang, H. (2008). *Sample size calculations in clinical research* (2nd ed.). Boca Raton: Chapman & Hall/CRC.
- Dobson, A., & Barnett, A. (2018). *An introduction to generalized linear models* (4th ed.). Boca Raton: CRC Press.
- Duda, R. O., Hart, P. E., & Stork, D. G. (2004). *Pattern classification* (2nd ed.). New York: Wiley-Interscience.
- Ghasemi, A., & Zahediasl, S. (2012). Normality tests for statistical analysis: A guide for non-statisticians. *International Journal of Endocrinology and Metabolism*, *10*(2), 486–489.
- Haslwanter, T. (2021). *Hands-on signal analysis with python*. Berlin: Springer International Publishing.
- Holm, S. (1979). A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, *6*, 65–70.
- Hyndman, R. J., & Athanasopoulos, G. (2018). *Forecasting: Principles and practice* (3rd ed.). OTexts.
- Ioannidis, J. P. A. (2005). Why most published research findings are false. *2*, e124.
- Kaplan, D. (2009). *Statistical modeling: A fresh approach*. Saint Paul: Macalester College.
- Kaplan, R. M., & Irvin, V. L. (2015). Likelihood of null effects of large nhlbi clinical trials has increased over time. *PLoS One*, *10*(8), e0132382.
- Klamroth-Marganska, V., Blanco, J., Campen, K., Curt, A., Dietz, V., Ettlin, T., et al. (2014). Three-dimensional, task-specific robot therapy of the arm after stroke: A multicentre, parallel-group randomised trial. *The Lancet Neurology*, *13*(2), 159–166.
- McCullagh, P. (1980). Regression models for ordinal data. *Journal of the Royal Statistical Society. Series B (Methodological)*, *42*(2), 109–142.
- McCullagh, P., & Nelder, J. (1989). *Generalized linear models* (2nd ed.). Berlin: Springer.
- McGraw, K. O., & Wong, S. P. (2022). A common language effect size statistic. *111*(2), 361–365.

- Montgomery, D. C. (2019). *Introduction to statistical quality control* (8th ed.). New York: Wiley.
- Nuzzo, R. (2014). Scientific method: Statistical errors. *Nature*, 506(7487), 150–152.
- OSC, O. S. C. (2015). Psychology. Estimating the reproducibility of psychological science. *Science*, 349(6251):aac4716.
- Pilon, C. D. (2015). Probabilistic programming and bayesian methods for hackers.
- Riffenburgh, R. (2012). *Statistics in medicine* (3rd ed.). Cambridge: Academic.
- Rosenbaum, P. R., & Rubin, D. B. (1983). The central role of the propensity score in observational studies for causal effects. *Biometrika*, 70(1), 41–55.
- Scopatz, A., & Huff, K. D. (2015). *Effective computation in physics*. Sebastopol: O'Reilly Media Inc.
- Sellke, T., Bayarri, M. J., & Berger, J. O. (2001). Calibration of p values for testing precise null hypotheses. *The American Statistician*, 55, 62–71.
- Shumway, R. H., & Stoffer, D. S. (2017). *Time series analysis and its applications*. Berlin: Springer.
- Smith, III, J. O. (2007). *Mathematics of the Discrete Fourier Transform (DFT): With audio applications*. W3K Publishing.
- Ulm, K. (1990). A simple method to calculate the confidence interval of a standardized mortality ratio (smr). *131*, 373–375.
- Wilkinson, G., & Rogers, C. (1973). Symbolic description of factorial models for analysis of variance. *Applied Statistics*, 22, 392–399.

Index

A

Akaike Information Criterion (AIC), 240, 248
Alternative hypothesis, 148
Anaconda, 9
Analysis of Variance (ANOVA), 168
 balanced, 168
 three-way, 175
 two-way, 174
Anscombe's quartet, 258
ARMA, 224
Array, 16
Autocorrelation, 217
Autoregressive Integrated Moving Average (ARIMA), 223

B

Backend, 63
Bayes' rule, 277
Bayes' Theorem, 275
Bayesian Information Criterion (BIC), 248
Bayesian Statistics, 275
Bias, 98
Bivariate, 90
Bivariate data, 78, 208
Blinding, 100
Block randomization, 99
Bonferroni correction, 173
Bootstrapping, 262

