Minimalist Data Wrangling with Python

Release [DRAFTv0.2.2]

Marek Gagolewski

2022-05-20T12:47:23+1000

Contents

This is a draft of *Minimalist Data Wrangling with Python* by Marek Gagolewski[1]. It's distributed in the hope that it'll be useful. Any bug/typos reports/fixes[2] are appreciated. Although available online, this is a whole course, and should be read from the beginning to the end. In particular, refer to the *Preface* for general introductory remarks. Enjoy.

Copyright (C) 2022, Marek Gagolewski[3]

This material is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (CC BY-NC-ND 4.0[4]). This book is available both in PDF[5] and HTML[6].

[1] https://www.gagolewski.com
[2] https://github.com/gagolews/datawranglingpy/blob/master/CODE_OF_CONDUCT.md
[3] https://www.gagolewski.com
[4] https://creativecommons.org/licenses/by-nc-nd/4.0/
[5] https://datawranglingpy.gagolewski.com/datawranglingpy.pdf
[6] https://datawranglingpy.gagolewski.com/

0 *Preface*

0.1 The Art of Data Wrangling

The broadly-conceived data science aims at making sense of and generating predictions from data that have been collected in large quantities from various sources, e.g., physical sensors, files, databases, or (pseudo)random number generators. It can take different forms, e.g., vectors, matrices and other tensors, graphs, audio/video streams, text, etc. With the advent of the internet era, data have become ubiquitous.

Exercise 0.1 *Think about how much information you consume and generate when you interact with social media or news feeds every day.*

Here are some application domains where data-driven decision making, modelling, and prediction has already proven itself very useful:

- healthcare,
- food production,
- pharmaceuticals,
- transportation,
- financial services (banking, insurance, investment funds),
- real estate,
- retail.

Okay, to be frank, the above list was generated by duckduckgoing the "biggest industries" query. That was a very easy task; data science (and its very different flavours, including statistics, operational research, machine learning, artificial intelligence, and so forth) is everywhere. Basically, wherever we have data and there is a need to improve some processes or discover new aspects about a problem domain, there is a place for data-driven solutions.

Of course, it's not all about business revenue (luckily). We can do a lot of great work for greater good; with the increased availability of open data, everyone can be a reporter, an engaged citizen that seeks truth. There are NGOs. Finally, there are researchers (remember that the main role of most universities is still to spread the advancement of knowledge and not to make money!) that need these methods to make new discoveries, e.g., in psychology, economics, sociology, agriculture, engineering, biotechnology, pharmacy, medicine, genetics, you name it.

Important: Data rarely come in a *tidy* and *tamed* form. Performing accurate exploration and modelling heavily relies on **data wrangling**, which is the very broad process of appropriately preparing raw data for further analysis.

And thus, in this course, we are going to explore methods for:

- performing exploratory data analysis, including aggregating and visualising numerical and categorical data,
- working with different types of data (e.g., text, time series) gathered from structured and unstructured sources,
- cleaning data by identifying outliers,
- handling missing data,
- transforming, selecting, and extracting features, dimensionality reduction,
- identifying naturally occurring data clusters,
- applying sampling techniques,
- data modelling using basic machine learning algorithms,
- maintaining data privacy and exercising ethics in data manipulation.

0.2 Aims and Scope

Most of the time during the course of this course, we will be writing code in Python[7]. The 2021 StackOverflow Developer Survey[8] lists it as the 2nd most popular programming language used nowadays (slightly behind JavaScript).

Over the last years, Python has proven a very robust choice for learning and applying

[7] https://www.python.org/
[8] https://insights.stackoverflow.com/survey/2021#technology-most-popular-technologies

data wrangling techniques. This is possible thanks to the famous[9] high quality packages written by the devoted community of open source programmers, including but not limited to **`numpy`**[10], **`scipy`**[11], **`pandas`**[12], **`matplotlib`**[13], **`seaborn`**[14], and **`sklearn`**[15].

Important: Note that we will introduce the Python language from scratch and that we do not require any prior programming experience. Nevertheless, learning how to code is hard work; if you want to succeed, you will have to spend a decent amount of time getting your hands dirty by writing programs which solve the suggested problem sets, studying technical manuals, and so forth. One does not became a qualified and respected engineer by simply *reading* online tutorials or books.

Of course, Python and third-party packages written therein is amongst many software tools which can help gain new knowledge from data. Other open source choices include, e.g., R[16] and Julia[17]. There are also some commercial solutions available on the market, but we believe that ultimately all software should be free[18].

Note: We put great emphasis on developing *transferable skills* so that all that we learn here can be then applied quite easily in other environments. In other words, this is a course on data wrangling (*with* Python), and not a course *on* Python (with examples in data wrangling). This is not a book of recipes. We are aiming for understanding and becoming independent learners.

The skills we are going to develop in this course are *fundamental* for the success in the numerous jobs available in our industry all over the world. And data engineers, data scientists, machine learning specialists, statisticians, and business analysts are amongst the most well-paid specialists[19]. Money does not bring joy, but luckily it is a very interesting domain anyway!

All that we shall learn can be used for improving different processes, in research, helping NGOs, debunking false news or wishful thinking, maintaining quality of various

[9] https://insights.stackoverflow.com/survey/2021#other-frameworks-and-libraries
[10] https://numpy.org/
[11] https://scipy.org/
[12] https://pandas.pydata.org/
[13] https://matplotlib.org/
[14] https://seaborn.pydata.org/
[15] https://scikit-learn.org/
[16] https://www.r-project.org/
[17] https://julialang.org/
[18] https://www.gnu.org/philosophy/free-sw.en.html
[19] https://insights.stackoverflow.com/survey/2021#other-frameworks-and-libraries

industrial processes, and doing any other good deeds for the advancement of humanity.

We're going to study many methods and algorithms that stood the test of time and that continue to inspire the researchers and practitioners. After all, many "complex" algorithms are merely variations on or clever combinations of the most basic ones. You might not see it now, but this will become evident as we progress.

Most importantly, however, we will get to know their limitations, which they are many. Being sceptical and cautious is one of the traits of a good scientist!

Note that we will definitely not be avoiding mathematical notation so as to maintain a healthy level of generality. Mathematics is both a universal tool and a language for describing the methods for processing various data structures and analysing the properties thereof. The people fluent in mathematics are those who have invented or derived most of the methods discussed herein, we must thus be too.

0.3 About the Author

Marek Gagolewski[20] is currently a Senior Lecturer in Applied AI at Deakin University in Melbourne, VIC, Australia and an Associate Professor in Data Science (on long-term leave) at the Faculty of Mathematics and Information Science, Warsaw University of Technology, Poland and Systems Research Institute of the Polish Academy of Sciences.

His research interests include machine learning, data aggregation and clustering, computational statistics, mathematical modelling (science of science, sport, economics, etc.), and free (libre) data analysis software (stringi[21], genieclust[22], among others[23]).

0.4 Acknowledgements

Minimalist Data Wrangling with Python bases on my experience as an author of a quite successful textbook *Przetwarzanie i analiza danych w języku Python* (Data Processing and Analysis in Python), [[GBC16]] that I have written (in Polish, 2016, published by PWN)

[20] https://www.gagolewski.com
[21] https://stringi.gagolewski.com
[22] https://genieclust.gagolewski.com
[23] https://github.com/gagolews

with my former (successful) PhD students Maciej Bartoszuk and Anna Cena – thanks! The current book is a completely different work, however its predecessor served as a great testbed for many ideas conveyed here.

The ideas have also been battle-tested at Warsaw University of Technology, Data Science Retreat (Berlin), and Deakin University (Melbourne) – I thank my students for the feedback given over the last 7 or so years.

This book has been prepared in a Markdown superset called MyST[24], Sphinx[25], and TeX (XeLaTeX) . Python code chunks have been processed with the R (sic!) package **knitr**. A little help of Makefiles, custom shell scripts, and Sphinx plugins (**sphinxcontrib-bibtex**[26], **sphinxcontrib-proof**[27]) dotted the *j*'s and crossed the *f*'s. The **Ubuntu Mono**[28] font is used for the display of `code`.

[24] https://myst-parser.readthedocs.io/en/latest/index.html
[25] https://www.sphinx-doc.org/
[26] https://pypi.org/project/sphinxcontrib-bibtex/
[27] https://pypi.org/project/sphinxcontrib-proof/
[28] http://font.ubuntu.com/

Part I

Introducing Python

1

Getting Started with Python

1.1 Installing Python

Python[29] was designed and implemented by a Dutch programmer Guido van Rossum in the late 1980s. It is a very popular[30] object-oriented programming language that is quite suitable for rapid prototyping. Its name is a tribute to the funniest British comedy troupe ever, therefore we will surely be having a jolly good laugh along our journey.

We will be using the reference implementation of the Python language (called CPython[31]), version 3.8 (not: 2.x!).

A convenient way to start with Python is by installing a data science-oriented distribution called Anaconda, which features the language interpreter, the **`conda`** package manager, as well as a number of pre-installed packages (including **`numpy`**, **`scipy`**, **`pandas`**, **`matplotlib`**, **`seaborn`**, **`jupyter`**, etc.).

Exercise 1.1 *Download[32] and install the latest version of Anaconda Individual Edition. If you run into any troubles, consult the official product documentation[33].*

Note: The users of Unix-like operating systems (GNU/Linux, FreeBSD, etc.), just like yours truly, should prefer downloading Python via their native package manager (e.g., `sudo apt install python3` in Debian and Ubuntu). Then, installing[34] additional Python packages can either be installed by the said manager or directly from the Python Package Index (PyPI[35]) via the **`pip`** tool.

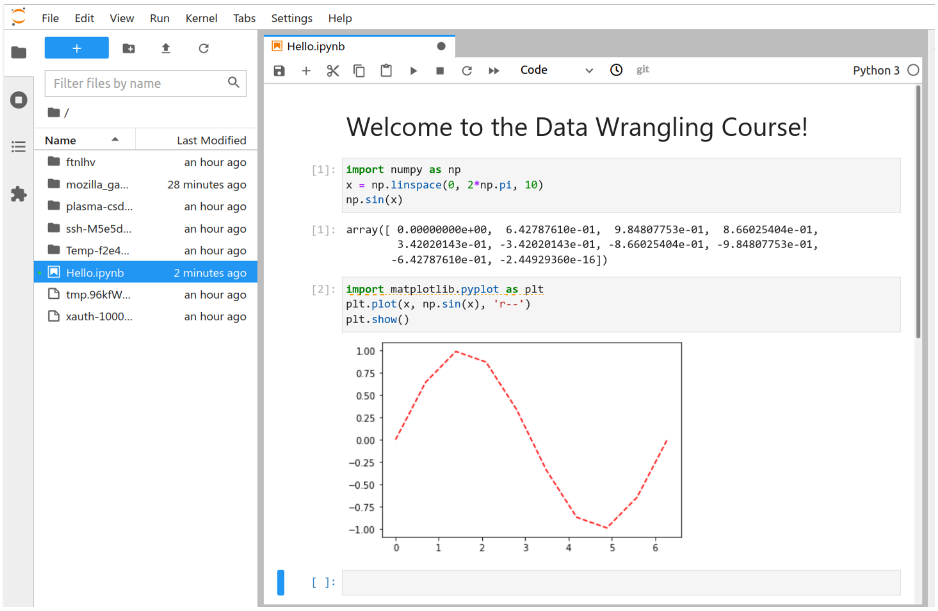
GNU/Linux is the language of choice of machine learning engineers and data scientists

[29] https://www.python.org/
[30] https://insights.stackoverflow.com/survey/2021#technology-most-popular-technologies
[31] https://www.python.org
[32] https://www.anaconda.com/products/individual
[33] https://docs.anaconda.com/anaconda/install/
[34] https://packaging.python.org/en/latest/tutorials/installing-packages/
[35] https://pypi.org/

both on the desktop and in the cloud. Switching to a free system at some point cannot be recommended highly enough.

1.2 Working with Jupyter Notebooks

JupyterLab[36] is a web-based development environment supporting numerous[37] programming languages, including of course Python; see Figure 1.1.



Name	Last Modified
ftnlhv	an hour ago
mozilla_ga...	28 minutes ago
plasma-csd...	an hour ago
ssh-M5e5d...	an hour ago
Temp-f2e4...	an hour ago
Hello.ipynb	2 minutes ago
tmp.96kfW...	an hour ago
xauth-1000...	an hour ago

Welcome to the Data Wrangling Course!

```
[1]: import numpy as np
     x = np.linspace(0, 2*np.pi, 10)
     np.sin(x)

[1]: array([ 0.00000000e+00,  6.42787610e-01,  9.84807753e-01,  8.66025404e-01,
             3.42020143e-01, -3.42020143e-01, -8.66025404e-01, -9.84807753e-01,
            -6.42787610e-01, -2.44929360e-16])

[2]: import matplotlib.pyplot as plt
     plt.plot(x, np.sin(x), 'r--')
     plt.show()

[ ]:
```

Figure1.1: JupyterLab at a glance

In JupyterLab, we can work with:

- Jupyter notebooks[38] – `.ipynb` documents combining code, text, plots, and other rich outputs; importantly, code chunks can be created, modified, and run interactively;

[36] https://jupyterlab.readthedocs.io/en/stable/
[37] https://github.com/jupyter/jupyter/wiki/Jupyter-kernels
[38] https://jupyterlab.readthedocs.io/en/stable/user/notebook.html

the results together with discussion in textual form can be embedded inside the documents, which makes it a good reporting tool for our data science needs;

- code consoles – terminals for running code chunks interactively (read-eval-print loop);
- source files for many different languages – with syntax highlighting and the ability to send code to the associated consoles;

and many more.

Exercise 1.2 *Head to the official documentation[39] of the JupyterLab project and watch the introductory video linked in the* Overview *section.*

Note: (**) JupyterLab is the definitely not the most convenient environment for doing real data science in Python (writing standalone scripts is the preferred option), but we have chosen it here because of its educative advantages (interactive, easy to start with, etc.).

If you are a more advanced student, you can consider, for example, **`jupytext`**[40] as a means to create `.ipynb` files directly from Markdown files.

1.2.1 Launching JupyterLab

How we launch JupyterLab will vary from system to system and everyone needs to determine what is the best way to do it by themselves.

1. Users of the Anaconda distribution should be able to start JupyterLab via the Anaconda Navigator[41], which can be accessed through the Start menu/application launcher. Read more in the official manual[42].
2. Alternatively, open the Anaconda Prompt (Windows) or start the ordinary system terminal (**`bash`**, **`zsh`**, etc.) and type:

```
cd your/favourite/directory  # change directory
jupyter lab  # or jupyter-lab, depending on the system
```

This should launch the JupyterLab server and open the corresponding web app in your default web browser.

[39] https://jupyterlab.readthedocs.io/en/stable/index.html
[40] https://jupytext.readthedocs.io/en/latest/
[41] https://docs.anaconda.com/anaconda/navigator/getting-started/#navigator-starting-navigator
[42] https://docs.anaconda.com/anaconda/user-guide/getting-started/#run-python-in-a-jupyter-notebook

Note: Some commercial cloud-hosted instances or forks of the open source JupyterLab project are available on the market, but we endorse none of them (even though some of them are provided gratis, there are always strings attached). It is best to run our applications locally, where we are free[43] to be in full control over the software environment.

1.2.2 First Notebook

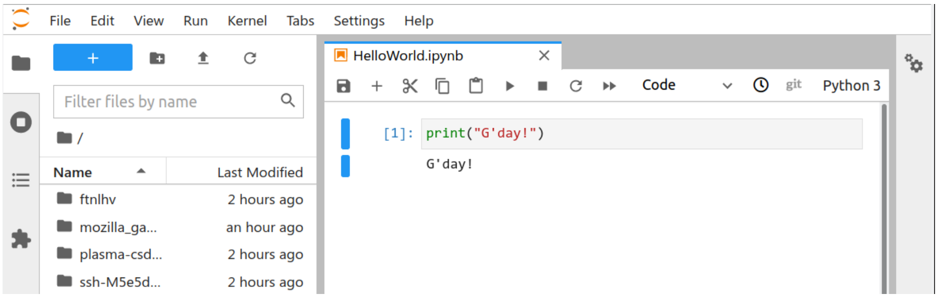
1. From JupyterLab, create a new notebook running a Python 3 kernel (for example, by selecting File → New → Notebook in the web app menu).
2. Select File → Rename Notebook and change the filename to `HelloWorld.ipynb`.

   Important: The file is stored relative to the current working directory of the running JupyterLab server instance. Make sure you are able to locate `HelloWorld.ipynb` on your disk using your favourite file explorer (by the way, `.ipynb` is just a JSON file which can also be edited using an ordinary text editor).

3. Input the following in the code cell:

   ```python
   print("G'day!")
   ```

4. Press `Ctrl+Enter` (or `Cmd+Return` on macOS) to execute the code cell and display the result; see also Figure 1.2.



```
[1]: print("G'day!")
     G'day!
```

Figure1.2: "Hello World" in a Jupyter Notebook

[43] https://www.youtube.com/watch?v=Ag1AKIl_2GM

1.2.3 More Cells

1. By pressing `Enter`, we can enter the *Edit mode*. Modify the cell's contents so that it now reads:

   ```
   # My first code cell (this is a comment)
   print("G'day!")  # prints a message (this is a comment too)
   print(2+5)  # prints a number
   ```

2. Press `Ctrl+Enter` to execute the code and replace the results with the new ones.

3. Change **`print`**`(2+5)` to **`PRINT`**`(2+5)`. Also, enter a command to print some other message that is to your liking. Note that character strings in Python must be enclosed in either double quotes or apostrophes.

4. Press `Shift+Enter`. This will not only execute the code cell, but also create a new one below and enter the edit mode.

5. In the new cell, enter and then execute the following:

   ```
   import matplotlib.pyplot as plt  # basic plotting library
   plt.bar(
       ["Python", "JavaScript", "HTML", "CSS"],  # a list of strings
       [80, 30, 10, 15]  # a list of integers (the corresponding bar heights)
   )
   plt.title("What makes you happy?")
   plt.show()
   ```

6. Add 3 more code cells, displaying some text or creating other bar plots.

Note: In the *Edit* mode, JupyterLab behaves like an ordinary text editor. Keyboard shortcuts probably known from elsewhere are available, for example:

- `Shift+LeftArrow`, `DownArrow`, `UpArrow`, or `RightArrow` – select text,
- `Ctrl+c` – copy,
- `Ctrl+x` – cut,
- `Ctrl+v` – paste,
- `Ctrl+z` – undo,
- `Ctrl+]` – indent,
- `Ctrl+[` – dedent,

- `Ctrl+/` – toggle comment.

1.2.4 Edit vs Command Mode

1. By pressing `ESC`, we can enter the *Command mode*.

   Important: We will be using `ESC` and `Enter` to switch between the *Command* and *Edit* modes, respectively.

2. In the *Command mode*, we can use the arrow `DownArrow` and `UpArrow` keys to move between the code cells.
3. In the *Command mode* pressing `d,d` (`d` followed by another `d`) deletes the currently selected cell.
4. Press `z` to undo the last operation.
5. Press `a` and `b` to insert a new blank cell above and below the current one, respectively.
6. Note that cells can be moved by a simple drag and drop with your mouse.

1.2.5 Markdown Cells

So far we have been playing with *Code* cells.

We can turn the current cell to become a Markdown cell by pressing `m` in the *Command mode*. Note that by pressing `y` we can turn it back to a *Code* cell.

Markdown is a lightweight, human-readable markup language widely used for formatting text documents.

1. Enter the following into a new Markdown cell:

```
# Section

## Subsection

This ~~was~~ *is* **really** nice.

* one
* two
    1. aaa
```

(continues on next page)

(continued from previous page)

~~~
    2. bbbb
* three

```python
# some code to display (but not execute)
2+2
```

![Python](https://www.python.org/static/img/python-logo.png)
~~~

2. Press `Ctrl+Enter` to display the formatted text.
3. Note that the Markdown call can be modified by entering the *Edit mode* as usual (`Enter` key).

Exercise 1.3 *Read the official introduction*[44] *to the Markdown syntax.*

Exercise 1.4 *Follow this*[45] *interactive Markdown tutorial.*

Exercise 1.5 *Apply what you have learned by making the current Jupyter notebook more readable. Add a header at the beginning of the report featuring your name and email address. Before and after each code cell, explain (in your own words) what we are about to do and how to interpret the obtained results.*

1.3 Initialising Each Session and Getting Example Data (!)

From now on, we assume that at the beginning of each session, the following commands have been issued:

```python
import numpy as np
import scipy.stats
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
plt.style.use("seaborn")  # plot style
pd.set_option("display.notebook_repr_html", False)  # disable "rich" output
```

(continues on next page)

[44] https://daringfireball.net/projects/markdown/syntax
[45] https://commonmark.org/help/tutorial/

(continued from previous page)

```
np.random.seed(123)  # initialise the pseudorandom number generator
plt.rcParams.update({
    "font.size": 12,
    "font.family": "sans-serif",
    "font.sans-serif": ["Alegreya Sans", "Alegreya"]
})
```

This imports the most frequently used packages (together with their common aliases, we'll get to that later) and sets up some further options that we particularly like. For the discussion on reproducible pseudorandom number generation, see Section 7.4.2.

The software we use receives feature upgrades, API changes, and bug fixes on a regular basis, therefore it is good to know which version of the Python environment was used to evaluate all the code included therein:

```
import sys
print(sys.version)
## 3.9.7 (default, Sep 10 2021, 14:59:43)
## [GCC 11.2.0]
```

The versions of the packages that we use in this course are given below. On a side note, they can be fetched by querying, for example, `np.__version__`, etc.

Package	Version
numpy	1.22.2
scipy	1.7.3
pandas	1.4.1
matplotlib	3.5.1
seaborn	0.11.2
sklearn	0.24.1

Important: All datasets that we use as examples are available for download at https://github.com/gagolews/teaching_data.

Exercise 1.6 *Make sure you are comfortable with accessing raw data files from the above repository. Chose any file, e.g.,* `nhanes_adult_female_height_2020.txt` *in the* `marek` *folder, and then click* Raw. *It is the URL that you have been redirected to, not the previous one, that includes information from within your Python session.*

Also note that each dataset starts with a number of comment lines explaining its structure, the meaning of the variables, etc.

1.4 Further Reading

Markdown is one of many markup languages. Other learn-worthy ones include HTML (for the Web) and LaTeX (especially for beautiful typesetting of maths, print-ready articles and books, e.g., PDF; see [[O+21]] for a good introduction).

Jupyter Notebooks can be converted to different formats using the **`jupyter-nbconvert`**[46] command line tool.

More generally, **`pandoc`**[47] is a generic converter between different formats, e.g., it can be used to convert between the highly universal (although primitive) Markdown and the said LaTeX and HTML. Also, it can be used for preparing presentations.

1.5 Exercises

Exercise 1.7 *What is the difference between the* Edit *and the* Command *mode in Jupyter?*

Exercise 1.8 *What is Markdown?*

Exercise 1.9 *How to format a table in Markdown?*

[46] https://pypi.org/project/nbconvert/
[47] https://pandoc.org/

2

Scalar Types and Control Structures in Python

In this part we quickly introduce the basics of the Python language itself. Being a general purpose tool, various packages supporting data wrangling operations are provided as third-party extensions (of course, all of them are free and open source). Based on the discussed concepts, we will be able to use **`numpy`**, **`scipy`**, **`pandas`**, **`sklearn`**, **`matplotlib`**, **`seaborn`**, and other packages with some healthy degree of confidence.

2.1 Scalar Types

The five ubiquitous scalar (i.e., *single* or *atomic* value) types are:

- `bool` – logical,
- `int`, `float`, `complex` – numeric,
- `str` – character.

2.1.1 Logical Values

There are only two possible logical (Boolean) values: `True` and `False`.

We can type:

```
True
## True
```

and instantiate one of them. This might seem boring, unless when trying to play with the above code by themself, the kind reader fell into the following pitfall:

Important: Python is case-sensitive. Writing “`TRUE`” or “`true`” instead of “`True`” results in an error.

2.1.2 Numeric Values

The three numeric scalar types are:

- `int` — integers, e.g., `1`, `-42`, `1_000_000`;
- `float` — floating-point (real) numbers, e.g., `-1.0`, `3.14159`, `1.23e-4`;
- `complex` (*) – complex numbers, e.g., `1+2j` (these are rather infrequently used in our applications).

In practice, `int` and `float` often work together seamlessly and we usually do not have to think about their being distinctive types.

Exercise 2.1 *`1.23e-4` and `9.8e5` are examples of numbers entered using the so-called scientific notation, where "`e`" stands for "times 10 to the power of". Moreover, `1_000_000` is a decorated (more human-readable) version of `1000000`. Use the `print` function to check their values.*

Arithmetic Operators

Here is the list of available arithmetic operators:

```
1 + 2     # addition
## 3
1 - 7     # subtraction
## -6
4 * 0.5   # multiplication
## 2.0
7 / 3     # float division (the result is always of type float)
## 2.3333333333333335
7 // 3    # integer division
## 2
7 % 3     # division remainder
## 1
2 ** 4    # exponentiation
## 16
```

The precedence of these operators is quite predictable[48], e.g., exponentiation has higher priority than multiplication and division which in turn binds more strongly than addition and subtraction. Hence:

```
1 + 2 * 3 ** 4  # the same as 1+(2*(3**4))
## 163
```

[48] https://docs.python.org/3/reference/expressions.html#operator-precedence

is different than, e.g., `((1+2)*3)**4)`.

Creating Named Variables

Named variables can be introduced using the *assignment operator*, `` `=` ``. They can store arbitrary Python objects and be referred to at any time.

```
x = 7  # read: let `x` from now on be equal to 7
print(x)
## 7
```

New variables can be created based on other ones:

```
y = x*x-10  # creates `y`
print(y)
## 39
```

Also, existing variables can be re-bound to any other value whenever we please:

```
x = x/3  # let new `x` be equal to the old `x` (7) divided by 3
print(x)
## 2.3333333333333335
```

Keep in mind that computers' floating-point arithmetic is precise only up to a few significant digits, hence the results we generate will often be approximate.

Exercise 2.2 *Create two named variables* `height` *and* `width`*. Based on them, determine your BMI*[49].

Note: (*) Augmented assignments are also available. For example:

```
x *= 3  # the same as: x = x*3
print(x)
## 7.0
```

They often modify the objects they act upon in-place, compare Section 3.5.

---

[49] https://en.wikipedia.org/wiki/Body_mass_index

2.1.3 Character Strings

Character strings (objects of type `str`), which can consist of arbitrary text, are created using either double quotes or apostrophes:

```
print("spam, spam, #, bacon, and spam")
print("□□□ Cześć □□ ¿Qué tal? Привет □□□□")
## spam, spam, #, bacon, and spam
## □□□ Cześć □□ ¿Qué tal? Привет □□□□
print('"G\'day, howya goin\'," he asked.\n"Fine, thanks," I responded.')
## "G'day, howya goin'," he asked.
## "Fine, thanks," I responded.
```

Above, "`\'`" (a way to include an apostrophe in an apostrophe-delimited string) and "`\n`" (newline character) are examples of *escape sequences*[50].

Multiline strings are also possible:

```
"""
spam\\spam
This is Python, therefore references
to Monty Python's Flying Circuseses are a must
sorry not sorry
tasty\t"spam"
lovely\t'spam'
"""
## '\nspam\\spam\nThis is Python, therefore references\nto Monty Python\'s□
↪Flying Circuseses are a must\nsorry not sorry\ntasty\t"spam"\nlovely\t\
↪'spam\'\n'
```

Exercise 2.3 *Call the **`print`** function on the above object to reveal the special meaning of the included escape sequences.*

Important: There are many string operations available – related for example to formatting, pattern searching, extracting chunks; they are very important in the art of data wrangling as oftentimes information comes to us in textual form. We shall be covering this topic in great detail in Chapter 15.

---

[50] https://docs.python.org/3/reference/lexical_analysis.html#string-and-bytes-literals

F-Strings (Formatted String Literals)

Also, the so-called *f-strings* (formatted string literals) can be used to prepare nice output messages:

```
x = 2
f"x is {x}"
## 'x is 2'
```

Note the `f` prefix. The `{x}` part was replaced with the value stored in the `x` variable.

There are many options available and it's best – as usual – to study the documentation[51] in search for interesting features. Here, let us just mention that we will frequently be referring to placeholders like `{variable:width}` and `{variable:width.precision}`, which specify the field width and the number of fractional digits of a number. This can result in a series of values nicely aligned one below another.

```
π = 3.14159265358979323846
e = 2.71828182845904523536
print(f"""
π = {π:10.8f}
e = {e:10.8f}
""")
##
## π = 3.14159265
## e = 2.71828183
```

`10.8f` means that a value is to be formatted as a `float`, be of at least width 10, and use 8 fractional digits.

2.2 Calling Built-in Functions

There are quite a few built-in functions ready for use. For instance:

```
e = 2.718281828459045
round(e, 2)
## 2.72
```

Rounds `e` to 2 decimal digits.

[51] https://docs.python.org/3/reference/lexical_analysis.html#f-strings

Exercise 2.4 *Call* ***help("round")*** *or* *?round* *(this is IPython/Jupyter-specific) to access the function's manual. Note that the second argument, called* ***ndigits****, which we have set to 2, has a default value of* None*. Check what happens when we omit it during the call.*

2.2.1 Positional and Keyword Arguments

As **round** has two parameters, number and ndigits, the following (and no other) calls are equivalent:

```
round(e, 2)  # two arguments matched positionally
round(e, ndigits=2)  # positional and keyword argument
round(number=e, ndigits=2)  # 2 keyword arguments
round(ndigits=2, number=e)  # order does not matter for keyword arguments
## 2.72
## 2.72
## 2.72
## 2.72
```

That no other form is allowed is left as an exercise, i.e., positionally matched arguments must be listed before the keyword ones.

2.2.2 Modules and Packages

Other functions are available in numerous Python modules and packages (which are collections of modules).

For example, **math** features many mathematical functions:

```
import math   # the math module must be imported prior its first use
print(math.log(2.718281828459045))  # the natural logarithm (base e)
## 1.0
print(math.sin(math.pi))  # sin(pi) equals 0 (with some numeric error)
## 1.2246467991473532e-16
print(math.floor(-7.33))  # the floor function
## -8
```

See the official documentation[52] for the comprehensive list of objects defined therein. Also, we assume that the reader is familiar with basic mathematical objects such as the said floor function. If not, please consult the English Wikipedia which is overall a reliable source of knowledge on topics from our discipline.

[52] https://docs.python.org/3/library/math.html

2.2.3 Slots and Methods

Python is an object-oriented programming language. Each object is an instance of some *class* whose name we can reveal by calling the **type** function:

```
x = 1+2j
type(x)
## <class 'complex'>
```

Important: Classes define the following kinds of *attributes*:

- *slots* – associated data,
- *methods* – associated functions.

Exercise 2.5 *Call* **help**("complex") *to reveal that the* complex *class features, amongst others, the* **conjugate** *method and the* real *and* imag *slots.*

Here is how we can access the two slots:

```
print(x.real)  # access the `real` slot of object `x` of class `complex`
## 1.0
print(x.imag)
## 2.0
```

And here is an example of a method call:

```
x.conjugate()  # equivalently: complex.conjugate(x)
## (1-2j)
```

Importantly, the user manual on this function can be accessed by typing **help**("complex.conjugate") *(class name – dot – method name)*.

2.3 Controlling Program Flow

2.3.1 Relational and Logical Operators

Further, we have a number of operators which return a single logical value:

```
1 == 1.0  # is equal to?
## True
2 != 3  # is not equal to?
## True
"spam" < "eggs" # is less than? (compare using the lexicographic order)
## False
```

Some more examples:

```
math.sin(math.pi) == 0.0  # well, numeric error...
## False
abs(math.sin(math.pi)) <= 1e-9  # is close to 0?
## True
```

Logical results might be combined using **and** (conjunction; for testing if both operands are true) and **or** (alternative; for determining whether at least one operand is true). Furthermore, **not** (negation) is available too.

```
3 <= math.pi and math.pi <= 4
## True
not (1 > 2 and 2 < 3) and not 100 <= 3
## True
```

Note that `not 100 <= 3` is equivalent to `100 > 3`. Also, via the de Morgan's laws, `not (1 > 2 and 2 < 3)` is true if and only if `1 <= 2 or 2 >= 3` holds.

Exercise 2.6 *Assuming that* `p`, `q`, `r` *are logical and* `a`, `b`, `c`, `d` *are float-type variables, simplify the following expressions:*

- `not not p`,
- `not p and not q`,
- `not (not p or not q or not r)`,
- `not a == b`,
- `not (b > a and b < c)`,
- `not (a>=b and b>=c and a>=c)`,
- `(a>b and a<c) or (a<c and a>d)`.

2.3.2 The `if` Statement

The **`if`** statement allows us to execute a chunk of code conditionally – whether the provided expression is true or not.

```
import numpy as np    # imports the `numpy` package and sets an alias
x = np.random.rand()  # pseudorandom value in [0.0, 1.0)
print(x)
if x < 0.5: print("spam!")
## 0.6964691855978616
```

Note the colon after the tested condition.

Further, a number of **`elif`** (*else-if*) parts can be added followed by an optional **`else`** part, which is executed if all the conditions tested are not true.

```
if x < 0.25:   print("spam!")
elif x < 0.5:  print("ham!")    # i.e., x in [0.25, 0.5)
elif x < 0.75: print("bacon!")  # i.e., x in [0.5, 0.75)
else:          print("eggs!")   # i.e., x >= 0.75
## bacon!
```

If more than one statement is to be executed conditionally, an indented code block can be introduced.

```
if x >= 0.25 and x <= 0.75:
    print("spam!")
    print("I love it!")
else:
    print("ham!")
    print("I'd rather eat spam!")
print("more spam!")  # executed regardless of the condition tested
## spam!
## I love it!
## more spam!
```

Important: We recommend using four spaces. The indentation must be neat and consistent. The reader is encouraged to try to execute the following code chunk and note what kind of error is generated:

```
if x < 0.5:
    print("spam!")
  print("ham!")    # :(
```

Exercise 2.7 *For a given BMI (see one of the above exercises), print out the corresponding category as defined by the WHO (underweight if below 18.5, normal range up to 25.0, etc.). Note that the BMI is a very simple measure and both the medical and statistical community point out its inherent limitations, read the Wikipedia article thereon for more details (and note there was a lot of data wrangling involved in its preparation – tables, charts, calculations; something that we will be able to do quite soon — give good reference data, of course).*

Exercise 2.8 *(*) Check if it is easy to find on the internet (at reliable sources) some raw data sets related to the body mass studies, e.g., measuring subjects' height, weight, body fat and muscle percentage, etc.*

2.3.3 The `while` Loop

The **`while`** loop executes a given statement or a series of statements as long as a given condition is true.

For example, here is a simple simulator determining how long we have to wait until drawing the first number not greater than 0.01 whilst generating numbers in the unit interval:

```
count = 0
while np.random.rand() > 0.01:
    count = count + 1
print(count)
## 117
```

Exercise 2.9 *Using the **`while`** loop, determine the arithmetic mean of 10 random numbers (i.e., the sum of the numbers divided by 10).*

2.4 Defining Own Functions

We can also define our own functions as a means for code reuse.

For instance, here is one that determines the minimum (with respect to the `<` relation) of three given objects:

```
def min3(a, b, c):
    """
    A function to determine the minimum of three
    given inputs.

    (this is the so-called docstring –
        call help("min3") later)
    """
    if a < b:
        if a < c:
            return a
        else:
            return c
    else:
        if b < c:
            return b
        else:
            return c
```

Example calls:

```
print(min3(10, 20, 30),
      min3(10, 30, 20),
      min3(20, 10, 30),
      min3(20, 30, 10),
      min3(30, 10, 20),
      min3(30, 20, 10))
## 10 10 10 10 10 10
```

Note that the function *returns* a value. Hence, the result can be fetched and used in further computations.

```
x = min3(np.random.rand(), 0.5, np.random.rand())  # minimum of 3 numbers
x = round(x, 3)  # do something with the result
print(x)
## 0.5
```

Exercise 2.10 *Write a function named* **`bmi`** *which computes and returns a person's BMI, given* *their weight (in kg) and height (in cm). As it is a good development practice to document your functions, do not forget about including the appropriate docstring.*

We can also introduce new variables inside a function's body. This can aid the function in doing the right thing, i.e., what it has been designed to do.

```
def min3(a, b, c):
    """
    A function to determine the minimum of three
    given inputs (alternative version).
    """
    m = a  # a local (temporary auxiliary) variable
    if b < m:
        m = b
    if c < m:    # be careful! no `else` or `elif` here — it's a separate `if`
        m = c
    return m
```

Example call:

```
m = 7
n = 10
o = 3
min3(m, n, o)
## 3
```

All *local variables* cease to exist after the function is called. Also note that `m` inside the function is a variable independent of `m` in the global (calling) scope.

```
print(m)  # this is still the global `m` from before the call
## 7
```

Exercise 2.11 *Write a function* **`max3`** *which determines the maximum of 3 given values.*

Exercise 2.12 *Write a function* **`med3`** *which defines the median of 3 given values (the one value that is in-between the other ones).*

Exercise 2.13 *(*) Write a function* **`min4`** *to compute the minimum of 4 values.*

2.5 Exercises

Exercise 2.14 *What does `import xxxxxx as x` mean?*

Exercise 2.15 *What is the difference between `if` and `while`?*

Exercise 2.16 *Name the scalar types we have introduced here.*

Exercise 2.17 *What is a docstring and how to create and access it?*

Exercise 2.18 *What are keyword arguments?*

3

Sequential and Other Types in Python

3.1 Sequential Types

Sequential objects store data items that can be accessed by index (position). The three main types of sequential objects are: lists, tuples, and ranges.

Actually, strings (which we often treat as scalars) can also be classified as such. Therefore, the four main types of sequential objects are:

- lists,
- tuples,
- ranges, and
- strings.

3.1.1 Lists

Lists consist of arbitrary Python objects and are created using square brackets:

```
x = [True, "two", 3, [4j, 5, "six"]]
x
## [True, 'two', 3, [4j, 5, 'six']]
```

Above is an example list featuring objects of type: `bool`, `str`, `int`, and `list` (yes, it is possible to have a list inside another list), in this order.

Note: We will often be using lists when creating vectors in **`numpy`** or data frame columns in **`pandas`**. Further, lists of lists of equal lengths can be used to create matrices.

Each list is *mutable* and hence its state may be changed arbitrarily. For instance, we can append a new object at its end:

```
x.append("spam")
print(x)
## [True, 'two', 3, [4j, 5, 'six'], 'spam']
```

3.1.2 Tuples

Next, *tuples* are like lists, but they are *immutable* (read-only) – once created, their state cannot be changed.

```
("one", [], (3j, 4, "five", "six"))
## ('one', [], (3j, 4, 'five', 'six'))
```

This gave us a triple (3-tuple) featuring a string, an empty list, and a 4-tuple.

Furthermore, note that we can drop the round brackets and still get a tuple:

```
1, 2, 3  # the same as `(1, 2, 3)`
## (1, 2, 3)
```

Also:

```
1,  # i.e., `(1,)`
## (1,)
```

Note that the above is not the same as simply `1` or `(1)`, representing an object of type `int`.

Note: Having a separate data type representing an immutable sequence actually makes sense in certain contexts. For example, a data frame's *shape* is its inherent property that should not be tinkered with. If a tabular dataset has 10 rows and 5 columns, we should not allow the user to set the former to 15 (without making further assumptions, providing extra data, etc.).

When creating collections of items, usually we prefer lists as they are more flexible a data type, but in Section 3.4.2 we note that many functions return tuples, therefore we should be able to handle them with confidence.

3.1.3 Ranges

Moreover, *ranges* are of the form **range**(from, to) or **range**(from, to, by) and represent arithmetic progressions of integers.

For the sake of illustration, we convert them to lists below:

```
list(range(0, 5))  # i.e., range(0, 5, 1) - from 0 to 5 (exclusive) by 1
## [0, 1, 2, 3, 4]
list(range(10, 0, -1))  # from 10 to 0 (exclusive) by -1
## [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Note that the rightmost boundary (to) is exclusive and that by defaults to 1.

3.1.4 Strings (Again)

Recall that we have already discussed character strings in Section 2.1.3.

```
"example\nstring"
## 'example\nstring'
```

Strings are often treated as scalars (atomic entities, as in: a sting as a whole). However, as we will soon find out, the individual characters are also accessible by index.

Furthermore, in Chapter 15 we will discuss numerous operations on text.

3.2 Working with Sequences

3.2.1 Extracting Elements

The index operator, `**[...]**`, can be applied on any sequential object to extract an element at a position specified by a single integer.

```
x = ["one", "two", "three", "four", "five"]
x[0]  # the first element
## 'one'
x[len(x)-1]  # the last element
## 'five'
```

Therefore, the valid indices are 0, 1, …, n-2, n-1, where n is the length of the sequence.

Important: Think of an index as the distance from the start of a sequence. For example, `x[3]` means "3 items away from the beginning", hence, the 4th element.

Negative indices count from the end:

```
x[-1]  # the last element
## 'five'
x[-2]  # the second to last
## 'four'
x[-len(x)]  # the first element; `len(x)` gives the number of elements in `x`
## 'one'
```

The index operator can be applied on any sequential object:

```
"string"[3]
## 'i'
```

Note that indexing a string returns a string – that is why we have classified strings as scalars too.

More examples:

```
range(0, 10)[-1]  # the last item in an arithmetic progression
## 9
(1, )[0]  # extract from a 1-tuple
## 1
```

3.2.2 Slicing

We can also use slices of the form `from:to` or `from:to:by` to select a subsequence of a given sequence. Slices are similar to ranges, but `:` can only be used within square brackets.

```
x = ["one", "two", "three", "four", "five"]
x[1:4]  # from 2nd to 5th (exclusive)
## ['two', 'three', 'four']
x[-1:0:-2]  # from last to first (exclusive) by every 2nd backwards
## ['five', 'three']
```

Actually, `from` and `to` are optional – when omitted, they default to one of the sequence boundaries.

```
x[3:]  # from 3rd to end
## ['four', 'five']
x[::2]  # every 2nd from the start
## ['one', 'three', 'five']
x[::-1]  # elements in the reversed order
## ['five', 'four', 'three', 'two', 'one']
```

And, of course, they can be applied on other sequential objects as well:

```
"spam, bacon, spam, and eggs"[13:17]
## 'spam'
```

Important: It is crucial to know the difference between element extraction and subsetting a sequence (creating a subsequence).

For example:

```
x[0]  # extraction (indexing with a single integer)
## 'one'
```

gives the object *at* that index.

```
x[0:1]  # subsetting (indexing with a slice)
## ['one']
```

gives the object of the same type as `x` (here, a list) featuring the items at that indices (in this case, only the first object, but a slice can potentially select more than one element).

pandas data frames and **numpy** arrays will behave similarly, but there will be many more indexing options (as discussed in Section 6.3, Section 9.5, and Section 11.5).

3.2.3 Modifying Elements

Lists are *mutable* – their state may be changed. The index operator can be used to replace the elements at given indices.

```
x = ["one", "two", "three", "four", "five"]
x[0] = "spam"  # replace the first element
```

(continues on next page)

(continued from previous page)

```
x[-3:] = ["bacon", "eggs"]  # replace last three with given two
print(x)
## ['spam', 'two', 'bacon', 'eggs']
```

Exercise 3.1 *There are quite a few methods which we can use to modify list elements:* ***append****,* ***insert****,* ***remove****,* ***pop****, etc. Invoke* **help**`("list")` *to access their descriptions and call them on a few example lists.*

3.2.4 Searching Elements

The **in** operator and its negation, **not in**, determine whether an element exists in a given sequence:

```
7 in range(0, 10)
## True
[2, 3] in [ 1, [2, 3], [4, 5, 6] ]
## True
```

For strings, **in** tests whether a string features a specific *substring*, so we don't have to restrict ourselves to single characters:

```
"spam" in "lovely spams"
## True
```

Exercise 3.2 *Check out the* ***count*** *and* ***index*** *methods in the* `list` *and other classes.*

3.2.5 Arithmetic Operators

Some arithmetic operators have been *overloaded* for certain sequential types, but they of course carry different meanings than those for integers and floats.

In particular, `+` can be used to join (concatenate) strings, lists, and tuples:

```
"spam" + " " + "bacon"
## 'spam bacon'
[1, 2, 3] + [4]
## [1, 2, 3, 4]
```

and `*` duplicates (recycles) a given sequence:

```
"spam" * 3
## 'spamspamspam'
(1, 2) * 4
## (1, 2, 1, 2, 1, 2, 1, 2)
```

3.3 Dictionaries (*)

Dictionaries (objects of type `dict`) are sets of `key: value` pairs, where the `values` (any Python object) can be accessed by `key` (usually a string).

```
x = {
    "a": [1, 2, 3],
    "b": 7,
    "z": "spam!"
}
print(x)
## {'a': [1, 2, 3], 'b': 7, 'z': 'spam!'}
```

We can also create a dictionary with string keys using the `dict` function which expects a number of keyword arguments:

```
dict(a=[1, 2, 3], b=7, z="spam!")
## {'a': [1, 2, 3], 'b': 7, 'z': 'spam!'}
```

The index operator can be used to extract specific elements:

```
x["a"]
## [1, 2, 3]
```

Note that `x[0]` is not valid – it is not an object of sequential type; a key of `0` does not exist in a given dictionary.

The **`in`** operator can be used to check whether a given key exists:

```
"a" in x, 0 not in x, "z" in x, "w" in x  # a tuple of 4 check results
## (True, True, True, False)
```

We can also add new elements to a dictionary:

```
x["f"] = "more spam!"
print(x)
## {'a': [1, 2, 3], 'b': 7, 'z': 'spam!', 'f': 'more spam!'}
```

Example 3.3 *(*) In practice, we often import JSON files (which is popular data exchange format on the internet) exactly in the form of Python dictionaries. Let's demo it quickly:*

```
import requests
x = requests.get("https://api.github.com/users/gagolews/starred").json()
```

Now x is a sequence of dictionaries giving the information on the repositories starred by yours truly on GitHub. As an exercise, the reader is encouraged to inspect its structure.

3.4 Iterable Types

All the objects we have discussed here are *iterable*. In other words, we can iterate through each element contained therein.

In particular, the **`list`** and **`tuple`** *functions* take any iterable object and convert it to a sequence of the corresponding type, for instance:

```
list("spam")
## ['s', 'p', 'a', 'm']
tuple(range(0, 10, 2))
## (0, 2, 4, 6, 8)
list({ "a": 1, "b": ["spam", "bacon", "spam"] })
## ['a', 'b']
```

Exercise 3.4 *Take a look at the documentation of the* **`extend`** *method in the* `list` *class. The manual page suggests that this operation takes any iterable object. Feed it with a list, tuple, range, and a string and see what happens.*

The notion of iterable objects is important, as they appear in many contexts. There are quite a few other iterable types which are for example non-sequential (we cannot access their elements at random using the index operator).

Exercise 3.5 *(*) Check out the* **`enumerate`**, **`zip`**, *and* **`reversed`** *functions and what kind of iterable objects do they return.*

3.4.1 The for loop

The **for** loop iterates over every element in an iterable object, allowing us to perform a specific action. For example:

```
x = [1, "two", ["three", 3j, 3], False]  # some iterable object
for el in x:    # for every element in `x`, let's call it `el`
    print(el)  # do something on `el`
## 1
## two
## ['three', 3j, 3]
## False
```

Another example:

```
for i in range(len(x)):
    print(i, ":", x[i])
## 0 : 1
## 1 : two
## 2 : ['three', 3j, 3]
## 3 : False
```

One more example – computing the elementwise multiply of two vectors of equal lengths:

```
x = [1,  2,   3,    4,     5]  # for testing
y = [1, 10, 100, 1000, 10000]  # just a test
z = []  # result list - start with an empty one
for i in range(len(x)):
    z.append(x[i] * y[i])
print(z)
## [1, 20, 300, 4000, 50000]
```

Yet another example: here is a function which determines the minimum of a given iterable object (compare the built-in `min` function, see `help("min")`).