C

Cell means model, 236
Censoring, *see* censorship

Censorship, 198
Centiles, 107
Central limit theorem, 123
Chi-square tests, 181, 183
Clinical investigation plan, 102
Clinical relevance, 149
Clinical significance, *see* clinical relevance
Code versioning, 285
Coefficient of determination, 214
 adjusted, 244
Cofactors, 94
Cohen's d, 149, 162
Condition number, 253
Confidence interval, 127, 135, 182
Confirmatory research, 146
Confoundings, 94
Consumer risk, 148
Contingency table, 181
Control group, 96
Correlation
 autocorrelation, 217
 coefficient, 208
 cross-correlation, 205
 Kendall's τ , 210
 matrix, 214
 Pearson, 209
 Spearman, 210
Covariance, 208
Covariate, 94, 235
Cox proportional hazards model, 202
Cox regression model, 202
Cross-correlation, 205
Cumulative Distribution Function (CDF), 107, 113
Cumulative frequency, 74

- D**
- Data
 - categorical, 89
 - nominal, 89
 - numerical, 89
 - ordinal, 89
 - ranked, 89
 - Data decomposition, 218
 - Data display, 59
 - Dataframe, 17
 - Data input, 49
 - Debugger, 283
 - Degrees of Freedom (DOF), 94
 - Density, 92
 - Dependent variable, 235
 - Design matrix, 234
 - Design of experiments
 - factorial, 100
 - observational, 96
 - Dictionary, 17
 - Distributions
 - Bernoulli, 116
 - binomial, 117
 - center, 105
 - chi-square, 128
 - continuous, 132
 - discrete, 115
 - exponential, 134
 - exponential family, 266
 - F-distribution, 129
 - Fréchet, 197
 - hypergeometric, 119
 - location, 111
 - lognormal, 132
 - normal, 120
 - poisson, 118
 - scale, 112
 - t-distribution, 126
 - uniform, 134
 - Weibull, 133
 - z, 120
 - Documentation, 102
 - Dummy coding, 236
- E**
- Effects, 236
 - Effect size, 162
 - Endogenous variable, 235
 - Error
 - Type I, 147
 - Type II, 148
- F**
- Excel, 54
 - Excess kurtosis, 112, 252
 - Exogenous variable, 235
 - Expected value, 92
 - Explanatory variable, 235
 - Exploratory research, 146
- G**
- Factorial design, 100
 - Factors, 94
 - Filliben's estimate, 141
 - Frequency, 181
 - Frequency tables, 181, 183
 - Frozen distribution, 114
 - Frozen distribution function, 116
- H**
- Gaussian distribution, 120
 - Generalized Linear Models (GLM), 266
 - Geometric mean, 107
 - Git, 285
 - Github, 286
 - Graphical User Interface (GUI), 287
- I**
- Holm-Bonferroni correction, 174
 - Holm correction, 174
 - Homoscedasticity, 258
 - Hypotheses, 144
 - Hypothesis test, 229
- IDE**
- Integrated Development Environment (IDE), 28
- Interactions**
- 95
- Intercept**
- 167, 235
- Interpretation**
- Bayesian, 275
 - frequentist, 275
- IQR**
- Inter-Quartile-Range (IQR), 108
- IPython**
- personalization, 31
 - Tips, 45

J

Jupyter
notebook, 30
Qt console, 29

K

Kaplan–Meier survival curve, 199
Kernel Density Estimator (KDE), 72
Kurtosis, 112, 252

L

Likelihood ratio, 154
Linear predictor, 266, 267
Linear regression, 211
assumptions, 257
Link function, 267, 270
List, 16
Log likelihood function, 247
Logistic function, 267
Logistic ordinal regression, 270
Logistic regression, 266, 267

M

Main effects, 95
Markov chain Monte Carlo modeling, 278
Matlab, data input from, 54
Matplotlib, 63
Maximum likelihood, 246
Mean, 105
Median, 106
Minimization, 99
Modal value, 106
Mode, 106
Models
ARIMA, 223
ARMA, 224
Module, 14
Moment
first, 92
second standardized, 93
Multicollinearity, 253
Multiple comparisons, 172
Multivariate, 90
Multivariate data, 80

N

Nan's, 106
Negative likelihood ratio, 154
Negative predictive value, 152
Non-parametric tests, 140, 159

Non-response bias, 98
Normal distribution, 120
examples, 123
sum of, 122
Normality check, 140
Nuisance factors, 94
Null hypothesis, 125, 145
Numpy, 8