```
import math
def mymin(x):
    """
    The smallest element in an iterable object x.
    We assume that x consists of numbers only.
    """
```

(continues on next page)

(continued from previous page)

```
    curmin = math.inf  # infinity is greater than any other number
    for e in x:
        if e < curmin:
            curmin = e  # a better candidate for the minimum
    return curmin
```

Exercise 3.6 *Write your own basic versions (using the **`for`** loop) of the built-in **`max`**, **`sum`**, **`any`**, and **`all`** functions.*

Exercise 3.7 *(*) The **`glob`** function in the **`glob`** module can be used to list all files in a given directory whose names match a specific wildcard, e.g., **`glob.glob`**`("~/Music/*.mp3")` (note that `"~"` points to the current user's home directory). Moreover, **`getsize`** from the **`os.path`** module returns the size of a given file, in bytes. Write a function that determines the total size of all the files in a given directory.*

3.4.2 Tuple Assignment

We can create many variables in one line of code by using the syntax `tuple_of_identifiers = iterable_object`, which unpacks the iterable:

```
a, b, c = [1, "two", [3, 3j, "three"]]
print(a)
print(b)
print(c)
## 1
## two
## [3, 3j, 'three']
```

This is useful, for example, when the swapping of two elements is needed:

```
a, b = 1, 2  # the same as (a, b) = (1, 2)
a, b = b, a  # swap a and b
print(a)
print(b)
## 2
## 1
```

Another use case is where we fetch outputs of functions that return many objects at once. For instance, in Chapter 12 we will learn about **`numpy.unique`** which (depending on arguments passed) can return a tuple of arrays:

```
import numpy as np
result = np.unique([1, 2, 1, 2, 1, 1, 3, 2, 1], return_counts=True)
result
## (array([1, 2, 3]), array([5, 3, 1]))
```

That this is indeed a tuple of length two (which we suspect anyway – note the round brackets and two objects separated by a comma) can be verified as follows:

```
type(result), len(result)
## (<class 'tuple'>, 2)
```

Hence, we could have written instead:

```
values, counts = np.unique([1, 2, 1, 2, 1, 1, 3, 2, 1], return_counts=True)
```

This gives two separate variables, each storing a different array. And now:

```
values
## array([1, 2, 3])
counts
## array([5, 3, 1])
```

Also note that if only the second item was of our interest, we could have written:

```
counts = np.unique([1, 2, 1, 2, 1, 1, 3, 2, 1], return_counts=True)[1]
```

because a tuple is a sequential object.

Example 3.8 *(*) For example, knowing that the* **`dict.items`** *method generates an iterable object that can be used to traverse through all the* (key, value) *pairs:*

```
x = { "a": 1, "b": ["spam", "bacon", "spam"] }
print(list(x.items()))  # just a demo
## [('a', 1), ('b', ['spam', 'bacon', 'spam'])]
```

we can utilise tuple assignments in contexts such as:

```
for k, v in x.items():   # or: for (k, v) in x.items()...
    print(k, v, sep=": ")
## a: 1
## b: ['spam', 'bacon', 'spam']
```

Note: (**) Also note that, if there are too many values to unpack, we can use the notation like `*name`. This will serve as a placeholder that gathers all the remaining values and wraps them up in a list:

```
a, b, *c, d = range(10)
print(a, b, c, d, sep="\n")
## 0
## 1
## [2, 3, 4, 5, 6, 7, 8]
## 9
```

This placeholder may appear only once on the lefthand side of the assignment operator.

3.5 Object References and Copying (*)

3.5.1 Copying References

It is important to always keep in mind that when writing:

```
x = [1, 2, 3]
y = x
```

the assignment operator does not create a copy of `x`; both `x` and `y` refer to the same object in computer's memory.

Important: If `x` is mutable, any change made to it will of course affect `y` (as, again, they are two different means to access the same object).

Hence:

```
x.append(4)
print(y)
## [1, 2, 3, 4]
```

gives the same result as a call to `print(x)`.

3.5.2 Pass by Assignment

Arguments are passed to functions by assignment too. In other words, they behave as if `=` was used – what we get is another reference to the existing object.

```
def myadd(z, i):
    z.append(i)
```

And now:

```
myadd(x, 5)
myadd(y, 6)
print(x)
## [1, 2, 3, 4, 5, 6]
```

3.5.3 Object Copies

If we find the above behaviour undesirable, we can always make a copy of an object. It is customary for the mutable objects to be equipped with a relevant method:

```
x = [1, 2, 3]
y = x.copy()
x.append(4)
print(y)
## [1, 2, 3]
```

This did not change the object referred to as `y`, because it is now a different entity.

3.5.4 Modify In-Place or Return a Modified Copy?

We now know that we *can* have functions or methods that change the state of a given object. Hence, for all the functions we apply, it is important to read their documentation to determine if they modify their inputs in-place or return a completely new object. Although surely some patterns can be identified (such as: a method is likely to modify an object in-place whereas a similar standalone function would be returning a copy) – ultimately the functions' developers are free to come up with some exceptions to them if they deem it more sensible or convenient.

Consider the following examples. The `sorted` function returns a sorted version of the input iterable:

```
x = [5, 3, 2, 4, 1]
print(sorted(x))  # returns a sorted copy of x (does not change x)
print(x)  # unchanged
## [1, 2, 3, 4, 5]
## [5, 3, 2, 4, 1]
```

The `list.sorted` method modifies the list it is applied on in-place:

```
x = [5, 3, 2, 4, 1]
x.sort()  # modifies x in-place
print(x)
## [1, 2, 3, 4, 5]
```

To add to joy, **`random.shuffle`** is a function (not a method) that changes the state of the argument:

```
x = [5, 3, 2, 4, 1]
import random
random.shuffle(x)  # modifies x in-place
print(x)
## [1, 3, 5, 4, 2]
```

Moreover, later we will learn about `Series` objects in **`pandas`**, which represent data frame columns. They have the **`sort_values`** method which by default returns a sorted copy of the object it acts upon:

```
import pandas as pd
x = pd.Series([5, 3, 2, 4, 1])
print(list(x.sort_values()))  # inplace=False
print(list(x))  # unchanged
## [1, 2, 3, 4, 5]
## [5, 3, 2, 4, 1]
```

However, this behaviour might be changed:

```
x = pd.Series([5, 3, 2, 4, 1])
x.sort_values(inplace=True)
print(list(x))  # changed
## [1, 2, 3, 4, 5]
```

Important: We should always carefully study the official documentation (not some random page on the internet displaying us ads) of every function we call.

3.6 Further Reading

Our overview of the Python language is by no means comprehensive, however it touches upon the most important topics from the perspective of data wrangling.

We will be elaborating on many other language elements and its standard library in the course of this course (list comprehensions, lambda expressions, exception handling, string formatting, argument unpacking, variadic arguments like `*args` and `**kwargs`, regular expressions, file handling, etc.).

We have also decided *not* to introduce some language constructs which we can easily do without (e.g., **`else`** clauses on **`for`** and **`while`** loops, the **`match`** statement) or are perhaps too technical for an introductory course (**`yield`**, **`iter`** and **`next`**, sets, name binding scopes, deep copying of objects, defining own classes, function factories and closures).

Also, we skipped the constructs that do not work well with the third party packages we are soon going to be using (e.g., notation like `x < y < z` is not valid if the three variables are **`numpy`** vectors).

The said simplifications have been brought in so that the reader is not overwhelmed – we strongly advocate for minimalism in software development. Python is the basis of one of a few programming environments for exercising data science and, in the long run, it is best to focus on developing the most *transferable* skills.

We encourage at some point to skim through at least the following chapters of the official Python 3 tutorial[53]:

- 3. An Informal Introduction to Python[54],
- 4. More Control Flow Tools[55],
- 5. Data Structures[56].

[53] https://docs.python.org/3/tutorial/index.html
[54] https://docs.python.org/3/tutorial/introduction.html
[55] https://docs.python.org/3/tutorial/controlflow.html
[56] https://docs.python.org/3/tutorial/datastructures.html

3.7 Exercises

Exercise 3.9 *Name the sequential objects we have introduced.*

Exercise 3.10 *Is every iterable object sequential?*

Exercise 3.11 *Is `dict` an instance of a sequential type?*

Exercise 3.12 *What is the meaning of `` `+` `` and `` `*` `` operations on strings and lists?*

Exercise 3.13 *Given a list `x` featuring numeric scalars, how to create a new list of the same length giving the squares of all the elements in the former?*

Exercise 3.14 *(*) How to make an object copy and when we should do so?*

Exercise 3.15 *What is the difference between `x[0]` and `x[:0]`, where `x` is a sequential object?*

Part II

Unidimensional Data

4 Unidimensional Numeric Data and Their Distribution

Let's assume that we have dozens of data points at hand, representing some measurements, such as sensor readings for some industrial processes, body measures for patients in a clinic, salaries of employees, sizes of cities, etc.

For instance, consider the heights of adult females (>= 18 years old, in cm) in the longitudinal study called National Health and Nutrition Examination Survey (NHANES[57]) conducted by the US Centres for Disease Control and Prevention.

```
heights = np.loadtxt("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/nhanes_adult_female_height_2020.txt")
```

Let's preview a few first observations:

```
heights[:6]  # first six
## array([160.2, 152.7, 161.2, 157.4, 154.6, 144.7])
```

This is an example of *quantitative* (numeric) data in form of a series of numbers on which it makes sense to apply various mathematical operations such as subtraction, division, taking logarithms, comparing which one is greater than the other, and so forth.

Most importantly, here, all the observations are *independent* of each other; the ordering of values does not matter.

Our data sample consists 4221 points on the real line (a "bag" of points), see Figure 4.1, but merely looking at the numbers themselves tells us nothing. There are too many of them.

In this chapter we introduce basic ways to create vectors in the **numpy** package, which are an efficient data structure for storing and operating on numeric data just like the ones above. Then, we discuss some basic tools for inspecting their the distribution.

[57] https://wwwn.cdc.gov/nchs/nhanes/search/datapage.aspx

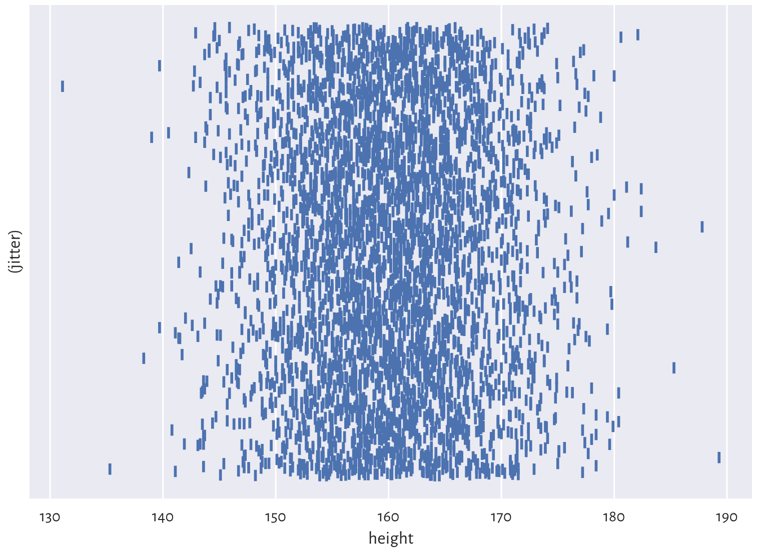


Figure4.1: The `heights` dataset is comprised of independent points on the real like; we have added some jitter on the y-axis for dramatic effects only

4.1 Creating Vectors in `numpy`

`numpy`[58] is an open source add-on for numerical computing written by Travis Oliphant and other developers in 2005 (although the project has a much longer history[59] and stands on the shoulders of many giants (e.g., concepts from the APL and Fortran languages). It adds support for multi-dimensional arrays and numerous operations on them, similar to those available in R, S, MATLAB, GNU Octave, Scilab, Julia, Perl (via **`Perl Data Language`**), and some numerical analysis libraries such as **`LAPACK`**, **`GNU GSL`**, etc.

Many other packages are built on top of **`numpy`** (including **`scipy`**, **`pandas`**, and **`sklearn`**), hence our studying it in very detail is utterly important. In particular, whatever we

[58] https://numpy.org/doc/stable/reference/index.html
[59] https://scipy.github.io/old-wiki/pages/History_of_SciPy

learn about vectors here will be transferable to the case of the processing of data frame columns.

Recall that Python lists, e.g., `[1, 2, 3]`, represent simple sequences of objects of any kind. Their use cases are very broad, which is both an advantage and something quite the opposite.

Vectors in **numpy** are 1-dimensional arrays which are like lists, but on steroids, which make them strong at scientific computing.

A vector of length n will often be used to represent a point in an n dimensional space (for example, GPS coordinates of a place on Earth assume n=2) or n readings of some 1-dimensional quantity (e.g., recorded heights of n people).

Note: (*) **numpy** vectors are powerful because of the underlying assumption that each object they store is of the same type. Although it is possible to save references to arbitrary objects therein, in majority of the cases we will be dealing with vectors of logical values, integers, and floating-point numbers.

Those are directly representable as simple arrays in the C programming language, in which **numpy** is written. Hence, operations on vectors will be very fast provided that we are using the functions working on them *as a whole*. Hence, the readers having some background in other lower-level languages, will need to change their habits of processing individual elements in a sequence, e.g., by means of a **for** loop.

It is customary to import the **numpy** package under the `np` alias:

```
import numpy as np
```

Hence, we can now refer to objects defined therein as `np.spam`, `np.bacon`, or `np.spam`.

4.1.1 Enumerating Elements

One way to create a vector is by calling the **numpy.array** function:

```
x = np.array([10, 20, 30, 40, 50, 60])
x
## array([10, 20, 30, 40, 50, 60])
```

Note that the vector elements were specified by means of an ordinary list. Ranges and tuples can also be used as content providers, which the kind reader is encouraged to check themself.

The length of the vector can either be read using the previously mentioned `len` function:

```
len(x)
## 6
```

or by reading the array's `shape` slot:

```
x.shape
## (6,)
```

Note that a vector is a 1-dimensional array, hence its shape is stored as a tuple of length 1 (the number of dimensions is stored as `x.ndim`). We can get its length by accessing `x.shape[0]`.

On a side note, matrices (2-dimensional arrays), which we shall study later, will be of shape like `(number_of_rows, number_of_columns)`.

All vector elements are generally of the same type. Above we have a sequence of integers:

```
x.dtype  # data type
## dtype('int64')
```

However, logical, float, complex, but also string vectors are possible too:

```
np.array([True, False, False, True])
## array([ True, False, False,  True])
x.astype(float)  # or np.array(x, dtype=float)
## array([10., 20., 30., 40., 50., 60.])
np.array(["spam", "spam", "bacon", "spam"])
## array(['spam', 'spam', 'bacon', 'spam'], dtype='<U5')
```

The in last example we have an array of strings in Unicode (i.e., capable of storing any character in any alphabet, emojis, mathematical symbols, etc.), each of no more than 5 code points in length. It is important to know this, because, as we will point out in Chapter 15, replacing any element with new content will result in the too long strings' truncation. This can be remedied by calling `x.astype("<U10")`, for example.

4.1.2 Arithmetic Progressions

`numpy`'s **`arange`** is similar to the built-in **`range`**, but outputs a vector:

```
np.arange(0, 10, 2)
## array([0, 2, 4, 6, 8])
```

`linspace` (*linear space*) creates a sequence of equidistant points in a given interval:

```
np.linspace(0, 1, 5)
## array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

Exercise 4.1 *Call* **`help`**`(np.linspace)` *and study the meaning of the* `endpoint` *argument. Also, find the same documentation page at the* **`numpy`** *project's website[60]. Another way is to use your favourite search engine such as DuckDuckGo and query "*`linspace site:numpy.org`*". Always remember to gather information from first hand sources! If you are serious about data science in Python, you will become a frequent visitor at this page (and similar ones). In particular, every so often it's a good idea to check out for significant updates at https://numpy.org/news/.*

4.1.3 Repeating Values

`numpy.repeat` repeats *each* value a number of times:

```
np.repeat(5, 6)
## array([5, 5, 5, 5, 5, 5])
np.repeat([1, 2], 3)
## array([1, 1, 1, 2, 2, 2])
np.repeat([1, 2], [3, 5])
## array([1, 1, 1, 2, 2, 2, 2, 2])
```

Here, each element from the 1st argument was repeated the *corresponding* number of times, as defined by the 2nd argument.

`numpy.tile`, on the other hand, repeats a whole sequence with *recycling*:

```
np.tile([1, 2], 3)
## array([1, 2, 1, 2, 1, 2])
```

See also **`numpy.zeros`** and **`numpy.ones`** for some specialised versions of the above.

4.1.4 Generating Pseudorandom Variates

The **`numpy.random`** module defines many functions to generate pseudorandom numbers. We will be discussing the reason for our using the *pseudo* prefix together with various

[60] https://numpy.org/doc/stable/reference/index.html

probability distributions in a further part, so let's now just quickly take note of a way to sample from the uniform distribution on the unit interval:

```
np.random.rand(5)
## array([0.49340194, 0.41614605, 0.69780667, 0.45278338, 0.84061215])
```

and picking values from a given set:

```
np.random.choice(np.arange(1, 10), 3, replace=False)
## array([1, 5, 8])
```

This gave 3 values from the set *{1, 2, ..., 9}*, without replacement (so that all the numbers are unique).

4.1.5 Loading Data from Files

We will usually be reading whole heterogeneous tabular data sets using **pandas.read_csv** being the topic we shall cover in Chapter 11.

It is worth knowing, though, that arrays with elements of the same type can be read efficiently from text (e.g., CSV) files using **numpy.loadtxt**, for example:

```
heights = np.loadtxt("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/nhanes_adult_female_height_2020.txt")
```

Exercise 4.2 *Use* ***numpy.loadtxt*** *to read this dataset*[61] *from GitHub (click* Raw *to get access to its contents).*

4.1.6 numpy.r_ (*)

numpy.r_ is perhaps the most flexible means for creating vectors involving quite a few of the aforementioned scenarios, however it has a quirky syntax.

For example:

```
np.r_[1, 2, 3, np.nan, 5, np.inf]
## array([ 1.,  2.,  3., nan,  5., inf])
```

Note that nan stands for a *not-a-number* and is used as a placeholder for missing values (discussed later) or *wrong* results, such as the square root of -1 in the domain of reals. inf, on the other hand, means *infinity*, ∞.

[61] https://github.com/gagolews/teaching_data/blob/master/marek/population_largest_cities_unnamed.txt

We see that square brackets are used instead of the round ones. This is actually smart, because we've mentioned that slices (`:`) can only be used inside them. And so they can:

```
np.r_[0:10:2]  # like np.arange(0, 10, 2)
## array([0, 2, 4, 6, 8])
```

What's more, **numpy** authors added the following syntactic sugar:

```
np.r_[0:1:5j]  # like np.linspace(0, 1, 5)
## array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

Finally, note that we can combine many chunks into one:

```
np.r_[1, 2, [3]*2, 0:3, 0:3:3j]
## array([1. , 2. , 3. , 3. , 0. , 1. , 2. , 0. , 1.5, 3. ])
```

4.2 Mathematical Notation

Mathematically, we will be denoting number sequences with

$$\boldsymbol{x} = (x_1, x_2, \dots, x_n),$$

where x_i is the i-th element therein and n is the size of the sample. The bold font (hopefully visible) is to emphasise that $\boldsymbol{x}$ is not an atomic entity (x), but rather a collection thereof.

For brevity, instead of saying "let $\boldsymbol{x}$ be a real-valued sequence[70] of length n", we shall write "let $\boldsymbol{x} \in \mathbb{R}^n$". Here, the "$\in$" symbol stand for "*is in*" or "*is a member of*" and $\mathbb{R}$ denotes the set of real numbers (the very one that features, 0, -358745.2394, 42 and π, amongst uncountably many others) and $\mathbb{R}^n$ is the set of real-valued sequences of length n (i.e., n such numbers considered at a time). For instance, $\mathbb{R}^2$ includes pairs such as $(1, 2)$, $(\pi/3, \sqrt{2}/2)$, and $(1/3, 10^3)$.

Note: Mathematical notation is pleasantly *abstract* (general) in the sense that $\boldsymbol{x}$ can be anything, e.g., data on the incomes of households, sizes of the largest cities in a given

[70] If $\boldsymbol{x} \in \mathbb{R}^n$, then we often say that $\boldsymbol{x}$ is *a sequence of n numbers*, *a (numeric) n-tuple*, *a n-dimensional real vector*, *a point in a n-dimensional real space*, or *an element of a real n-space*, etc. In many contexts, these are more or less synonyms of each other.

country, or heights of participants in a study. At a first glance, such a representation of objects from the so-called real world might seem overly simplistic, especially if we'd wish to store information on very complex entities. However, it turns out that in most cases expressing them as *vectors* (i.e., establishing a set of numeric attributes that best describe them in a task at hand) is not only natural but also perfectly sufficient for achieving whatever we aim at.

Exercise 4.3 *Consider the following problems:*

- *How would you represent a patient in a clinic (for the purpose of conducting research in cardiology)?*
- *How would you represent a car in an insurance company's database (for the purpose of determining how much a driver should pay annually for the mandatory policy)?*
- *How would you represent a student in an university (for the purpose of granting them scholarships)?*

In each case, list a few numeric features that best describe the reality of concern. Note that descriptive labels can always be encoded *as numbers, e.g., female = 1, male = 2.*

Using the programming syntax, n corresponds to `len(x)` or equivalently `x.shape[0]` and x_i is `x[i-1]` (because the first element is at index 0).

Furthermore, by $x_{(i)}$ (note the bracket) we will denote the i-th smallest value in $\boldsymbol{x}$ (some textbooks denote it with $x_{i:n}$, but we will not). In particular, $x_{(1)}$ is the minimum and $x_{(n)}$ is the maximum.

```
x = np.array([5, 4, 2, 1, 3])  # just an example
x_sorted = np.sort(x)
x_sorted[0], x_sorted[-1]  # the minimum and the maximum
## (1, 5)
```

We shall sometimes be referring to $x_{(i)}$ as the i-th *order statistic*.

Also, for to avoid the clutter of notation, in certain formulas we will be assuming that $x_{(0)}$ is the same as $x_{(1)}$ and $x_{(n+1)}$ is equivalent to $x_{(n)}$.

4.3 Histograms

Histograms are one of the most intuitive tools for depicting the empirical distribution of a data sample. We'll be drawing them using the statistical data visualisation package called **seaborn**[62] [[Was21]] (written by Michael Waskom), which was built on top of the classic plotting library **matplotlib**[63] [[Hun07]] (originally developed by John D. Hunter). Let's import both packages and set their traditional aliases:

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

Note: We will frequently be calling a single function from **seaborn** and then performing a series of additional calls to **matplotlib** to tweak the display details. It is important to remember that the former uses the latter to achieve its goals, not the other way around.

4.3.1 heights – A Bell-Shaped Distribution

Let's draw a histogram for the `heights` dataset, see Figure 4.2.

```
sns.histplot(heights, bins=11)
plt.show()
```

The data were split into 11 bins and plotted in such a way that bar heights are proportional to the number of observations falling into each interval.

The bins are non-overlapping, adjacent to each other, and of equal lengths and we can read their coordinates by looking at the bottom side of each rectangular bar.

This distribution is bell-shaped – nicely symmetrical around about 160 cm. It resembles a *normal* distribution (which we cover in Chapter 7) – the most typical (*normal*) observations are somewhere in the middle, and the probability mass decreases quickly. We will learn that, for instance, observations outside the (more or less) $[139, 181]$ interval (mean $\pm$ 3 standard deviations – the 3σ rule) are very rare (less than 1% probability).

Note: For some of us, a normal distribution is a prototypical one – we think of many phenomena to be distributed in this way. And that is indeed the case, e.g., in psychology (IQ or personality tests), physiology (the above heights), or when measuring stuff with not-so-precise devices (distribution of errors).

(*) In fact, we have a proposition saying that the sum or average of many observations, assuming that they are independent and follow the same (any!) distribution with finite

[62] https://seaborn.pydata.org
[63] https://matplotlib.org/

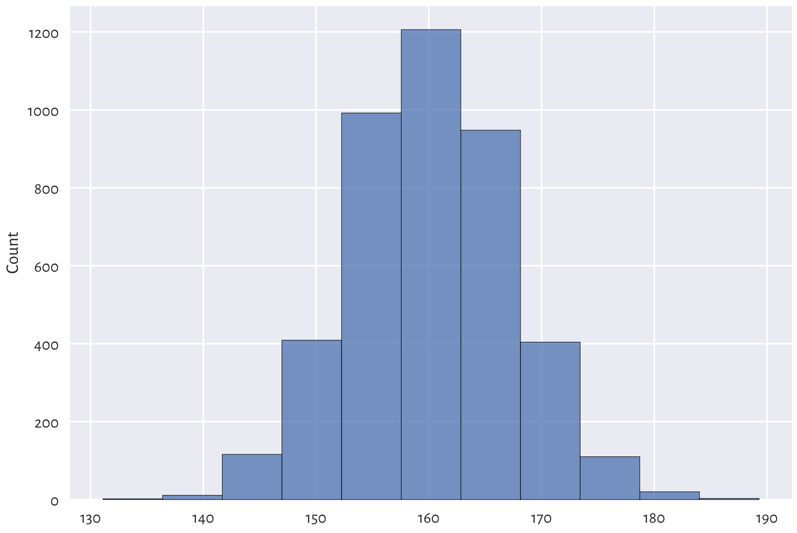


Figure4.2: Histogram for the `heights` dataset

variance, is approximately normally distributed. This is called the Central Limit Theorem and it is a very strong mathematical result.

We might be tempted to think now that everything is normally distributed, but this is very much untrue.

4.3.2 `income` – A Right-Skewed Distribution

Let us therefore consider another dataset. In Figure 4.3 we depict the distribution of a simulated sample of disposable income (financial year ending 2020) of 1000 randomly chosen UK households, in £ (of course, the UK Office for National Statistics does not provide us with details on each tax payer, for privacy and other reasons, hence we needed to recreate it based on data from this report[64]):

[64] https://www.ons.gov.uk/peoplepopulationandcommunity/personalandhouseholdfinances/incomeandwealth/bulletins/householddisposableincomeandinequality/financialyear2020

```
income = np.loadtxt("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/uk_income_simulated_2020.txt")
sns.histplot(income, stat="percent", bins=20)
plt.show()
```

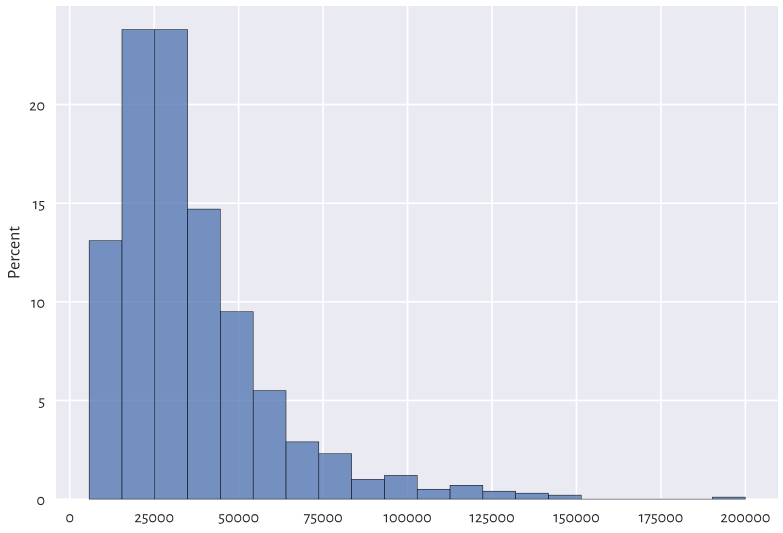


Figure4.3: Histogram for the `income` dataset

We have normalised (`stat="percent"`) the bar heights so that they all sum to 1 (or, equivalently, 100%), which resulted in a *probability* histogram.

Now we see that the probability density quickly increases, reaches peak at around £15,500–£35,000 and then slowly goes down. It has a *long tail* on the right (and hence it is *right- or positive-skewed*), which means that high salaries are quite likely.

Thus, it's quite a non-normal distribution. Most people are rather poor, they do not earn the average salary. We will get back to that later.

Also take note of the higher bars, as compared to their neighbours, at ca. £100,000 and £120,000. Some of us might be tempted to try to invent a *story* about why there can be some difference in the relative probability mass, but we will refrain from it. Our data sample is quite small and it is likely that they are merely due to some natural variability.

Of course, there might be some reasons behind it (theoretically), but we cannot read this by merely looking at a single histogram – a careful further inspection is necessary, but maybe not today.

Example 4.4 *In the documentation of* ***seaborn.histplot*** *(and* ***matplotlib.pyplot.hist*** *which the former extends), we read that the* `bins` *argument is passed to* ***numpy.histogram_bin_edges****, which determines the intervals into which our data is split. Similarly,* ***numpy.histogram*** *uses the same function and additionally returns the corresponding counts (how many observations fall into each bin) instead of plotting them:*

```
counts, bins = np.histogram(income, 20)
counts
bins
## array([131, 238, 238, 147,  95,  55,  29,  23,  10,  12,   5,   7,   4,
##          3,   2,   0,   0,   0,   0,   1])
## array([  5750.   ,  15460.95,  25171.9 ,  34882.85,  44593.8 ,  54304.75,
##         64015.7 ,  73726.65,  83437.6 ,  93148.55, 102859.5 , 112570.45,
##        122281.4 , 131992.35, 141703.3 , 151414.25, 161125.2 , 170836.15,
##        180547.1 , 190258.05, 199969.   ])
```

Thus, there are 238 observations both in the [15460.95, 25171.9) and [25171.9, 34882.85) intervals. We will discuss the topic of manual data binning in greater detail in Chapter 12.

Exercise 4.5 *There is also the corresponding* `nhanes_adult_female_weight_2020`[65] *dataset in our data repository, giving weights (in kgs) of the participants. Draw a histogram. Does its shape resemble the* `income` *or* `heights` *distribution more?*

4.3.3 `marathon` (up to 3 hours) – A Left-Skewed Distribution

Next, let's consider the 37th PZU Warsaw Marathon (2015) results.

```
marathon = np.loadtxt("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/37_pzu_warsaw_marathon_mins.txt")
```

Here are the top 5 gun times (in minutes):

```
marathon[:5]  # preview first 5 (data are already sorted increasingly)
## array([129.32, 130.75, 130.97, 134.17, 134.68])
```

Plotting the histogram for the participants who finished the 42.2 km run in less than three hours reveals (see Figure 4.4) that the data are highly *left*-skewed.

[65] https://github.com/gagolews/teaching_data/blob/master/marek/nhanes_adult_female_weight_2020.txt

```
sns.histplot(marathon[marathon < 180])
plt.show()
```

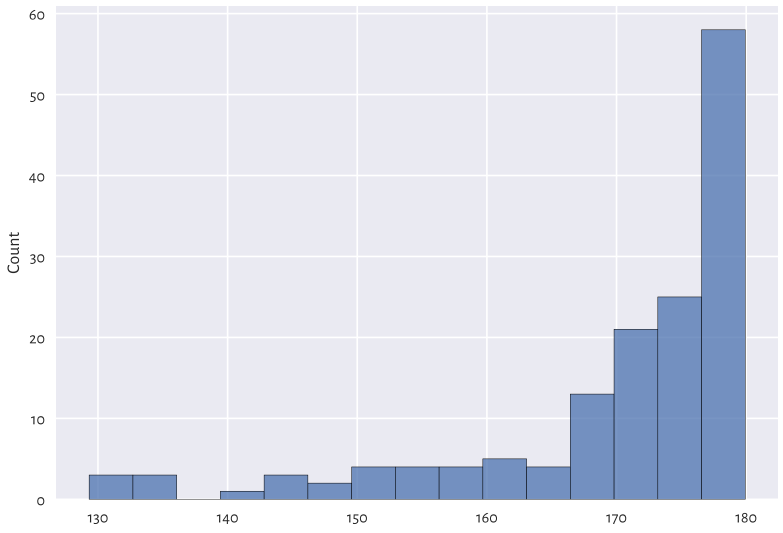


Figure4.4: Histogram for a subset of the `marathon` dataset

This is of course to be expected – there are only few elite runners in the game. Yours truly wishes his personal best becomes < 180 minutes someday. We shall see. Running is fun, so is walking; why not taking a break for an hour and going outside?

Exercise 4.6 *Plot the histogram of the untruncated (complete) version of this dataset.*

4.3.4 How Many Bins?

Unless some stronger assumptions about the data distribution are made, choosing the right number of bins is more art than science:

- too many will result in a rugged histogram,
- too few might result in our missing of important details.

Figure 4.5 illustrates this.

```
plt.subplot(121)  # 1 row, 2 columns, 1st plot
sns.histplot(income, bins=5)
plt.subplot(122)  # 1 row, 2 columns, 2nd plot
sns.histplot(income, bins=200)
plt.ylabel(None)
plt.show()
```

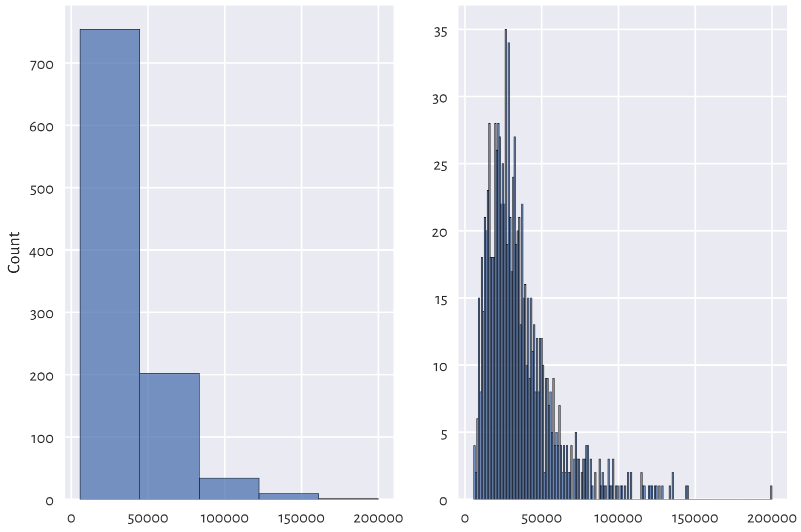


Figure4.5: Too few and too many histogram bins (the `income` dataset

For example, in the histogram with 5 bins, we miss the information that the ca. £16,000 income is more popular than the ca. £12,000 one.

The other histogram seems too fine-grained already.

Important: As usual, the "truth" is probably somewhere in-between. When preparing a histogram for publication (e.g., in a report or on a webpage), we might be tempted to think "I need to choose one and only one bin count". In fact, we do not have to – even though some people will insist on it, remember that it is we who are responsible for the data be presented in an as unambiguous fashion as possible. Thus, actually, providing 2 or 3 histograms is often a much better idea.

Further, note that someone might want to trick us by choosing the number of bins that depict the reality in a good light, when the truth is quite the opposite. For instance, the histogram on the left above hides the poorest households inside the first bar – the first income bracket is very wide. If we cannot request access to the original data, best thing we can do is to simply ignore such a data visualisation instance as non-informative and tell others not to trust it. A real scientist is a sceptic, first and foremost.

Exercise 4.7 *Studying the manual of the* **`seaborn.histplot`** *function, we see that we can provide the function with custom bin breaks. Plot a histogram which has breaks at 0, 10,000, 20,000, etc.*

Note: (*) There are quite a few heuristics to determine the number of bins automagically, see **`numpy.histogram_bin_edges`** for a few formulae. Check out how different values of the `bins` argument (e.g., `"sturges"`, `"fd"`) affect the histogram shapes on both `income` and `heights` datasets. Each has its own limitations, but some might be a good starting point for further fine-tuning.

4.3.5 Manual Binning (*)

`numpy.histogram` can be used to apply the binning of a numeric vector manually and then count the number of items falling into each bin. This can for instance be useful for data reporting – a table of ranges and corresponding counts is more informative and takes less space. Also, it can increase privacy (making subjects less identifiable) or hide some uncomfortable facts ("there are 10 people in our company earning more than £200,000 p.a." – this can be as much as £10,000,000, but shush).

We will discuss this topic in more detail in Section 12.1.5.

Exercise 4.8 *(*) Use* **`numpy.histogram`** *on the* `marathon` *data.*

4.3.6 `peds` – A Bimodal Distribution (Already Binned)

Sometimes data we get access to have already been binned by somebody else. For instance, here are the December 2021 hourly averages pedestrian counts[66] near the Southern Cross Station in Melbourne:

[66] http://www.pedestrian.melbourne.vic.gov.au/

```
peds = np.loadtxt("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/southern_cross_station_peds_2019_dec.txt")
peds
## array([  31.22580645,   18.38709677,   11.77419355,    8.48387097,
##           8.58064516,   58.70967742,  332.93548387, 1121.96774194,
##        2061.87096774, 1253.41935484,  531.64516129,  502.35483871,
##         899.06451613,  775.        ,  614.87096774,  825.06451613,
##        1542.74193548, 1870.48387097,  884.38709677,  345.83870968,
##         203.48387097,  150.4516129 ,  135.67741935,   94.03225806])
```

We cannot thus use **`seaborn.histplot`** to depict them. Instead, we can rely on a more low-level function, **`matplotlib.pyplot.bar`**, see Figure 4.6.

```
plt.bar(np.arange(0, 24), width=1, height=peds,
    edgecolor="black", alpha=0.8)
plt.show()
```

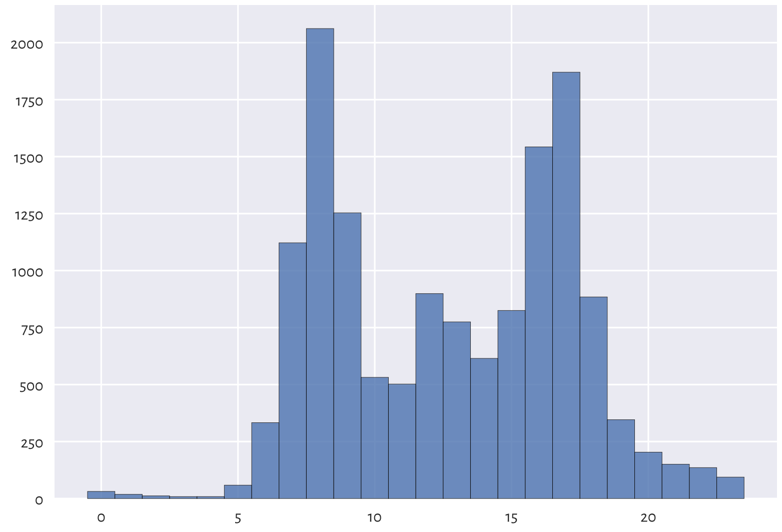


Figure 4.6: Histogram for the `peds` dataset

This is an example of a bimodal (or even trimodal) distribution: there is a morning peak and an evening peak (and some analysts probably would distinguish a lunch-time one too).

4.3.7 `matura` – A Bell-Shaped Distribution (Almost)

Figure 4.7 depicts a histogram for another interesting dataset which has already been pre-summarised.

```
matura = np.loadtxt("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/matura_2019_polish.txt")
plt.bar(np.arange(0, 71), width=1, height=matura,
    edgecolor="black", alpha=0.8)
plt.show()
```

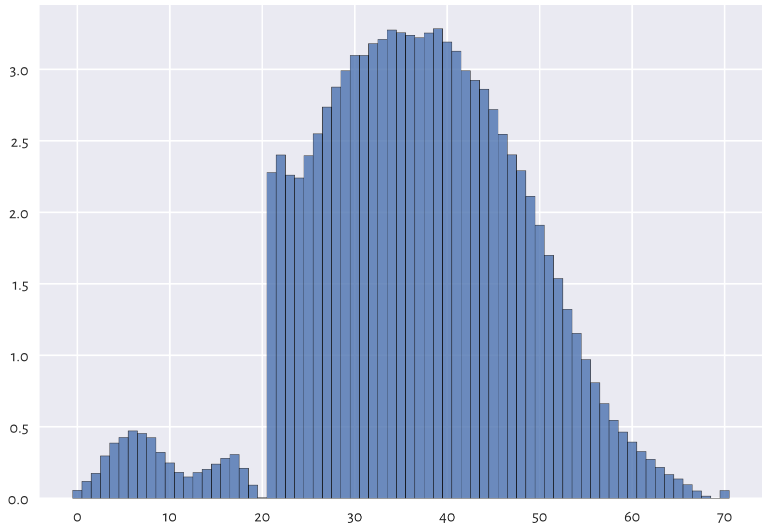


Figure4.7: Histogram for `matura` dataset

This gives the distribution[67] of the 2019 Matura[68] exam scores in Poland (in %) – Polish literature at basic level.

It seems that the distribution should be bell-shaped, but someone tinkered with it. However, knowing that:

- the examiners are good people – we teachers love our students,
- 20 points were required to pass,
- 50 points were for an essay – and beauty is in the eye of beholder,

this actually starts to make sense. Without graphically depicting this dataset, we wouldn't know that such (albeit lucky for some students) *anomalies* occurred.

4.4 Cumulative Counts and the Empirical Cumulative Distribution Function (*)

Let's get back to the `heights` dataset. The histogram above told us that, e.g., ca. 30% of women are approximately 160±5 cm tall.

However, sometimes we might be more interested in *cumulative* counts, see Figure 4.8.

```
sns.histplot(heights, stat="percent", cumulative=True)
plt.show()
```

This has a different interpretation: we can read that, e.g, 60% of all women are *no more than* ca. 162 cm tall (or that 40% are taller).

Very similar is the plot of the *empirical cumulative distribution function* (ECDF), which for a sample $\boldsymbol{x} = (x_1, \dots, x_n)$ we denote as $\hat{F}_n$.

```
n = len(heights)
heights_sorted = np.sort(heights)
plt.plot(heights_sorted, np.arange(1, n+1)/n, drawstyle="steps-post")
plt.xlabel("$t$")
plt.ylabel("$\\hat{F}_n(t)$, i.e., Prob(height $\\leq$ t)")
plt.show()
```

[67] https://cke.gov.pl/images/_EGZAMIN_MATURALNY_OD_2015/Informacje_o_wynikach/2019/sprawozdanie/Sprawozdanie%202019%20-%20J%C4%99zyk%20polski.pdf

[68] https://en.wikipedia.org/wiki/Matura

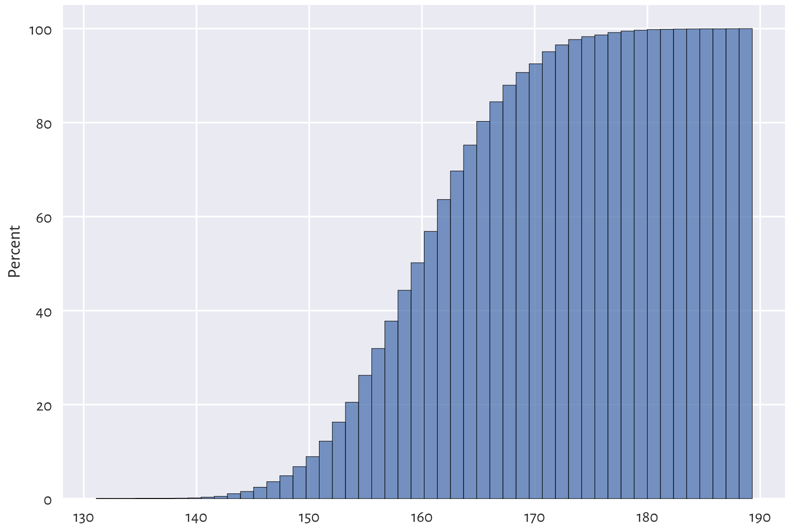


Figure4.8: Cumulative histogram for the `heights` dataset

At any given point $t \in \mathbb{R}$, $\hat{F}_n(t)$ is a step function (we cannot see the steps in Figure 4.9, because the points are too plentiful) that gives the *proportion of observations in our sample that are not greater than* t:

$$\hat{F}_n(t) = \frac{|i : x_i \leq t|}{n}$$

(read $|i : x_i \leq t|$ as the number of is such that the corresponding x_i is greater than or equal to t)

Given ordered inputs, $x_{(1)} \leq x_{(2)} \leq \dots \leq x_{(n)}$, we have:

$$\hat{F}_n(t) = \begin{cases} 0 & \text{for } t < x_{(1)}, \\ k/n & \text{for } x_{(k)} \leq t < x_{(k+1)}, \\ 1 & \text{for } t \geq x_{(n)}. \end{cases}$$

Note that drawing the ECDF does not involve binning – we only need to arrange the observations in an ascending order. Then, assuming that all observations are unique (there are no ties), the arithmetic progression $1/n, 2/n, \dots n/n$ is plotted against them.

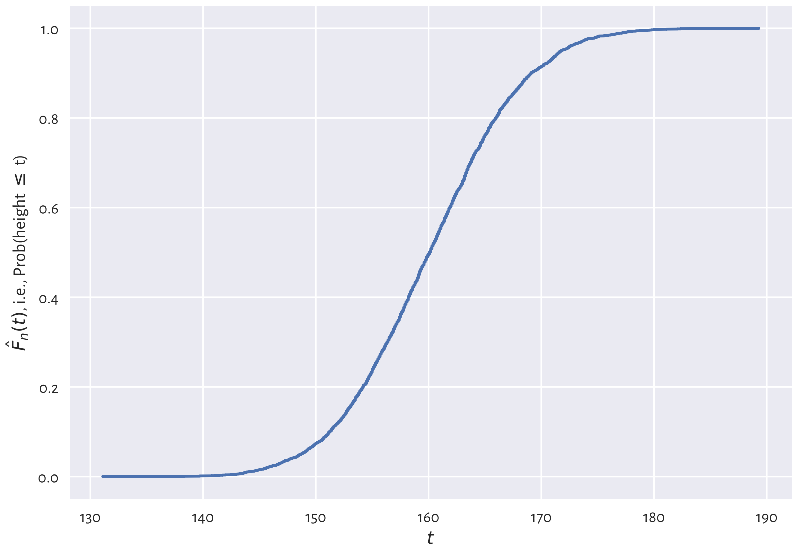


Figure4.9: Empirical cumulative distribution function for the `heights` dataset

Note: (*) Quantiles that we introduce in Section 5.1.3 can be considered a generalised inverse of the ECDF.

4.5 Log-scale and Heavy-Tailed Distributions

Consider the dataset[69] on the populations of cities in the 2000 US Census:

```
cities = np.loadtxt("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/other/us_cities_2000.txt")
```

Let's restrict ourselves only to the cities whose population is no less than 10,000. Even

[69] https://arxiv.org/abs/0706.1062v2

though they constitute ca. 14% of all the US settlements, about 84% of all the citizens live there.

```
large_cities = cities[cities >= 10000]
```

Here are the populations of the 5 largest cities:

```
large_cities[-5:]  # preview last 5 - data are sorted increasingly
## array([1517550., 1953633., 2896047., 3694742., 8008654.])
```

The histogram is depicted in Figure 4.10.

```
sns.histplot(large_cities, bins=20)
plt.show()
```

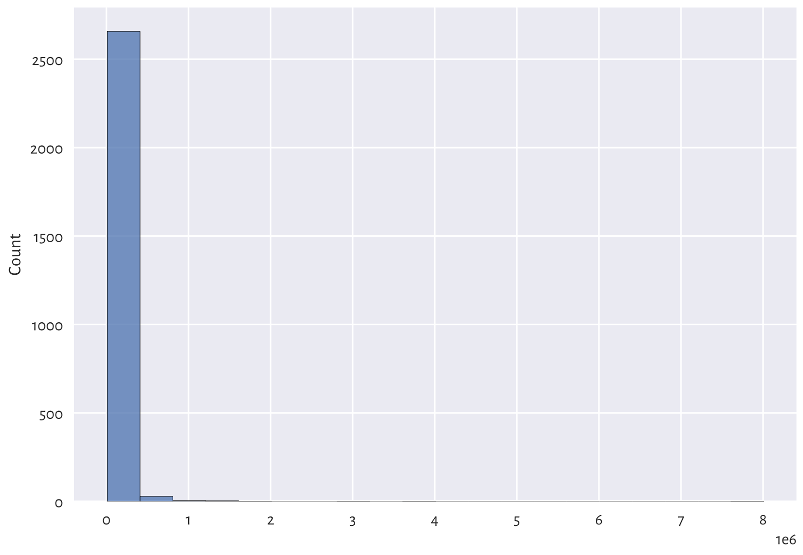


Figure4.10: Histogram for the `large_cities` dataset

is basically unreadable because the distribution is not just right-skewed; it's extremely *heavy-tailed*: most cities are small, and those that are large – such as New York – are *really* unique. Had we plotted the whole dataset (`cities` instead of `large_cities`), the results' intelligibility would be even worse.

This is why we should rather draw such a distribution on the *logarithmic* scale, see Figure 4.11.