O

One-hot-encoding, 236
One-tailed t-test, 161
Ordinal logistic regression, 266
Ordinary Least-Squares (OLS), 211
Outliers, 140

P

Pandas, 20
Parametric tests, 140, 159
Partial autocorrelation, 221
Patsy, 233
Pattern, 205
Percentiles, *see* centiles
Pingouin, 40
Plots
3D, 80
box plot, 76
error bars, 75
histogram, 71
interactive, 66
kde, 72
pp-plot, 141
probability plot, 140, 141
probplot, 140
qq-plot, 141
rug, 73
scatter, 71
surface, 80
univariate data, 71
wireframe, 80

Population, 87

Positive likelihood ratio, 154
Positive predictive value, 152
Post-hoc analysis, 172
Posterior probability, 276
Power, 147, 149
Power analysis, 98, 149
Predictor variable, 235
Predictor, linear, *see* linear predictor
Prevalence, 153
Primary outcome measure, 95

- Prior probability, 276
 Probability Density Function (PDF), 92, 120
 Probability distribution, 91
 Probability Mass Function (PMF), 90, 91, 116
 Process
 ARMA, 223
 autoregressive (AR), 223
 moving average (MA), 223
 Producer risk, 147
 Proportional odds model, 272
 P-value, 145
 Pylab, 63
 Pyplot, 63
 PyQtgraph, 289
 PySimpleGUI, 288
 Python, 7
 data structures, 15
 distributions, 8
 IDEs, 36
 installation, 10
 plotting, 62
 resources, 13
 Tips, 44
- Q**
 Qt console, 34
- R**
 Random variate, 90
 Randomization, 99
 Randomized controlled trial, 96
 Range, 107
 Rank correlation, 210
 Regressand, 235
 Regression
 linear, 211
 multilinear, 214, 233, 236
 multiple, 233
 Regression coefficients, 236
 Regressor, 235
 Regular expressions, 53
 Repository, 6
 Residuals, 95, 211
 Response variable, 235
 Revision control, 285
 Right-censored data, 199
 ROC curve, 155
- S**
 Sample, 87
 Sample mean, 92
 Sample selection, 97
 Sample size, 149
 Sample standard deviation, 93
 Sample variance, 169
 Scatterplot matrix, 214
 Scipy, 8
 Seaborn, 39
 Sensitivity, 152
 Shape parameters, 112
 Significance level, 145
 Simple linear regression, 232
 Skewness, 112, 252
 Slicing, 17
 Specificity, 152
 Standard deviation, 93
 Standard error, 109
 Standard normal distribution, 120
 Stationary process, 223
 Statistical inference, 145
 Statistical modeling, 229
 Statsmodels, 41
 Stratified randomization, 99
 Studentized range, 172
 Study
 case control, 96
 cohort, 96
 controlled, 96
 crossover, 97
 cross-sectional, 96
 experimental, 96
 longitudinal, 96
 observational, 96
 prospective, 96
 retrospective, 96
 uncontrolled, 96
 Study design, 96
 Sum of squares, 169
 Survival analysis, 197
 Survival times, 197
- T**
 Test, 284
 ANOVA, 168, 174
 binomial, 118
 chi square, contingency, 186
 chi square, one way, 185
 Cochran's Q, 181, 193
 Durbin-Watson, 253, 259
 F-test, 171

- Fisher's exact, 188
Friedman, 176, 177
Jarque-Bera, 253
Kolmogorov-Smirnov, 142
Kruskal-Wallis, 174
Levene, 169
Lilliefors, 142
logrank, 202
Mann-Whitney, 166
McNemar's, 191
nose, 284
omnibus, 143, 252
paired t-test, 164
PyTest, 284
Shapiro-Wilk, 142
t-test, independent groups, 165
t-test, one sample, 159
Tukey's, 172
unittest, 284
variance ratio, 171
Wilcoxon signed rank sum, 162
Time series analysis, 217
Tortoisegit, 287
Transformation, 143
- Treatments, 94
Tuple, 16
Two-tailed t-test, 161
- U**
- Unexplained variance, 213
Unimodal, 110
Univariate, 90, 105
- V**
- Variability, 93
Variance, 93
Variate, 90, 114, 123
Vectors, 16
Version control, 285
 code versioning, 285
- W**
- Weibull modulus, 133
Wing, 39
WinPython, 9