```
sns.histplot(large_cities, bins=20, log_scale=True)
plt.show()
```

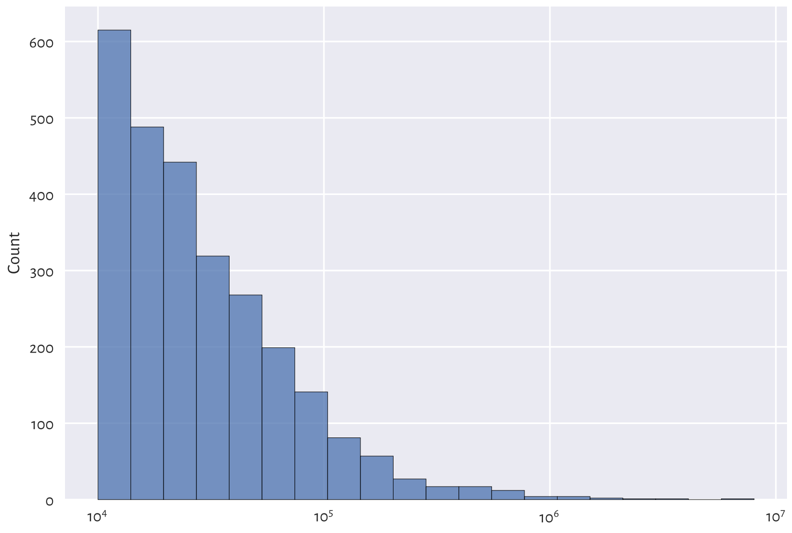


Figure4.11: Histogram for the `large_cities` dataset (logarithmic scale)

The log-scale on the x axis does not increase linearly – based on steps of equal sizes 0, 1,000,000, 2,000,000, …, and so forth. Now the increases are geometrical: 10,000, 100,000, 1,000,000.

This is a right-skewed distribution even on the logarithmic scale. Many real-world datasets have a similar behaviour, for instance, the frequencies of occurrences of words in books. In Section 12.1.5 we will discuss the Pareto distribution family which yields similar histograms.

Exercise 4.9 *Draw the histogram of `income` on the logarithmic scale. Does is resemble a bell-shaped distribution?*

Exercise 4.10 *(*) Use **`numpy.geomspace`** and **`numpy.histogram`** to apply logarithmic binning of the `large_cities` dataset manually, i.e., to create bins of equal lengths on the log-scale.*

4.6 Exercises

Exercise 4.11 *What are some benefits of using a* **`numpy`** *vector over an ordinary Python list? What are the drawbacks?*

Exercise 4.12 *What is the difference between* **`numpy.arange`** *and* **`numpy.linspace`**?

Exercise 4.13 *(*) What happens when we convert a logical vector to a numeric one? (We will discuss that later but you might want to check it yourself now.)*

Exercise 4.14 *(*) What happens when we convert a numeric vector to a logical one?*

Exercise 4.15 *What is a bell-shaped distribution?*

Exercise 4.16 *What is a right-skewed distribution?*

Exercise 4.17 *What is a heavy-tailed distribution?*

Exercise 4.18 *What is a multi-modal distribution?*

Exercise 4.19 *(*) When does logarithmic binning make sense?*

5 Aggregating Numeric Data

Histograms are based on binned data and hence provide us with snapshots of how much probability mass is allocated in different parts of the data domain.

For instance, here is a textual summary of the UK income data that we have studied in the previous part (compare also Figure 4.3):

```
income = np.loadtxt("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/uk_income_simulated_2020.txt")
b = [0, 10000, 20000, 30000, 40000, 50000, 60000, 80000, np.inf]  # bounds
c = np.histogram(income, bins=b)[0]  # counts
for i in range(len(c)):
    print(f"{b[i]:5}-{b[i+1]:5}: {c[i]:4}")
##     0-10000:   29
## 10000-20000:  207
## 20000-30000:  263
## 30000-40000:  196
## 40000-50000:  117
## 50000-60000:   74
## 60000-80000:   64
## 80000-  inf:   50
```

Instead of dealing with a large dataset, we obtained just a few counts. Below we will also be using the `heights` dataset as an illustration (recall Figure 4.2), so let's load it now.

```
heights = np.loadtxt("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/nhanes_adult_female_height_2020.txt")
```

Generally, the process of binning and its textual or visual depictions are useful in determining whether the distribution is unimodal or multimodal, left or right skewed or maybe symmetric around some point, what range of values contains most of the observations, how small or large are extreme values, etc.

Still, too much information may sometimes be overwhelming. Also, revealing it might not be a good idea for privacy or confidentially reasons (although we strongly advocate for all information of concern to the general public be openly available!).

Thus, oftentimes we will be interested in even more synthetic descriptions – data aggregates which reduce the whole dataset into a *single* number reflecting one of its many characteristics and thus providing a kind of bird's eye view on some aspect of it. We refer to such a process as data *aggregation* [[Gag15a], [GMMP09]].

In this part we discuss a few noteworthy *measures* of:

- *location*; e.g., central tendency measures such as mean and median;
- *dispersion*; e.g., standard deviation and interquartile range;
- distribution *shape*; e.g., skewness.

We also introduce *box and whisker plots*.

5.1 Measures of Location

5.1.1 Arithmetic Mean and Median

Two main measures of *central tendency* are:

- *the arithmetic mean* (sometimes for simplicity called the mean or average), defined as the sum of all observations divided by the sample size:

$$\bar{x} = \frac{(x_1 + x_2 + \cdots + x_n)}{n} = \frac{1}{n}\sum_{i=1}^{n} x_i,$$

- *the median*, being the middle value in a sorted version of the sample if its length is odd or the arithmetic mean of the two middle values otherwise:

$$m = \begin{cases} x_{(n+1)/2} & \text{if } n \text{ is odd,} \\ \frac{x_{(n/2)} + x_{(n/2+1)}}{2} & \text{if } n \text{ is even.} \end{cases}$$

They can be computed using the **numpy.mean** and **numpy.median** functions.

```
np.mean(heights), np.median(heights)
## (160.13679222932953, 160.1)
np.mean(income), np.median(income)
## (35779.994, 30042.0)
```

We note what follows:

- for symmetric distributions, the arithmetic mean and the median are expected to be more or less equal,
- for skewed distributions, the arithmetic mean will be biased towards the heavier tail.

Exercise 5.1 *Compute the arithmetic mean and median for the `37_pzu_warsaw_marathon_mins` dataset mentioned in Chapter 4.*

Exercise 5.2 *(*) Write a function that computes the median without the use of* **`numpy.median`** *(based on its mathematical definition and* **`numpy.sort`***).*

Note: (*) On a technical note, the arithmetic mean can also be computed using the **`mean`** *method* in the `numpy.ndarray` class – it will sometimes be the case that we have many different ways to perform the same operation. Or, we can "implement" it manually using the `sum` function. Thus, the following expressions are equivalent:

```
np.mean(heights), heights.mean(), \
    np.sum(heights)/len(heights), heights.sum()/heights.shape[0]
## (160.13679222932953, 160.13679222932953, 160.13679222932953, 160.
↪13679222932953)
```

On the other hand, there exists the **`numpy.median`** function but, unfortunately, the **`median`** method for vectors is not available.

5.1.2 Sensitive to Outliers vs Robust

The arithmetic mean is strongly influenced by very large or very small observations (which in some context we refer to as *outliers*). For instance, assume that we are adding one billionaire to the `income` dataset:

```
income2 = np.append(income, [1_000_000_000])
np.mean(income2)
## 1034745.2487512487
```

We feel we are all richer now, right? In fact, the arithmetic mean reflects the income each of us would get if the all the wealth was gathered in a single Santa Claus (or Robin Hood) sack and then distributed equally amongst all of us. But we do not live in a utopia, so let us move on.

On the other hand, the median is the value such that 50% of the observations are less than or equal to it and 50% of the remaining ones are no less than it.

Hence, it completely ignores most of the data points – on both the left and the right side of the distribution:

```
np.median(income2)
## 30076.0
```

Because of this, we cannot say that one measure is better than the other. Certainly for symmetrical distributions with no outliers (e.g., `heights`), the mean will be better as it uses *all* data (and its efficiency can be proven for certain statistical models). For skewed distributions (e.g., `income`), the median has a nice interpretation: what is the "middle" value? Let us still remember that these are data summaries – they only allow us to look at a single data aspect, and there can be many different, valid perspectives. The reality is complex.

We will explore the concept of robust aggregation measures in more detail in Section 17.3.2.

5.1.3 Sample Quantiles

Quantiles generalise the notion of the sample median and the inverse of the empirical cumulative distribution function (Section 4.4), giving the value that is not exceeded by the elements in a given sample with a predefined probability.

Before proceeding with a formal definition, which is quite technical, let's note that for larger sample sizes, we have the following rule of thumb.

Important: For any p between 0 and 1, the p-quantile, denoted with q_p, is a value dividing the sample in such a way that approximately *100p*% of observations are not greater than q_p, and the remaining ca. *100(1-p)*% are not less than q_p.

Quantiles appear under many different names, but they all refer to the same concept. In particular, the we can speak about *100p*-th *percentiles*, e.g., the 0.5-quantile is the same as the 50th percentile.

Furthermore:

- 0-quantile (q_0) – the minimum (also: **`numpy.min`**),
- 0.25-quantile ($q_{0.25}$) – the 1st quartile (denoted Q_1),

- 0.5-quantile ($q_{0.5}$) – the 2nd quartile a.k.a. median (denoted m or Q_2),
- 0.75-quantile ($q_{0.75}$) – the 3rd quartile (denoted Q_3),
- 1.0-quantile (q_1) – the maximum (also: **numpy.max**).

Here are the above five aggregates for the two datasets:

```
np.quantile(income, [0, 0.25, 0.5, 0.75, 1])
## array([  5750.  ,  20669.75,  30042.  ,  44123.75, 199969.  ])
np.quantile(heights, [0, 0.25, 0.5, 0.75, 1])
## array([131.1, 155.3, 160.1, 164.8, 189.3])
```

Exercise 5.3 *What is the income bracket for 95% of the "most typical" taxpayers? In other words, determine the 2.5- and 97.5-percentiles.*

Exercise 5.4 *Compute the* midrange *of* `income` *and* `heights`*, being the arithmetic mean of the minimum and the maximum. Note that this measure is extremely sensitive to outliers.*

Note: (*) As we do not like the *approximately* part in the "asymptotic definition" above, in this course we shall assume that for any $p \in [0, 1]$, the p-quantile is given by

$$q_p = x_{(\lfloor k \rfloor)} + (k - \lfloor k \rfloor)(x_{(\lfloor k \rfloor + 1)} - x_{(\lfloor k \rfloor)}),$$

where $k = (n-1)p + 1$ and $\lfloor k \rfloor$ is the floor function, i.e., the greatest integer less than or equal to k (e.g., $\lfloor 2.0 \rfloor = \lfloor 2.1 \rfloor = \lfloor 2.999 \rfloor = 2$ and $\lfloor 3.0 \rfloor = 3$).

q_p is thus a function that linearly interpolates between the points featuring the consecutive order statistics, $((k-1)/(n-1), x_{(k)})$ for $k = 1, \dots, n$. For instance, for $n = 5$, we are linearly interpolating between the points $(0, x_{(1)})$, $(0.25, x_{(2)})$, $(0.5, x_{(3)})$, $(0.75, x_{(4)})$, $(1, x_{(5)})$, whereas for $n = 6$, we do the same for $(0, x_{(1)})$, $(0.2, x_{(2)})$, $(0.4, x_{(3)})$, $(0.6, x_{(4)})$, $(0.8, x_{(5)})$, $(1, x_{(6)})$, see Figure 5.1 for an illustration.

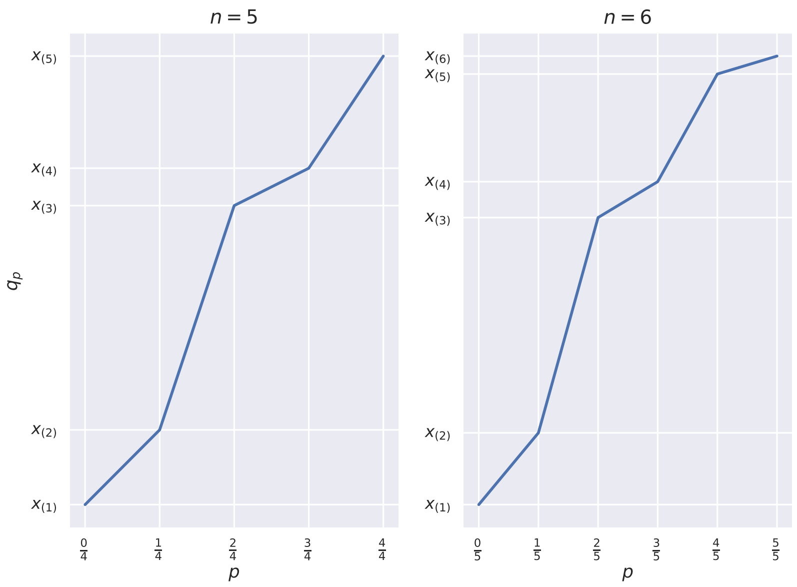


Figure5.1: q_p as a function of p for two example vectors of length 5 (left subfigure) and 6 (right)

Note that for p=0.5 we get the median regardless of whether n is even or not.

Note: (**) There are many definitions of quantiles across statistical software, see the `method` argument to **`numpy.quantile`**. They have been nicely summarised in [[HF96]] as well as in the corresponding Wikipedia[71] article. They are all approximately equivalent for large sample sizes (i.e., asymptotically), but the best practice is to be explicit about which variant we are using in the computations when reporting data analysis results. And thus, in our case, we say that we are relying on the type-7 quantiles as described in [[HF96]], see also [[Gum39]].

Actually, simply mentioning that our computations are done with **`numpy`** version 1.xx implicitly implies that the default method parameters are used everywhere, unless otherwise stated.

---

[71] https://en.wikipedia.org/wiki/Quantile

5.2 Measures of Dispersion

Measures of central tendency quantify the location of the most *typical* value (whatever that means, and we already know it's complicated). That of dispersion (spread), on the other hand, will tell us how much *variability* is in our data.

After all, when we say that the height of a group of people is 160 cm (on average) ± 14 cm (here, 2 standard deviations), the latter piece of information is a valuable addition (and is much different to the imaginary ± 4 cm case).

Some degree of variability might be good in certain contexts and bad in other ones. A bolt factory should keep the variance of the fasteners' diameters as low as possible – after all, this is how we define quality products (assuming that on average they all meet the required specification). On the other hand, too much diversity in human behaviour, where everyone feels that they are special, is not really sustainable (but lack thereof would be extremely boring), and so forth.

5.2.1 Standard Deviation (and Variance)

The standard deviation (** in the so-called uncorrected for bias version), is the average distance to the arithmetic mean:

$$s = \sqrt{\frac{(x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + \cdots + (x_n - \bar{x})^2}{n}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2},$$

Computing the above with **numpy**:

```
np.std(income), np.std(heights)
## (22888.77122437908, 7.062021850008261)
```

The standard deviation measures the average degree of spread around the arithmetic mean. Thus, it makes most sense for data distributions that are symmetric around the mean. This measure is useful overall for making comparisons across different samples. However, without further assumptions, it's quite difficult to express the meaning of a particular value of s (e.g., the statement that the standard deviation of income is £22,900 is hard to interpret).

Note: (*) For bell-shaped data (like `heights`, more precisely, for normally-distributed samples, see the next chapter), we sometimes report $\bar{x} \pm 2s$, as the theoretical expectancy is that ca. 95% of data points fall into the $[\bar{x} - 2s, \bar{x} + 2s]$ interval (the so-called 2-sigma rule).

Note: (*) Passing `ddof=1` (*delta degrees of freedom*) to **`numpy.std`** will apply division by *n-1* instead of *n* in the denominator. This estimator has slightly better statistical properties (which we normally explore in a course in mathematical statistics, which this one is not); interestingly, the `std` methods in the **`pandas`** package has `ddof=1` by default, therefore we might be interested in setting `ddof=0` therein.

Further, the *variance* is the square of the standard deviation, s^2 Note that if data are expressed in centimetres, then the variance is in centimetres *squared*, which is not very intuitive. The standard deviation does not have this drawback. Mathematicians find the square root annoying though (for many reasons); that is we will come across s^2 every now and then below too.

5.2.2 Interquartile Range

The interquartile range (IQR) is another popular measure of dispersion. It is defined as the difference between the 3rd and the 1st quartile:

$$\mathrm{IQR} = q_{0.75} - q_{0.25} = Q_3 - Q_1.$$

Computing the above is almost effortless:

```
np.quantile(income, 0.75) - np.quantile(income, 0.25)
## 23454.0
np.quantile(heights, 0.75) - np.quantile(heights, 0.25)
## 9.5
```

The IQR has an appealing interpretation, because we may say that this is the range comprised of the 50% *most typical* values. It is a quite robust measure, as it ignores the 25% smallest and 25% largest observations. Standard deviation, on the other hand, is extremely sensitive to outliers.

Furthermore, by *range* (or support) we will mean a measure based on extremal quantiles: it is the difference between the maximal and minimal observation.

5.3 Measures of Shape (*)

Note that from a histogram we can easily deduce if a dataset is symmetric or skewed. It turns out that we can also give a numerical summary of such a feature. Namely, the *skewness* is given by:

$$g = \frac{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^3}{\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}\right)^3}.$$

For symmetric distributions, skewness is approximately zero. Positive and negative skewness indicates a heavier right and left tail, respectively.

For example:

```
scipy.stats.skew(heights)
## 0.0811184528074054
```

is thus an instance of almost-symmetric distribution.

```
scipy.stats.skew(income)
## 1.9768735693998942
```

Income is right-skewed, and now we have this expressed as a single number.

5.4 Box (and Whisker) Plots

The box and whisker plot (or the box plot for short) depicts some of the most noteworthy features of a data sample and is based only on the aggregates

```
plt.subplot(211)  # 2 rows, 1 column, 1st subplot
sns.boxplot(data=income, orient="h")
plt.title("income")

plt.subplot(212)  # 2 rows, 1 column, 2nd subplot
sns.boxplot(data=heights, orient="h")
plt.title("heights")
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

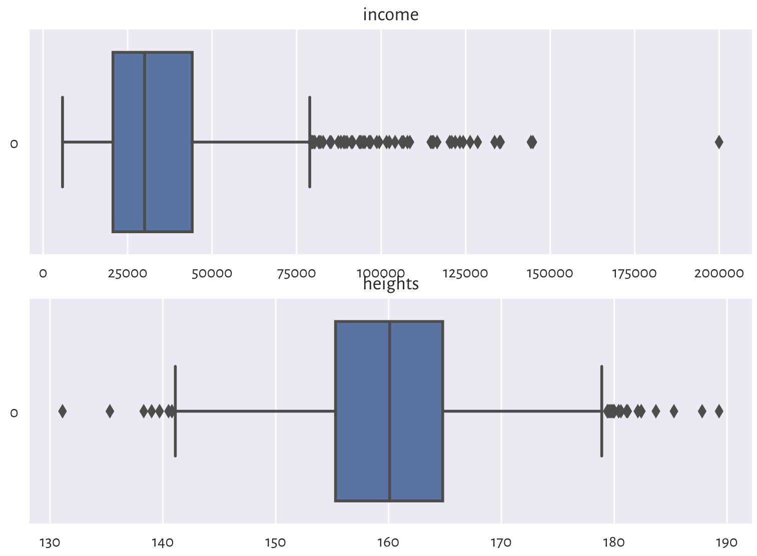


Figure5.2: Example box-and-whisker plots

Each box plot (compare Figure 5.2) consists of:

- the box, which spans between the 1st and the 3rd quartile:
    - the median is clearly marked by a vertical bar inside the box;
    - note that the width of the box corresponds to the IQR;
- the whiskers, which span between:
    - the smallest observation (the minimum) or $Q_1 - 1.5\mathrm{IQR}$ (the left side of the box minus 3/2 of its width), whichever is larger, and
    - the largest observation (the maximum) or $Q_3 + 1.5\mathrm{IQR}$ (the right side of the box plus 3/2 of its width), whichever is smaller.

Additionally, all observations that are less than $Q_1 - 1.5\mathrm{IQR}$ (if any) or greater than $Q_3 + 1.5\mathrm{IQR}$ (if any) are separately marked.

Note: We are used to referring to the marked points as *outliers*, however, it doesn't automatically mean there is anything *anomalous* about them. They are *atypical* in the sense that they are considerably farther away from the box. However, they might also indicate some problems in data quality (e.g., when one made a typo entering the data). Actually, box plots are calibrated (via the nicely round magic constant 1.5) in such a way that we expect there to be no or only few outliers if the data are normally distributed. For skewed distributions, there will naturally be many outliers on either side.

Important: Most of the statistical packages *do not* include the arithmetic mean in the figure. If they do, it is marked with a special symbol.

Important: Box plots will be particularly useful for comparing data samples with each other (e.g., body measures of men and women separately); both in terms of the relative shift (location) as well as spread and skewness, see Figure 13.1.

Exercise 5.5 *Call `matplotlib.pyplot.plot(numpy.mean(..data..), 0, "wX")` after `seaborn.boxplot` to mark the arithmetic mean with a white cross.*

Note: We may also sometimes be interested in a *violin plot*, which combines the box plot (although with no outliers marked) and the so-called kernel density estimator (which is a smoothened version of a histogram), compare Figure 5.3.

```
sns.violinplot(data=income, orient="h")
plt.show()
```

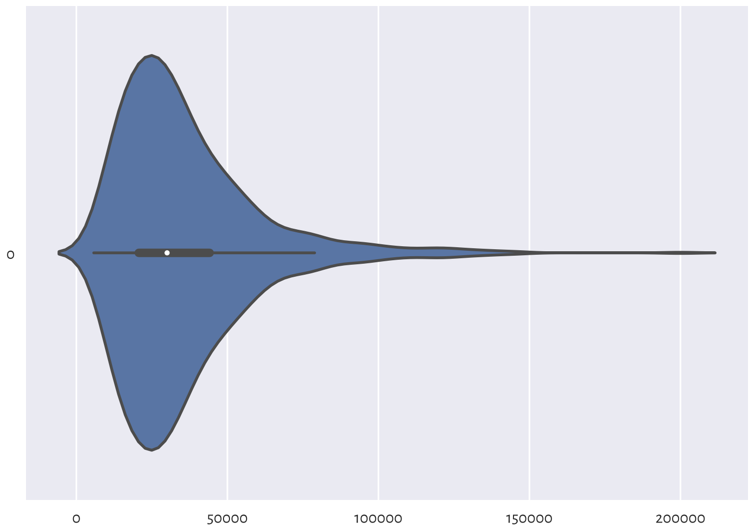


Figure 5.3: An example violin plot

5.5 Further Reading

We have said that the arithmetic mean is very sensitive to extreme observations. In Section 17.3.2, we will consider the *trimmed* and *winsorised* means which are more *robust* in presence of outliers – they remove or replace a few smallest and largest observations in a sample.

Similarly, the *mean absolute deviation from the mean* or from the median or even the *median absolute deviation from the median* can be used as more robust versions of dispersion measures.

The arithmetic mean is not the only mean of interest. The two other very famous means are the *harmonic* and *geometric* ones. The former can be used for computing the average speed from speed measurements at sections of identical lengths and the latter is more meaningful for averaging growth rates and speedups.

The *coefficient of variation*, being the standard deviation dived by the arithmetic mean, is an example of a *relative* (or normalised) spread measure. It can be useful for comparing data on different scales, as it is unit-less (think how standard deviation changes when you convert between metres and kilometres).

The *Gini index* widely used in economics, can serve as a measure of dispersion (or shape – depending how we look at it), but assumes that all data points are nonnegative. It is normalised so that it takes values in the unit interval. An index of 0 reflects the situation where all values in a sample are the same (0 variance; perfect equality). If there is a single entity in possession of all the "wealth", and the remaining ones are 0, then the index is equal to 1.

Entropy can be used as a measure of spread for data which sum to 1, e.g., probabilities.

Kurtosis (or excess kurtosis) is another measure of shape, describing whether a distribution is heavy- or thin-tailed.

For some more generic treatment of aggregation functions see [[GMMP09]] and [[Gag15b]].

5.6 Exercises

Exercise 5.6 *How can we interpret the different values of the arithmetic mean, median, standard deviation, interquartile range, and skewness?*

Exercise 5.7 *There is something scientific and magical about* numbers *that makes us approach them with some kind of respect. However, taking into account that there are many possible data aggregates, there is a risk that a party may be cherry-picking – reporting the one that portrays the analysed entity in a good or bad light. For instance, reporting the mean instead of the median or vice versa. Is there anything that can be done about it?*

Exercise 5.8 *Although – mathematically speaking – all measures can be computed on all data, it does not mean that it always makes sense. For instance, some distributions will yield skewness of 0, but we should not automatically assume that they are nicely symmetric and bell-shaped (e.g., this can be a bimodal distribution). It is thus best to always visualise your data. Give some examples of datasets and measures where we should be critical about the obtained results.*

Exercise 5.9 *Give some examples, where simple data preprocessing can drastically change the values of chosen sample aggregates.*

Exercise 5.10 *(**) Reflect on the famous*[72] *saying* not everything that can be counted counts, and not everything that counts can be counted.

Exercise 5.11 *(**) Being a data scientist can be a frustrating job, especially when you care for some causes. Reflect on:* some things that count can be counted, but we will not count them, because there's no budget for it.

Exercise 5.12 *(**) Being a data scientist can be a frustrating job, especially when you care for the truth. Reflect on:* some things that count can be counted, but we will not count them, because some people might be offended.

Exercise 5.13 *(**) Assume you were to establish your own nation on some island and become the benevolent dictator thereof. How would you* measure *if your people are happy or not? Let's say that you need to come up with 3 qualitative measures (key performance indicators). What would happen if your policy making were solely be focused on optimising those KPIs? How about the same problem but with regards to your company and employees?*

[72] https://quoteinvestigator.com/2010/05/26/everything-counts-einstein/

6

Transforming and Filtering Numeric Data

It is extremely rare for our datasets to bring interesting and valid insights out of the box. The ones we are using for illustrational purposes in the first part of our series have already been curated. After all, this is an introductory course, and we need to build up the necessary skills and not overwhelm the kind reader with too much information at the same time. We learn simple things first, learn them well, so that then we can move to more complex matters with a healthy level of confidence.

In real life, various *data cleansing* and *feature engineering* techniques will need to be performed on data. Generally, most of them can be reduced to simple operations on vectors:

- summarising data (for example, computing the median or sum),
- transforming values (applying mathematical operations on each element, such as subtracting a scalar or taking the natural logarithm),
- filtering (selecting or removing observations that meet specific criteria, e.g., those that are larger than the arithmetic mean ± 5 standard deviations).

The first topic has already been covered in Chapter 5, it is thus time for the two remaining ones.

Important: The same operations we are going to be applying on individual data frame columns in Chapter 11.

6.1 Vectorised Mathematical Functions

Numpy, just like any other comprehensive numerical computing package, library, or environment (e.g., R and Julia), defines many basic mathematical functions:

- absolute value: **`numpy.abs`**,

- square and square root: **numpy.square** and **numpy.sqrt**,
- exponential function: **numpy.exp**,
- logarithms: **numpy.log** (the natural logarithm, i.e., base e), **numpy.log10** (base 10), etc.,
- trigonometric functions: **numpy.sin**, **numpy.cos**, **numpy.tan**, etc.,
- rounding and truncating: **numpy.round**, **numpy.floor**, **numpy.ceil**, **numpy.trunc** (when in doubt about their definitions, we should, as usual, refer to the official manual).

Each of these functions is *vectorised*: if we apply, say, f, on a vector like $\boldsymbol{x} = (x_1, \dots, x_n)$, we obtain a sequence of the same size with all elements appropriately transformed:

$$f(\boldsymbol{x}) = (f(x_1), f(x_2), \dots, f(x_n)).$$

We will frequently be using such operations for adjusting the data, e.g., as in Figure 7.4 where we note that the *logarithm* of the UK incomes has a bell-shaped distribution.

For example:

```
np.abs([-2, -1, 0, 1, 2, 3])
## array([2, 1, 0, 1, 2, 3])
```

Note that the input list has automatically been converted to a **numpy** vector.

Other examples:

```
np.exp([-1, 0, 1, 2])
## array([0.36787944, 1.        , 2.71828183, 7.3890561 ])
np.log(np.exp([-1, 0, 1, 2]))  # the natural logarithm is the inverse of exp
## array([-1.,  0.,  1.,  2.])
```

and so forth.

Thanks to the vectorised functions, our code is not only more readable, but also it runs faster – as otherwise we would have to employ a **while** or **for** loop to traverse through each element in a given sequence.

6.2 Arithmetic Operators

We can apply the standard binary (two-argument) arithmetic operators `+`, `-`, `*`, `/`, `**`, `%`, `//` on vectors too. Below we discuss the possible cases of the operands' lengths.

6.2.1 Vector-Scalar Case

Often we will be referring to the binary operators in contexts where one operand is a vector and the other one is a single value, for example:

```
np.array([-2, -1, 0, 1, 2, 3])**2
## array([4, 1, 0, 1, 4, 9])
(np.array([-2, -1, 0, 1, 2, 3])+2)/5
## array([0. , 0.2, 0.4, 0.6, 0.8, 1. ])
```

In such a case, each element in the vector is being operated upon (e.g., squared, divided by 5) and we get a vector of the same length in return. Hence, in this case the operators behave just like the vectorised mathematical functions discussed above.

Mathematically, it is common to assume that scalar multiplication and, less commonly, addition is performed in this way.

$$c\boldsymbol{x} + t = (cx_1 + t, cx_2 + t, \dots, cx_n + t).$$

We will also become used to writing, $(\boldsymbol{x} - t)/c$ which is of course equivalent to $(1/c)\boldsymbol{x} + (-t/c)$.

6.2.2 Application: Feature Scaling

Such vector-scalar operations and aggregation functions are the basis for the most commonly applied *feature scalers*, which can increase the interpretability of data points:

- standardisation,
- normalisation,
- min-max scaling and clipping,

bringing them onto a common, unitless scale. All of them are linear transforms of the form $\boldsymbol{y} = c\boldsymbol{x} + t$, which geometrically we can interpret as scaling (stretching or shrinking) and shifting (translating), see Figure 6.1 for an illustration.

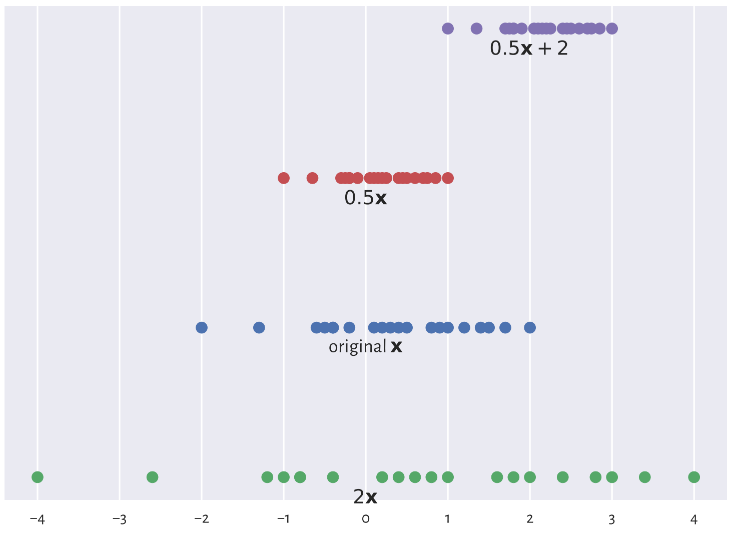


Figure6.1: Scalled and shifted versions of a given vector

Note: Let $\boldsymbol{y} = c\boldsymbol{x}+t$ and let $\bar{x}, \bar{y}, s_x, s_y$ denote the vectors' arithmetic means and standard deviations. The following properties hold.

- the arithmetic mean and all the quantiles (including, of course, the median), are equivariant with respect to translation and scaling; it holds, for instance, $\bar{y} = c\bar{x}+t$;
- the standard deviation, the interquartile range, and the range are invariant to translations and equivariant to scaling; e.g., $s_y = cs_x$

As a by-product, for variance we get of course… $s_y^2 = c^2 s_x^2$.

Standardisation and Z-scores

Consider the female heights dataset:

```
heights = np.loadtxt("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/nhanes_adult_female_height_2020.txt")
```

(continues on next page)

(continued from previous page)

```
heights[-5:] # preview
## array([157. , 167.4, 159.6, 168.5, 147.8])
```

whose mean $\bar{x}$ and standard deviation s are as follows:

```
np.mean(heights), np.std(heights)
## (160.13679222932953, 7.062021850008261)
```

Important: A *standardised* version of a vector $\boldsymbol{x} = (x_1, \dots, x_n)$ consists in subtracting, from each element, the sample arithmetic mean (which we call *centring*) and then dividing it by the standard deviation, i.e., $\boldsymbol{z} = (\boldsymbol{x} - \bar{x})/s$.

Thus, we transform each x_i so as to obtain:

$$z_i = \frac{x_i - \bar{x}}{s}.$$

With **numpy**, we simply apply 2 aggregation functions and 2 arithmetic operations:

```
heights_std = (heights-np.mean(heights))/np.std(heights)
heights_std[-5:] # preview
## array([-0.44417764,  1.02848843, -0.07601113,  1.18425119, -1.74692071])
```

Important: What we obtain in result is sometimes referred to as the *z-scores*.

Z-scores are nicely interpretable:

- z-score of 0 corresponds to an observation equal to the sample mean (perfectly average);
- z-score of 1 is obtained for a datum 1 standard deviation above the mean;
- z-score of -2 means that it is a value 2 standard deviations below the mean;

and so forth.

We should also note that because of the way they emerge, the mean of z-scores is always 0 and standard deviation is 1 (up to a tiny numerical error, as usual):

```
np.mean(heights_std), np.std(heights_std)
## (1.8920872660373198e-15, 1.0)
```

Moreover, z-scores are unitless (e.g., the original `heights` were measured in centimetres).

Important: Standardisation enables the comparison of measurements on different scales (think: height in cm vs weight in kg or apples vs oranges).

Exercise 6.1 *We have a patient whose height z-score is 1 and weight z-score is -1. How to interpret this information?*

Exercise 6.2 *How about a patient whose weight z-score is 0 but BMI z-score is 2?*

Note: (*) Standardisation makes most sense for bell-shaped distributions, in particular normally-distributed ones. Recalling the 2σ rule for the normal family, it is *expected* that 95% of observations should have z-scores between -2 and 2. Further, z-scores less than -3 and greater than 3 are highly unlikely.

Note: (*) Sometimes we might be interested in performing some form of *robust* standardisation (e.g., for skewed data or those that feature some outliers). In such a case, we can replace the mean with the median and the standard deviation with the IQR.

Min-Max Scaling and Clipping

A less frequently but still noteworthy transformation is called *min-max scaling* and involves subtracting the minimum and then dividing by the range.

```
x = np.array([-1.5, 0.5, 3.5, -1.33, 0.25, 0.8])
(x - np.min(x))/(np.max(x)-np.min(x))
## array([0.   , 0.4  , 1.   , 0.034, 0.35 , 0.46 ])
```

Here, the smallest value is mapped to 0 and the largest one is equal to 1. Note that 0.5 does not mean that the value is equal to the mean (unless we are very lucky!).

Also, *clipping* can be used to replace all values less than 0 with 0 and those greater than 1 with 1.

```
np.clip(x, 0, 1)
## array([0.  , 0.5 , 1.  , 0.  , 0.25, 0.8 ])
```

The function is of course flexible; another popular choice is clipping to *[-1, 1]*. This can also be implemented manually by means of the vectorised pairwise minimum and maximum functions.

```
np.minimum(1, np.maximum(0, x))
## array([0.  , 0.5 , 1.  , 0.  , 0.25, 0.8 ])
```

Normalisation (l_2; Dividing by Length) (*)

Normalisation is the scaling of a given vector so that it is of *unit length*. Usually, by *length* we mean the square root of the sum of squares, i.e., the Euclidean (l^2) norm:

$$\|(x_1,\dots,x_n)\| = \sqrt{x_1^2 + x_2^2 + \dots + x_n^2} = \sqrt{\sum_{i=1}^n x_i^2},$$

Its special case for *n=2* we know well from high school: the length of a vector (a, b) is $\sqrt{a^2+b^2}$, e.g., $\|(1,2)\| = \sqrt{5} \simeq 2.236$. Also, it's good to program our brains so that when next time we see $\|\boldsymbol{x}\|^2$, we immediately think of the *sum of squares*.

We can thus write:

```
x = np.array([1, 5, -4, 2, 2.5])
x/np.sqrt(np.sum(x**2))  # x divided by the Euclidean norm of x
## array([ 0.13834289,  0.69171446, -0.55337157,  0.27668579,  0.34585723])
```

to mean the normalised vector

$$\frac{\boldsymbol{x}}{\|\boldsymbol{x}\|} = \left(\frac{x_1}{\|\boldsymbol{x}\|}, \frac{x_2}{\|\boldsymbol{x}\|}, \dots, \frac{x_n}{\|\boldsymbol{x}\|}\right).$$

Note: Note that normalisation is very similar to standardisation if data are already centred (when mean was subtracted). Actually, we can obtain one from the other via scaling by $\sqrt{n}$.

Important: A common confusion is that normalisation is supposed to make data more "normally" distributed. This is not the case, as we only scale (stretch or shrink) the observations here.

Normalisation (l_1; Dividing by Sum) (*)

At other times, by *length* we can also mean the Manhattan (l^1) norm,

$$\|(x_1, \dots, x_n)\|_1 = |x_1| + |x_2| + \dots + |x_n| = \sum_{i=1}^{n} |x_i|,$$

being the sum of absolute values.

```
x / np.sum(np.abs(x))
## array([ 0.06896552,  0.34482759, -0.27586207,  0.13793103,  0.17241379])
```

l_1 normalisation is frequently applied on vectors of nonnegative values, whose normalised versions can be interpreted as *probabilities* or *proportions*: values between 0 and 1 which sum to 1 (or, equivalently, 100%). In particular, on binned data:

```
c, b = np.histogram(heights, [-np.inf, 150, 160, 170, np.inf])
print(c)  # counts
## [ 306 1776 1773  366]
```

And now, converting the counts to empirical probabilities:

```
p = c/np.sum(c)
print(p)
## [0.07249467 0.42075338 0.42004264 0.08670931]
```

We did not apply **`numpy.abs`**, because the values were already nonnegative.

6.2.3 Vector-Vector Case

The arithmetic operators can also be applied on two vectors of equal lengths. In such a case, they will act *elementwisely*: taking each element from the first operand and combining it with the *corresponding* element from the second argument:

```
np.array([2, 3, 4, 5]) * np.array([10, 100, 1000, 10000])
## array([   20,    300,  4000, 50000])
```

We see that the first element in the left operand (2) was multiplied by the first element in the right operand (10). Then, we computed 3*100 (the second elements), and so forth.

Such a behaviour of the binary operators is inspired by the usual convention in vector algebra where applying + (or −) on $\boldsymbol{x} = (x_1, \dots, x_n)$ and $\boldsymbol{y} = (y_1, \dots, y_n)$ means exactly:

$$\boldsymbol{x} + \boldsymbol{y} = (x_1 + y_1, x_2 + y_2, \dots, x_n + y_n).$$

Using other operators this way (elementwisely) is less standard in mathematics (for instance multiplication might denote the dot product), but in **numpy** it is really convenient.

Example 6.3 *Another example:*

```
p = np.array([0.1, 0.3, 0.25, 0.15, 0.12, 0.08])  # example vector
-np.sum(p*np.log(p))
## 1.6790818544987114
```

computes the value of the expression $h = -(p_1 \log p_1 + \cdots + p_n \log p_n)$, *i.e.,* $h = -\sum_{i=1}^{n} p_i \log p_i$. *Note that it involves the use of a unary vectorised minus (change sign), an aggregation function (sum), a vectorised mathematical function (log), and an elementwise multiplication.*

Example 6.4 *Let's say that for some reason we would like to plot two mathematical functions:* $f(x) = \sin x \cos^2 \frac{x}{3}$ *and* $g(x) = \frac{\sin(-2x)}{x^2+1}$ *for x in the interval* $[-2\pi, 4\pi]$.

To do this, we can probe the values of f and g at sufficiently many points using the vectorised operations discussed so far and then use the **matplotlib.pyplot.plot** *function to draw what we see in Figure 6.2.*

```
x = np.linspace(-2*np.pi, 4*np.pi, 1001)  # many points in the said interval
yf = np.sin(x)*np.cos(x/3)**2
yg = np.sin(-2*x)/(x**2+1)
plt.plot(x, yf, 'r-', label="f(x)")  # red solid line
plt.plot(x, yg, 'b:', label="g(x)")  # blue dotted line
plt.legend()
plt.show()
```

Note that decreasing the number of points in x will reveal that the plotting function in fact draws a series of straight line segments.

Exercise 6.5 *Based on the* nhanes_adult_female_height_2020[73] *and* nhanes_adult_female_weight_2020[74] *datasets (discussed in the part on histograms), using a single line of code, compute the vector of BMIs of all persons.*

[73] https://github.com/gagolews/teaching_data/blob/master/marek/nhanes_adult_female_height_2020.txt

[74] https://github.com/gagolews/teaching_data/blob/master/marek/nhanes_adult_female_weight_2020.txt

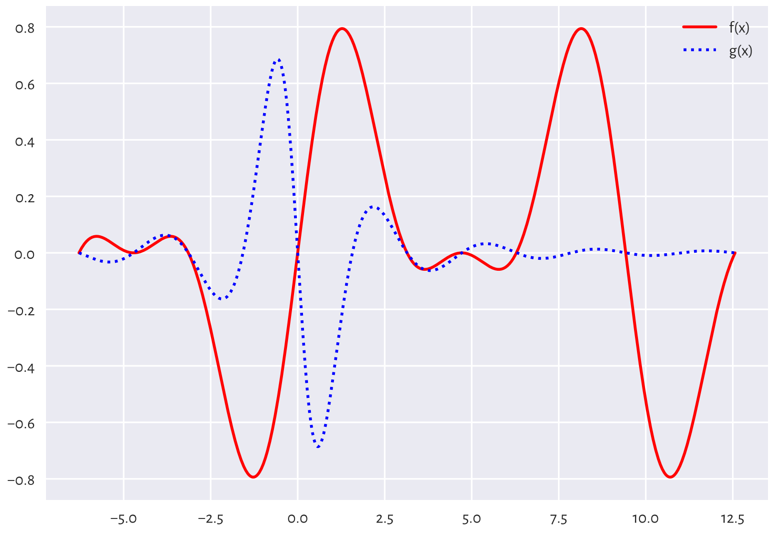


Figure6.2: With vectorised functions, it is easy to generate plots like this one

6.3 Vector Indexing

Recall that sequential objects in Python (lists, tuples, strings, ranges) support indexing using scalars and slices:

```
x = [10, 20, 30, 40, 50]
x[1]  # scalar index - extract
## 20
x[1:2]  # slice index - subset
## [20]
```

numpy vectors additionally support two other indexing schemes: using integer and boolean sequences.

6.3.1 Integer Indexing

Indexing with a single integer *extracts* a particular element:

```
x = np.array([10, 20, 30, 40, 50])
x[0], x[1], x[-1]  # first, second, last
## (10, 20, 50)
```

We can also use lists of vector of integer indexes, which return a subvector with elements at the specified indices:

```
x[ [0] ], x[ [0, 1, -1] ], x[ [] ], x[ [0, 1, 0, 0] ]
## (array([10]), array([10, 20, 50]), array([], dtype=int64), array([10, 20,
↪10, 10]))
```

Note that we added some spaces between the square brackets, because, for example, `x[[0, 1, -1]]` might look slightly more enigmatic (What are these double square brackets? Nah, it is a list inside the index operator).

6.3.2 Logical Indexing

Subsetting using a vector of the same length as the indexed vector is possible too:

```
x[ [True, False, True, True, False] ]
## array([10, 30, 40])
```

Returned the 1st, 3rd, and 4th element (select 1st, don't select 2nd, select 3rd, select 4th, don't select 5th).

This is particularly useful as a *data filtering* technique. Knowing that the relational operators `` `<` ``, `` `<=` ``, `` `==` ``, `` `!=` ``, `` `>=` ``, and `` `>` `` on vectors are also performed elementwisely (just like `` `+` ``, `` `*` ``, etc.), for instance:

```
x >= 30
## array([False, False,  True,  True,  True])
```

we can write:

```
x[ x >= 30 ]
## array([30, 40, 50])
```

to mean "select those elements in x which are not less than 30".

Of course, the indexed vector and the vector used in comparisons do not have to be the same:

```
y = (x/10) % 2
y  # 0 if a number is a multiply of 10 times an even number
## array([1., 0., 1., 0., 1.])
x[ y == 0 ]
## array([20, 40])
```

Important: If we would like to combine many logical vectors, unfortunately we cannot use the `and`, `or`, and `not` operators, because they are not vectorised (this is a limitation of our language per se).

`numpy` uses the `` `&` ``, `` `|` ``, and `` `~` `` operators, which unfortunately have lower order of precedence than `` `<` ``, `` `<=` ``, `` `==` ``, etc. Therefore, the bracketing of the comparisons is obligatory.

For example:

```
x[ (20 <= x) & (x <= 40) ]  # check what happens if we skip the brackets
## array([20, 30, 40])
```

means "elements in x between 20 and 40" (greater than or equal to 20 and less than or equal to 40).

Exercise 6.6 *Compute the BMIs only of the persons whose height is between 150 and 170 cm.*

6.3.3 Slicing

Just as with ordinary lists, slicing with ":" can be used to fetch the elements at indexes in a given range `from:to` or `from:to:by`.

```
x[::-1], x[3:], x[1:4]
## (array([50, 40, 30, 20, 10]), array([40, 50]), array([20, 30, 40]))
```

Important: (*) For efficiency reasons, slicing returns a *view* on existing data – it does not make an independent copy of the subsetted elements.

In other words, both x and its sliced version share the same memory. This is important when we apply operations which modify a given vector in place, such as the sort method.

```
y = np.array([6, 4, 8, 5, 1, 3, 2, 9, 7])
y[::2].sort()
y
## array([1, 4, 2, 5, 6, 3, 7, 9, 8])
```

This sorted every second element in y (i.e., [6, 8, 1, 2, 7] was rearranged so as to obtain [1, 2, 6, 7, 8]).

Indexing with an integer or logical vector always returns a copy.

```
y[ [1, 3, 5, 7] ].sort()
y
## array([1, 4, 2, 5, 6, 3, 7, 9, 8])
```

This *did not* modify the original vector.

6.4 Other Operations

6.4.1 Sorting and Ranking

The **numpy.sort** function returns a sorted copy of the given vector, i.e., determines the order statistics.

```
x = np.array([10, 30, 50, 40, 20, 50])
np.sort(x)
## array([10, 20, 30, 40, 50, 50])
```

Recall that x.sort() sorts the vector in-place.

Next, **scipy.stats.rankdata** returns a vector of *ranks*, which is the inverse of **numpy.argsort**.

```
scipy.stats.rankdata(x)
## array([1. , 3. , 5.5, 4. , 2. , 5.5])
```

Element 10 is the smallest (“the winner”, say, the quickest racer), hence it has rank 1. Next is element 30, which is the 3rd on the podium, hence its rank is 3.

Exercise 6.7 *Consult the manual page of* **`scipy.stats.rankdata`** *and test various methods for dealing with* tied *elements, e.g., a situation where some values are nonunique (for instance, when two racers finished in exactly the same time).*

Note: (*) There are many methods in nonparametric statistics (those that do not make any too particular assumptions about the underlying data distribution) that are based on ranks, e.g., the Spearman correlation coefficient which we cover later.

`numpy.argsort` returns a sequence of indexes that lead to an ordered version of a given vector (i.e., an ordering permutation).

```
np.argsort(x)
## array([0, 4, 1, 3, 2, 5])
```

Which means that the smallest element is at index 0, then the 2nd smallest is at index 4, 3rd smallest at index 1, etc. Therefore:

```
x[np.argsort(x)]
## array([10, 20, 30, 40, 50, 50])
```

is equivalent to **`numpy.sort`**`(x)`.

On a side note, **`numpy.random.permutation`** rearranges the elements in a given vector.

Note: (**) If there are tied observations in a vector `x`, **`numpy.argsort`**`(x, kind="stable")` will use a *stable* sorting algorithm (timsort[75], a variant of mergesort), which guarantees that the ordering permutation is unique: tied elements are placed in order of appearance

---

Note: (**) Readers with some background in discrete mathematics will be interested in the fact that calling **`numpy.argsort`** on a vector representing a permutation of elements in fact generates its inverse. In particular, **`np.argsort`**`(`**`np.argsort`**`(x, kind="stable"))+1` is equivalent to **`scipy.stats.rankdata`**`(x, method="ordinal")`.

---

[75] https://github.com/python/cpython/blob/3.7/Objects/listsort.txt

6.4.2 Searching for Certain Indexes

`numpy.argmin` and **`numpy.argmax`** return the index at which we can find the smallest and the largest observation in a given vector.

```
x = np.array([10, 30, 50, 40, 20, 50])
np.argmin(x), np.argmax(x)
## (0, 2)
```

Note that if there are tied observations, the smallest index is returned.

We can use **`numpy.where`** to get the indexes where a logical vector has elements equal to `True`. For example:

```
np.where(x == np.max(x))
## (array([2, 5]),)
```

This is a version of **`numpy.argmax`** that lets us decide what we would like to do with the tied maxima.

Exercise 6.8 *Let x be a vector with possible ties. Write an expression that returns a randomly chosen index pinpointing the sample maximum.*

Exercise 6.9 ***`numpy.where`*** *can also act as a vectorised version of the `if` statement. Consult its documentation*

6.4.3 Cumulative Sums and Iterated Differences

Recall that the \`**+**\` operator acts elementwisely on two vectors and that the **`numpy.sum`** function aggregates all values into a single one. We have one more similar function, but vectorised in a slightly different fashion. Namely, **`numpy.cumsum`** returns the vector of *cumulative sums*:

```
np.cumsum([5, 3, -4, 1, 1, 3])
## array([5, 8, 4, 5, 6, 9])
```

gave, in this order: the first element, the sum of first two elements, the sum of first three elements, …, the sum of all elements.

Further, *iterative differences* are a somewhat inverse operation:

```
np.diff([5, 8, 4, 5, 6, 9])
## array([ 3, -4,  1,  1,  3])
```

returned the difference between the 2nd and 1st element, then the difference between the 3rd and the 2nd, and so forth. The resulting vector is 1 element shorter than the input one.

We often make use of cumulative sums and iterated differences when processing time series, e.g., stock exchange data (e.g., by how much the price changed since the previous day?; Section 18.3.1) or cumulative distribution functions (Section 4.4).

6.4.4 Vectorising Scalar Operations with List Comprehensions (*)

List comprehensions of the form `[ expression for name in iterable ]` are part of base Python. They allow us to create lists based on transformed versions of individual elements in a given iterable object. Hence, they might work in cases where a task at hand cannot be solved by means of vectorised **numpy** functions.

For example, here is a way to generate squares of 10 first natural numbers:

```
[ i**2 for i in range(1, 11) ]
## [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

The result can be passed to **numpy.array** to convert it to a vector.

And if we wish to filter our all elements that are not greater than 0, we can write:

```
np.random.seed(123)  # make pseudorandom number generation reproducible
x = np.round(np.random.rand(10)*2-1, 2)
[ e for e in x if e > 0 ]
## [0.39, 0.1, 0.44, 0.96, 0.37]
```

Further, we can use the ternary operator of the form `x_true if cond else x_false` to return either `x_true` or `x_false` depending on the truth value of `cond`.

```
e = -2
e**0.5 if e >= 0 else (-e)**0.5
## 1.4142135623730951
```

Combined with a list comprehension, we can write, for instance:

```
[ round(e**0.5 if e >= 0 else (-e)**0.5, 2) for e in x ]
## [0.62, 0.66, 0.74, 0.32, 0.66, 0.39, 0.98, 0.61, 0.2, 0.47]
```

Exercise 6.10 *Write equivalent versions of the above expressions using vectorised* **`numpy`** *functions.*

Exercise 6.11 *Write equivalent versions of the above expressions using base Python lists, the* **`for`** *loop and the* **`list.append`** *method (start from an empty list which will store the result).*

Note: There is also a tool which vectorises a scalar function so that it can be used on **`numpy`** vectors:

```
def clip1(x):
    if x < 0:     return 0
    elif x > 1:   return 1
    else:         return x

clip = np.vectorize(clip1)  # returns a function object
clip([0.3, -1.2, 0.7, 4, 9])
## array([0.3, 0. , 0.7, 1. , 1. ])
```

6.5 Exercises

Exercise 6.12 *Give the mathematical definitions, use cases, and interpretations of standardisation, normalisation, and min-max scaling.*

Exercise 6.13 *How are* **`numpy.log`** *and* **`numpy.exp`** *related to each other? How about* **`numpy.log`** *vs* **`numpy.log10`**, **`numpy.cumsum`** *vs* **`numpy.diff`**, **`numpy.min`** *vs* **`numpy.argmin`**, **`numpy.sort`** *vs* **`numpy.argsort`**, *and* **`scipy.stats.rankdata`** *vs* **`numpy.argsort`**?

Exercise 6.14 *What is the difference between* **`numpy.trunc`**, **`numpy.floor`**, **`numpy.ceil`**, *and* **`numpy.round`**?

Exercise 6.15 *What happens when we apply `` `+` `` on two vectors of different lengths?*

Exercise 6.16 *List the four ways to index a vector.*

Exercise 6.17 *What is wrong with the expression `x[ x >= 0 and x <= 1 ]`, where `x` is a numeric vector? How about `x[ x >= 0 & x <= 1 ]`?*

Exercise 6.18 *(*) What does it mean that slicing returns a view on existing data?*

7

Continuous Probability Distributions (*)

Each successful data analyst will deal with hundreds or thousands of datasets in their lifetime. In the long run, at some level, most of them will be deemed *boring*. This is because a few common patterns will be occurring over and over again.

In particular, the previously mentioned bell-shapedness and right-skewness is quite prevalent in nature. But, surprisingly, this is exactly when things become scientific and interesting – allowing us to study various phenomena at an appropriate level of generality.

Mathematically, such idealised patterns in the histogram shapes can be formalised using the notion of a *probability distribution of a continuous, real-valued random variable*, and more precisely in this case, a *probability density function* of the corresponding random variable.

Intuitively[79], a *probability density function* is a nicely smooth curve that would arise if we drew a histogram for the whole *population* (e.g., all women living currently on Earth and beyond or otherwise an extremely large data sample obtained by independently querying the same underlying data generating process) in such a way that the total area of all the bars are equal to 1 and the the bin sizes being very small.

As stated at the beginning, we do not intend this to be a course in probability theory and mathematical statistics, but a one that precedes and motivates them (e.g., [[Bil95], [DKLM05], [Gen09], [Gen20]]), therefore our definitions are out of necessity simplified so that they are digestible. Hence, for the purpose of our illustrations, let us consider the following characterisation.

Important: (**) We call an integrable function $f : \mathbb{R} \to \mathbb{R}$ a *probability density function* (PDF) if $f(x) \geq 0$ for all x and $\int_{-\infty}^{\infty} f(x)\, dx = 1$, i.e., it is nonnegative and normalised in such a way that the total area under the whole curve is 1.

For any $a < b$, we treat the area under the fragment of the $f(x)$ curve for x between a

[79] (**) Fear not, this intuition is theoretically grounded and is based on the asymptotic behaviour of the histograms as the estimators of the underlying probability density function, see, e.g., [[FD81]] and the many references therein.

and b, i.e., $\int_a^b f(x)\,dx$, as the probability of the underlying real-valued random variable (theoretical data generating process) falling into the $[a, b]$ interval.

Some distributions appear more frequently than others and fit empirical data or parts thereof particularly well (or there is some established wishful thinking/delusion that they serve as their good-enough approximations). Let us thus review a few noteworthy probability distributions: the normal, log-normal, Pareto, and uniform families (in Section 18.2.4 we also mention the exponential one).

7.1 Normal Distribution

A *normal distribution* is the one that has a prototypical, nicely symmetric, bell-shaped density. It is described by two parameters: $\mu \in \mathbb{R}$ (the expected value, at which the density is centred) and $\sigma > 0$ (the standard deviation, saying how much the distribution is dispersed around μ).

The probability density function of N(μ, σ) is given by:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2} \right).$$

7.1.1 Estimating Parameters

A course in statistics (which, again, this one is not, we are merely making an illustration here), may tell us that the sample arithmetic mean and standard deviation are natural, statistically well-behaving *estimators* of the said parameters: if all samples would really be drawn independently from N(μ, σ) each, the we *expect* the mean and standard deviation be equal to, more or less, μ and σ (the larger the sample size, the smaller the error).

Recall the `heights` (females from the NHANES study) dataset and its bell-shaped histogram in Figure 4.2. Let's estimate these parameters for this sample:

```
heights = np.loadtxt("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/nhanes_adult_female_height_2020.txt")
mu = np.mean(heights)
sigma = np.std(heights, ddof=1)
mu, sigma
## (160.13679222932953, 7.062858532891359)
```

Mathematically, we will denote these two with $\hat{\mu}$ and $\hat{\sigma}$ (mu and sigma with a hat) to emphasise that they are merely guesstimates of the unknown respective parameters μ and σ. On a side note, we use `ddof=1` in the latter case, because this estimator has slightly better statistical properties.

Let's draw the fitted density function, i.e., the PDF of N(160.1, 7.06), on top of the histogram, see Figure 7.1. We pass `stat="density"` to **`seaborn.histplot`** so that the histogram bars are normalised (i.e., the total area of these rectangles sums to 1).

```
sns.histplot(heights, stat="density")
x = np.linspace(np.min(heights), np.max(heights), 1000)
plt.plot(x, scipy.stats.norm.pdf(x, mu, sigma), "r--",
    label=f"PDF of N({mu:.1f}, {sigma:.2f})")
plt.legend()
plt.show()
```

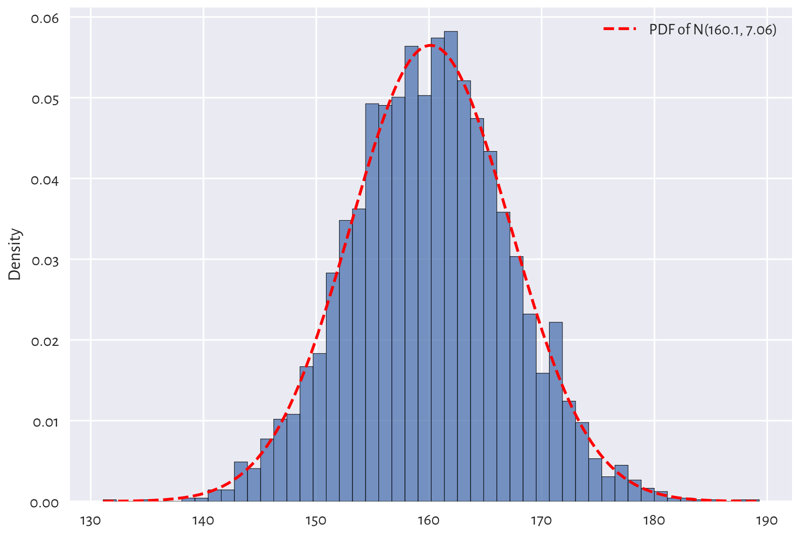


Figure7.1: A histogram and the probability density function of the fitted normal distribution for the `heights` dataset

At a first glance, this is a very nice match. Before proceeding with some ways of assessing the goodness of fit slightly more rigorously, let us praise the potential benefits of having an idealised *model* of our dataset at our disposal.

Important: (**) It might be the case that we will have to obtain the estimates of the probability distribution's parameters by numerical optimisation, for example, by minimising

$$\mathcal{L}(\mu, \sigma) = \sum_{i=1}^{n} \left(\frac{(x_i - \mu)^2}{\sigma^2} + \log \sigma^2 \right)$$

with respect to μ and σ (corresponding to the objective function in maximum likelihood estimation problem for the normal distribution family). In our case, however, we are lucky; there exist open-form formulae expressing the solution to the above in the form of the aforementioned sample mean and standard deviation, but for other distributions things can get a little trickier.

Sometimes, we will have many options for point estimators to choose from, which might be more suitable if data aren't of top quality (e.g., contain outliers). For instance, in the normal model, it can be shown that we can also estimate μ and σ via the sample median and IQR/1.349.

7.1.2 Data Models Are Useful

If (provided that, assuming that, on condition that) our sample is a realisation of independent random variables following a given distribution, or a data analyst judges that such an approximation might be justified, then we have a set of many numbers *reduced* to merely few parameters.

In the above case, we might want to risk the statement that data follow the normal distribution (assumption 1) with parameters $\mu = 160.1$ and $\sigma = 7.06$ (assumption 2). Note that the choice of the distribution family is one thing, and the way we estimate the underlying parameters (in our case, we use $\hat{\mu}$ and $\hat{\sigma}$) is another.

This not only saves storage space and computational time, but also – based on what we can learn from a course in probability and statistics (by appropriately integrating the density function) – we can imply facts such as for normally distributed data:

- ca. 95% of (i.e., *most*) women are $\mu \pm 2\sigma$ tall (the 2σ rule),
- ca. 99.7% of (i.e., *almost all*) women are $\mu \pm 3\sigma$ tall (the 3σ rule).

Also, if we knew that the distribution of heights of men is also normal with some other parameters, we could be able to make some comparisons between the two samples. For example, what is the probability that a woman randomly selected from the crowd is taller than a male passerby.

Furthermore, there is a range of *parametric* (assuming some distribution family) of statistical methods that cannot be used if we don't assume the data normality, e.g., the t-test to compare the expected values. So many options.

Exercise 7.1 *How different manufacturing industries can make use of such knowledge? Are simplifications necessary when dealing with complexity? What are the alternatives?*

7.2 Assessing Goodness of Fit

7.2.1 Comparing Cumulative Distribution Functions

A better way of assessing the extent to which a sample deviates from a hypothesised distribution, is by comparing the theoretical *cumulative distribution function* (CDF) and the empirical one ($\hat{F}_n$, see Section 4.4).

Important: If f is a density function, then the corresponding theoretical CDF is defined as $F(x) = \int_{-\infty}^{x} f(t)\, dt$, i.e., the probability of the underlying random variable's taking a value not greater than x.

By definition[80], each CDF takes values in the unit interval ($[0, 1]$) and is nondecreasing.

For the normal distribution family, the values of the theoretical CDF can be computed by calling **scipy.stats.norm.cdf**; see Figure 7.2.

```
n = len(heights)
x = np.linspace(np.min(heights), np.max(heights), 1001)
probs = scipy.stats.norm.cdf(x, mu, sigma)  # sample the CDF at many points
plt.plot(x, probs, "r--", label=f"CDF of N({mu:.1f}, {sigma:.2f})")
heights_sorted = np.sort(heights)
plt.plot(heights_sorted, np.arange(1, n+1)/n,
    drawstyle="steps-post", label="Empirical CDF")
```

(continues on next page)

[80] Actually, the probability distribution of any real-valued random variable X can be uniquely defined by means of a nondecreasing, right (upward) continuous function $F : \mathbb{R} \to [0, 1]$ such that $\lim_{x\to-\infty} F(x) = 0$ and $\lim_{x\to\infty} F(x) = 1$, in which case $\Pr(X \leq x) = F(x)$. The probability density function only exists for continuous random variables and is defined as the derivative of F.

(continued from previous page)

```
plt.xlabel("$x$")
plt.ylabel("Prob(height $\\leq$ x)")
plt.legend()
plt.show()
```

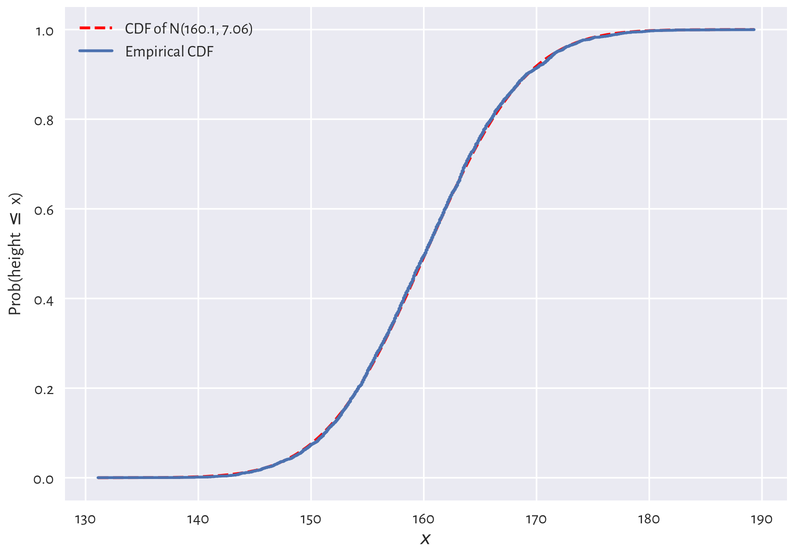


Figure7.2: The empirical CDF and the fitted normal CDF for the `heights` dataset

This looks like a superb match.

Important: $F(b) - F(a) = \int_a^b f(t)\,dt$ is the probability of generating a value in the interval $[a, b]$.

Let us empirically verify the aforementioned 3σ rule:

```
F = scipy.stats.norm.cdf
F(mu+3*sigma, mu, sigma)-F(mu-3*sigma, mu, sigma)
## 0.9973002039367398
```

Indeed, almost all observations are within $[\mu - 3\sigma, \mu + 3\sigma]$, if data are normally distributed.

Note: (*) A common way to summarise the discrepancy between the empirical and a given theoretical CDF is by computing the greatest absolute deviation between these two functions

$$D_n = \sup_{t\in\mathbb{R}} |\hat{F}_n(t) - F(t)|.$$

Note that the supremum is a continuous version of the maximum.

It holds:

$$D_n = \max_{k=1,\dots,n} \left\{ \max\left\{ \left|\tfrac{k-1}{n} - F(x_{(k)})\right| \right\}, \max\left\{ \left|\tfrac{k}{n} - F(x_{(k)})\right| \right\} \right\},$$

i.e., F needs to be probed only at the n points from the sorted input sample.

```
Fx = scipy.stats.norm.cdf(heights_sorted, mu, sigma)
k = np.arange(1, n+1)  # 1, 2, ..., n
Dn1 = np.max(np.abs((k-1)/n - Fx))
Dn2 = np.max(np.abs(k/n - Fx))
Dn = max(Dn1, Dn2)
Dn
## 0.010470976524201148
```

If the difference is *sufficiently small* (the larger the sample size, the less tolerant we should be with regards to the size of this disparity), then we can assume that a normal model describes data quite well. This is indeed the case here: we may estimate the probability of someone being as tall as any given height with error less than about 1%.

Note: (**) More precisely, the popular goodness-of-fit test by Kolmogorov and Smirnov can give us a conservative tolerance margin for the concrete value of D_n as a function of n within the framework of frequentist hypotheses testing. Refer to a course in mathematical statistics for more. A more advanced reader who is already equipped with some deeper understanding of what p-values are, might be curious to know that the said test suggests not to reject the hypothesis of normality of data:

```
scipy.stats.kstest(heights, scipy.stats.norm.cdf, (mu, sigma))
## KstestResult(statistic=0.010470976524201148, pvalue=0.7395860276984034)
```

7.2.2 Comparing Quantiles (*)

A Q-Q (quantile-quantile) plot is another graphical method for comparing two distributions. This time, instead of working with a cumulative distribution function F, we will be dealing with their (generalised) inverses, i.e., quantile functions.

Given a CDF F, the corresponding *quantile function* is defined for any $p \in (0, 1)$ as

$$Q(p) = \inf\{x : F(x) \geq p\},$$

i.e., the smallest x such that the probability of drawing a value not greater than x is at least p.

Important: If a CDF F is continuous, and this is the assumption in the current chapter, then Q is exactly its inverse, i.e., it holds $Q(p) = F^{-1}(p)$ for all $p \in (0, 1)$.

The theoretical quantiles can be generated by the **scipy.stats.norm.ppf** function. Here, *ppf* stands for the percent point function which is another (yet quite esoteric) name for the above Q.

For instance, given our N(160.1, 7.06)-distributed NHANES dataset, $Q(0.9)$ is the height not exceeded by 90% of female population. In other words, only 10% of American women are taller than:

```
scipy.stats.norm.ppf(0.9, mu, sigma)
## 169.18820963937648
```

The sample quantiles that we have introduced in Section 5.1.3 are natural estimators of the theoretical quantile function. However, we have also mentioned that there are quite a few possible definitions thereof in the literature, compare [[HF96]].

A Q-Q plot draws a version of sample quantiles as a function of the corresponding theoretical quantiles. For simplicity, instead of using the **numpy.quantile** function, we will assume that the $\frac{i}{n+1}$-quantile is equal to $x_{(i)}$, i.e., the i-th smallest value in a given sample $(x_1, x_2, \dots, x_n)$ and consider only $i = 1, 2, \dots, n$.

Our simplified setting successfully avoids the problem which arises when the 0- or 1-quantile of the theoretical distribution, i.e., $Q(0)$ and $Q(1)$ is infinite (and this is the case for the normal distribution family).

```
n = len(heights)
q = np.arange(1, n+1)/(n+1)  # 1/(n+1), 2/(n+2), ..., n/(n+1)
```

(continues on next page)

(continued from previous page)

```
heights_sorted = np.sort(heights)         # theoretical quantiles
quantiles = scipy.stats.norm.ppf(q, mu, sigma)  # sample quantiles
plt.plot(quantiles, heights_sorted, "o")
plt.axline((heights_sorted[n//2], heights_sorted[n//2]), slope=1,
    linestyle=":", color="gray")
plt.xlabel(f"Quantiles of N({mu:.1f}, {sigma:.2f})")
plt.ylabel("Sample quantiles")
plt.show()
```

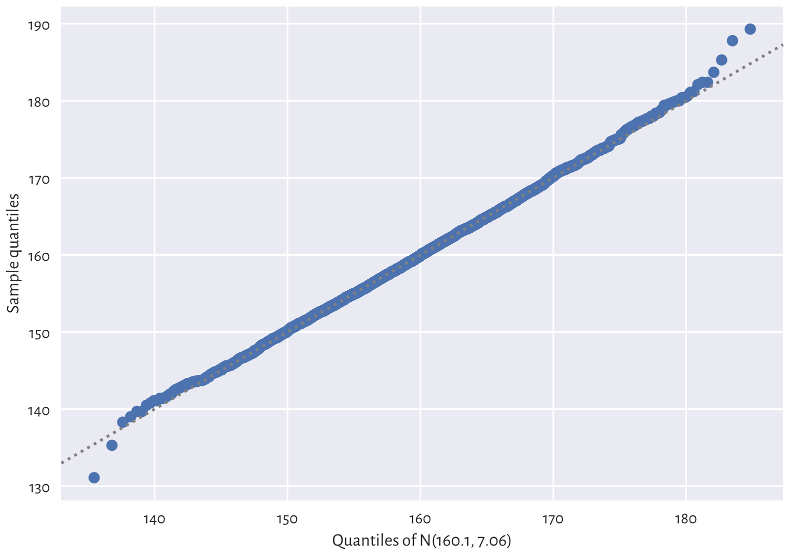


Figure7.3: Q-Q plot for the `heights` dataset

Figure 7.3 depicts the Q-Q plot for our example dataset.

Ideally, the points should be arranged on the $y = x$ line (which was added for readability). This happens if the sample quantiles match the theoretical ones perfectly.

In our case, there are small discrepancies in the tails (e.g., the smallest observation was slightly smaller than expected, and the largest one was larger than expected), although it is quite *normal* a behaviour for small samples and certain distribution families

Overall, we can say that this is a very good fit.

7.3 Other Noteworthy Distributions

7.3.1 Log-normal Distribution

We say that a sample is *log-normally distributed*, if its logarithm is normally distributed.

In particular, it is sometimes observed that the income of most individuals (except the richest) is distributed, at least approximately, log-normally. Let us investigate whether this is the case for the UK taxpayers.

```
income = np.loadtxt("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/uk_income_simulated_2020.txt")
```

The plotting of the histogram for the logarithm of income is left as an exercise (we can pass `log_scale=True` to **`seaborn.histplot`**; we'll plot it soon anyway in a different way). We proceed directly with the fitting of a log-normal model, LN(μ, σ). The fitting process is similar to the normal case, but this time we determine the mean and standard deviation based on the logarithms of data, which we can compute using a vectorised mathematical function discussed in Section 6.1.

```
lmu = np.mean(np.log(income))
lsigma = np.std(np.log(income), ddof=1)
lmu, lsigma
## (10.314409794364623, 0.5816585197803816)
```

Figure 7.4 depicts the fitted probability density function together with the histograms on the log- and original scale. When creating this plot, there are two pitfalls, though. Firstly, **`scipy.stats.lognorm`** parametrises the distribution via the parameter s equal to σ and scale equal to e^{μ}, hence computing the PDF at different points must be done as follows:

```
x  = np.linspace(np.min(income), np.max(income), 101)
fx = scipy.stats.lognorm.pdf(x, s=lsigma, scale=np.exp(lmu))
```

Second, passing both `log_scale=True` and `stat="density"` to **`seaborn.histplot`** does not normalise the values on the y-axis correctly. In order to make the histogram on the log-scale comparable with the true density, we need to turn the log-scale manually and pass manually generated bins that are equidistant on the logarithmic scale (via **numpy.geomspace**).

```
b = np.geomspace(np.min(income), np.max(income), 30)
```

And now:

```
plt.subplot(121)
sns.histplot(income, stat="density", bins=b)  # own bins!
plt.xscale("log")
plt.plot(x, fx, "r--")
print(plt.xlim())
plt.subplot(122)
sns.histplot(income, stat="density")
plt.plot(x, fx, "r--", label=f"PDF of LN({lmu:.1f}, {lsigma:.2f})")
plt.legend()
plt.show()
```

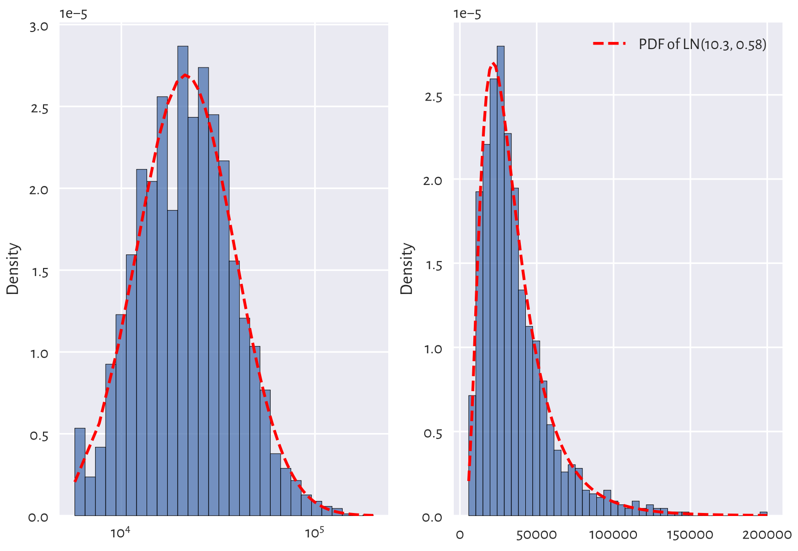


Figure7.4: A histogram and the probability density function of the fitted log-normal distribution for the `income` dataset, on log- (left) and original (right) scale

Overall, this fit is not too bad. Note that we deal with only a sample of 1000 households here; the original UK Office of National Statistics data[76] could tell us more about the quality of this model in general, but it's beyond the scope of our simple exercise.

Exercise 7.2 *Compare the empirical CDF for* `income` *and the theoretical CDF of LN(10.3, 0.58).*

Furthermore, Figure 7.5 gives the quantile-quantile plot on a double-logarithmic scale for the above log-normal model. Additionally, we (empirically) verify the hypothesis of normality (using a "normal" normal distribution, not it's "log" version).

```python
n = len(income)
q = np.arange(1, n+1)/(n+1)
income_sorted = np.sort(income)

plt.subplot(121)
quantiles = scipy.stats.lognorm.ppf(q, s=lsigma, scale=np.exp(lmu))
plt.plot(quantiles, income_sorted, "o")
plt.axline((income_sorted[n//2], income_sorted[n//2]), slope=1,
    linestyle=":", color="gray")
plt.xlabel(f"Quantiles of LN({lmu:.1f}, {lsigma:.2f})")
plt.ylabel("Sample quantiles")
plt.xscale("log")
plt.yscale("log")

plt.subplot(122)
mu = np.mean(income)
sigma = np.std(income, ddof=1)
quantiles2 = scipy.stats.norm.ppf(q, mu, sigma)
plt.plot(quantiles2, income_sorted, "o")
plt.axline((income_sorted[n//2], income_sorted[n//2]), slope=1,
    linestyle=":", color="gray")
plt.xlabel(f"Quantiles of N({mu:.1f}, {sigma:.2f})")

plt.show()
```

We see that definitely the hypothesis that our data follow a normal distribution (the right subplot) is most likely false.

The log-normal model (the left subplot), on the other hand, might be quite usable. It again reduced the whole dataset to merely two numbers, μ and σ, based on which (and probability theory), we may deduce that:

[76] https://www.ons.gov.uk/peoplepopulationandcommunity/personalandhouseholdfinances/incomeandwealth/bulletins/householddisposableincomeandinequality/financialyear2020

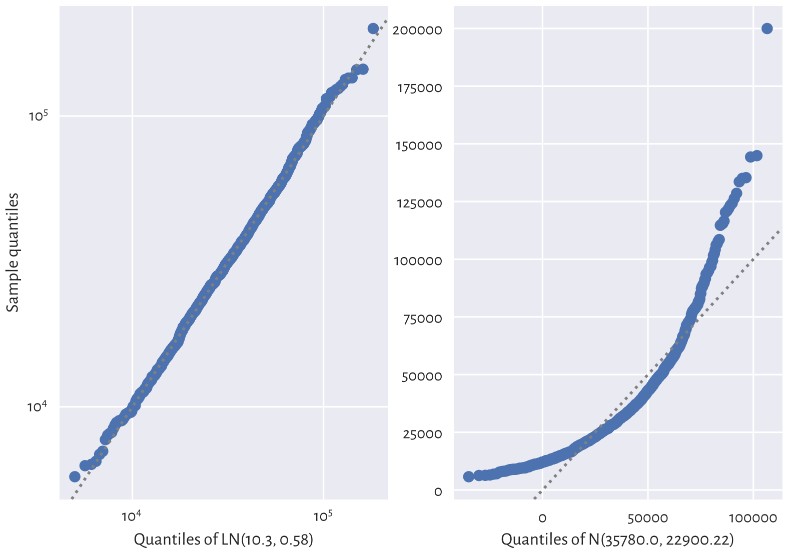


Figure 7.5: Q-Q plots for the `income` dataset vs a fitted log-normal (good fit; left) and normal (bad fit; right) distribution

- the expected average (mean) income is $e^{\mu+\sigma^2/2}$,
- median is e^{μ},
- most probable one (mode) in $e^{\mu-\sigma^2}$,

etc.

Note: Recall again that for skewed distributions such as this one, reporting the mean might be misleading. This is why *most* people get angry when they read the news about the prospering economy – "yeah, I'd like to see that kind of money in my pocket". Hence, it's not only μ that matters, it's also σ which quantifies the discrepancy between the rich and the poor (too much inequality is bad, but also too much uniformity is to be avoided).

Note that for a normal distribution the situation is very different, because, the mean, median, and most probable outcomes tend to be the same – the distribution is symmetric around μ.

Exercise 7.3 *(*) What is the fraction of people with earnings below the mean in our LN(10.3, 0.58) model?*

7.3.2 Pareto Distribution

Consider again the dataset[77] on the populations of the US cities in the 2000 US Census:

```
cities = np.loadtxt("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/other/us_cities_2000.txt")
len(cities), sum(cities)  # number of cities, total population
## (19447, 175062893.0)
```

Figure 7.6 gives the histogram of the city sizes with the populations on the log-scale. It kind-of looks like a log-normal distribution again, which the reader can inspect themself when they are feeling playful.

```
sns.histplot(cities, bins=20, log_scale=True)
plt.show()
```

We, however, this time will be interested in not what's *typical*, but what's in some sense *anomalous* or *extreme*. Let's take a look at the *truncated* version of the city size distribution by considering the cities with 10,000 or more inhabitants – i.e., we will only study the right tail the original data:

```
s = 10_000
large_cities = cities[cities >= s]
len(large_cities), sum(large_cities)  # number of cities, total population
## (2696, 146199374.0)
```

Plotting the above on a double-logarithmic scale can be performed by passing `log_scale=(True, True)` to **`seaborn.histplot`**, which is left as an exercise. Anyway, doing so will lead to a similar picture as in Figure 7.7. This reveals something very interesting: the bar tops on the double-log-scale are arranged more or less on a straight line. There are many datasets which exhibit this behaviour; we say that they follow a *power law* (power in the arithmetic sense, not social one), see [[CSN09], [New05]] for discussion.

Let's introduce the *Pareto distribution* family which has a prototypical power law-like density. It is identified by two parameters:

- the (what **`scipy`** calls it) scale parameter $s > 0$ is equal to the shift from 0,

[77] https://arxiv.org/abs/0706.1062v2

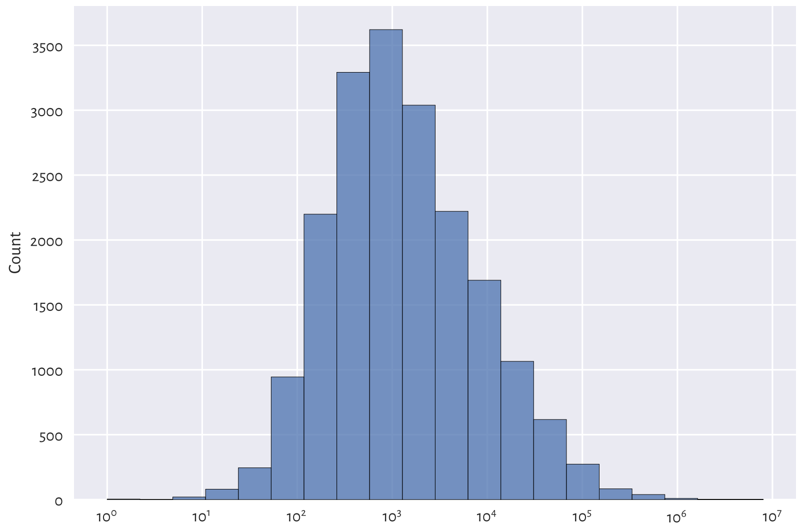


Figure7.6: Histogram for the unabridged `cities` dataset

- the shape parameter, $\alpha > 0$, controls the slope of the said line on the double-log-scale.

The probability density function of P(α, s) is given by:

$$f(x) = \frac{\alpha s^\alpha}{x^{\alpha+1}}$$

for $x \geq s$ and $f(x) = 0$ otherwise.

s is usually taken as the sample minimum (i.e., 10000 in our case). α can be estimated through the reciprocal of the mean of the scaled logarithms of our observations:

```
alpha = 1/np.mean(np.log(large_cities/s))
alpha
## 0.9496171695997675
```

Unfortunately, we have already noted that comparing the theoretical densities and an empirical histogram on a log-scale is quite problematic with **seaborn**, therefore we again must apply logarithmic binning manually in order to come up with what we see in Figure 7.7.

```
b = np.geomspace(s, np.max(large_cities), 21)  # bins' boundaries
sns.histplot(large_cities, bins=b, stat="density")
plt.plot(b, scipy.stats.pareto.pdf(b, alpha, scale=s), "r--",
    label=f"PDF of P({alpha:.3f}, {s})")
plt.xscale("log")
plt.yscale("log")
plt.legend()
plt.show()
```

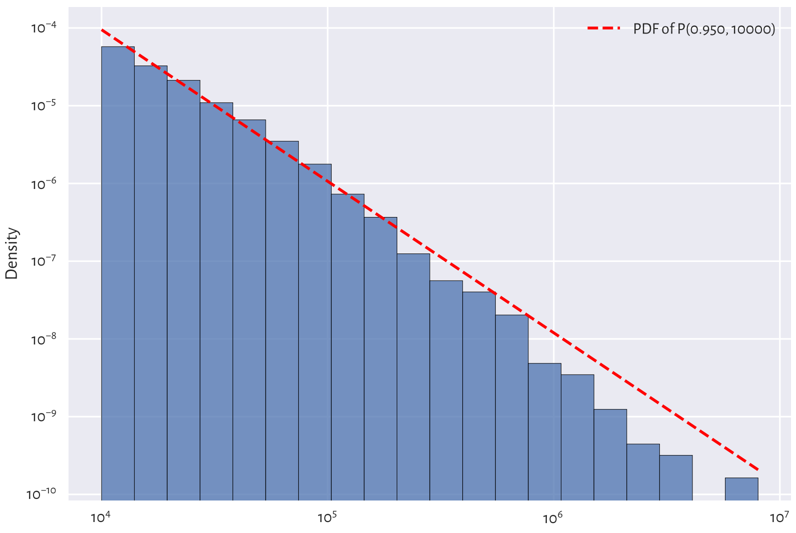


Figure7.7: Histogram for the `large_cities` dataset and the fitted density on a double log-scale

Figure 7.8 gives the corresponding Q-Q plot on a double-logarithmic scale.

```
n = len(large_cities)
q = np.arange(1, n+1)/(n+1)
cities_sorted = np.sort(large_cities)
```

(continues on next page)

(continued from previous page)

```
quantiles = scipy.stats.pareto.ppf(q, alpha, scale=s)
plt.plot(quantiles, cities_sorted, "o")
plt.axline((quantiles[n//2], cities_sorted[n//2]), slope=1,
    linestyle=":", color="gray")
plt.xlabel(f"Quantiles of P({alpha:.3f}, {s})")
plt.ylabel("Sample quantiles")
plt.xscale("log")
plt.yscale("log")
plt.show()
```

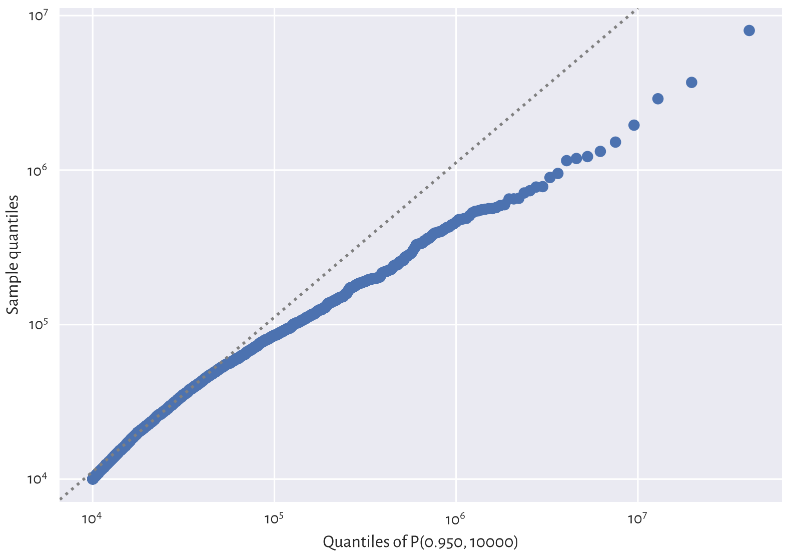


Figure7.8: Q-Q plot for the `large_cites` dataset vs the fitted Paretian model

We see that the populations of the largest cities are overestimated. The model could be better, but the cities are still growing, right?

Example 7.4 *(*) It might also be interesting to see how well can we predict the probability of a randomly selected city being at least a given size. Let us thus denote with* $S(x) = 1 - F(x)$ *the* complementary cumulative distribution function *(CCDF; sometimes referred to as the* *survival function) and with $\hat{S}_n(x) = 1 - \hat{F}_n(x)$ its empirical version. Figure 7.9 compares the empirical and the fitted CCDFs with probabilities on the linear- and log-scale.*

```
x = np.geomspace(np.min(large_cities), np.max(large_cities), 1001)
probs = scipy.stats.pareto.cdf(x, alpha, scale=s)
for i in [1, 2]:
    plt.subplot(1, 2, i)
    plt.plot(x, 1-probs, "r--", label=f"CCDF of P({alpha:.3f}, {s})")
    plt.plot(cities_sorted, 1-np.arange(1, n+1)/n,
        drawstyle="steps-post", label="Empirical Complementary CDF")
    plt.xlabel("$x$")
    plt.xscale("log")
    plt.yscale(["linear", "log"][i-1])
    if i == 1:
        plt.ylabel("Prob(city size > x)")
        plt.legend()
plt.show()
```

In terms of the maximal absolute distance between the two functions, D_n, the fit is actually pretty good (this can be seen on the left plot, note that the log-scale overemphasises the relatively small differences in the right tail and should not be used for judging the value of D_n).

7.3.3 Uniform Distribution

Consider the Polish *Lotto* lottery, where 6 numbered balls $\{1, 2, \dots, 49\}$ are drawn without replacement from an urn. We have a data set that summarises the number of times each ball has been drawn in all the drawings in the period 1957–2016.

```
lotto = np.loadtxt("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/lotto_table.txt")
lotto
## array([720., 720., 714., 752., 719., 753., 701., 692., 716., 694., 716.,
##        668., 749., 713., 723., 693., 777., 747., 728., 734., 762., 729.,
##        695., 761., 735., 719., 754., 741., 750., 701., 744., 729., 716.,
##        768., 715., 735., 725., 741., 697., 713., 711., 744., 652., 683.,
##        744., 714., 674., 654., 681.])
```

Each event seems to occur more or less with the same probability. Of course, the numbers on the balls are integer, but in our idealised scenario we may try modelling this dataset using a continuous *uniform distribution*, which yields arbitrary real numbers on a given interval (a, b), i.e., between some a and b. We denote such a distribution with

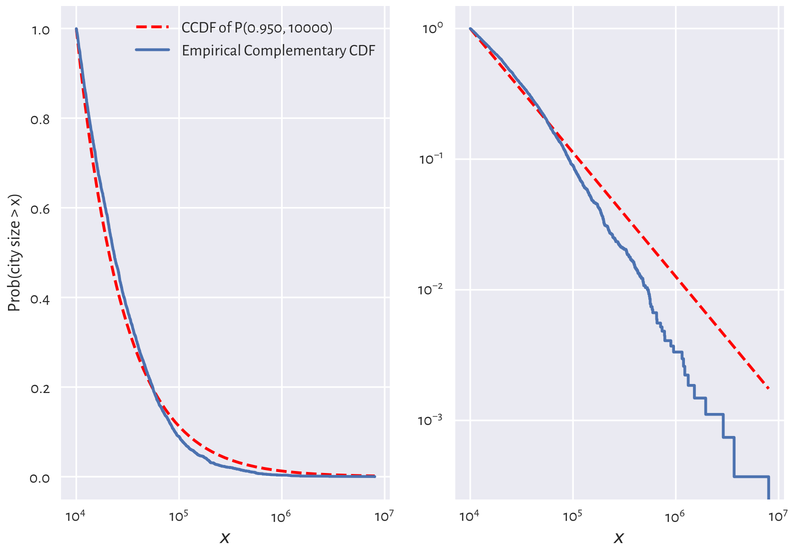


Figure7.9: Empirical and theoretical complementary cumulative distribution functions for the `large_cities` dataset with probabilities on the linear- (left) and log-scale (right) and city sizes on the log-scale

U(a, b). It has the probability density function given by

$$f(x) = \frac{1}{b-a},$$

for $x \in (a, b)$ and $f(x) = 0$ otherwise.

Note that **`scipy.stats.uniform`** uses parameters `a` and `scale` equal to $b - a$ instead.

In our case, it makes sense to set *a=1* and *b=50* and interpret an outcome like *49.1253* as representing the 49th ball (compare the notion of the floor function, $\lfloor x \rfloor$).

```
x = np.linspace(1, 50, 1001)
plt.bar(np.arange(1, 50), width=1, height=lotto/np.sum(lotto),
    edgecolor="black", align="edge")
plt.plot(x, scipy.stats.uniform.pdf(x, 1, scale=49), "r--",
    label="PDF of U(1, 50)")
```

(continues on next page)

(continued from previous page)

```
plt.legend()
plt.show()
```

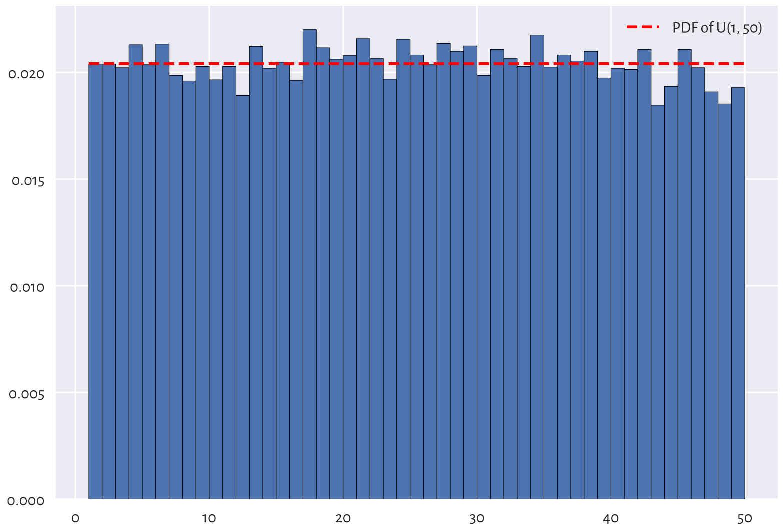


Figure7.10: Histogram for the `lotto` dataset

Visually, see Figure 7.10, this model makes much sense, but again, some more rigorous statistical testing would be required to determine if someone hasn't been tampering with the lottery results, i.e., if data does not deviate from the uniform distribution significantly.

Exercise 7.5 *Does playing lotteries and engaging in gambling make* rational *sense at all, from the perspective of an individual player? Well, we see that 16 is the most frequently occurring outcome in* Lotto, *maybe there's some magic in it? Also, some people became millionaires after all, right?*

Note: In data modelling (e.g., Bayesian statistics), sometimes a uniform distribution is chosen as a placeholder for "we know nothing about a phenomenon, so let's just assume that every event is equally likely". However, overall, it is quite fascinating that the real world tends to be structured after all; emerging patterns are plentiful and, furthermore, they are subject to qualitative analysis.

7.3.4 Distribution Mixtures (**)

Some datasets may fail to fit into simple models such as the ones describe above. It may sometimes be due to their non-random behaviour: statistics gives just one means to create data idealisations, we also have partial differential equations, approximation theory, graphs and complex networks, agent-based modelling, and so forth, which might be worth giving a study (and then try).

Other reasons may be that what we observe is in fact a *mixture* (creative combination) of simpler processes.

The dataset representing the December 2021 hourly averages pedestrian counts[78] near the Southern Cross Station in Melbourne might be a good instance of such a scenario, compare Figure 4.6.

```
peds = np.loadtxt("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/southern_cross_station_peds_2019_dec.txt")
```

It might not be a bad idea to try to fit a probabilistic (convex) combination of three normal distributions f_1, f_2, f_3, corresponding to the morning, lunch-time, and evening pedestrian count peaks. This yields the PDF

$$f(x) = w_1 f_1(x) + w_2 f_2(x) + w_3 f_3(x)$$

for some coefficients $w_1, w_2, w_3 \geq 0$ such that $w_1 + w_2 + w_3 = 1$.

Figure 7.11 depicts a mixture of N(8, 1), N(12, 1), and N(17, 2) with the corresponding weights of 0.35, 0.1, and 0.55. This particular data set is quite coarse-grained (we only have 24 bar heights at our disposal), therefore these estimated parameters and coefficients should be taken with a pinch of chilli pepper.

```
plt.bar(np.arange(24), width=1, height=peds/np.sum(peds), edgecolor="black")
x = np.arange(0, 25, 0.1)
p1 = scipy.stats.norm.pdf(x, 8, 1)
p2 = scipy.stats.norm.pdf(x, 12, 1)
p3 = scipy.stats.norm.pdf(x, 17, 2)
p = 0.35*p1 + 0.1*p2 + 0.55*p3  # weighted (convex) combination of 3
 ↪densities
```

(continues on next page)

[78] http://www.pedestrian.melbourne.vic.gov.au/

(continued from previous page)

```
plt.plot(x, p, "r--", label="PDF of a normal mixture")
plt.legend()
plt.show()
```

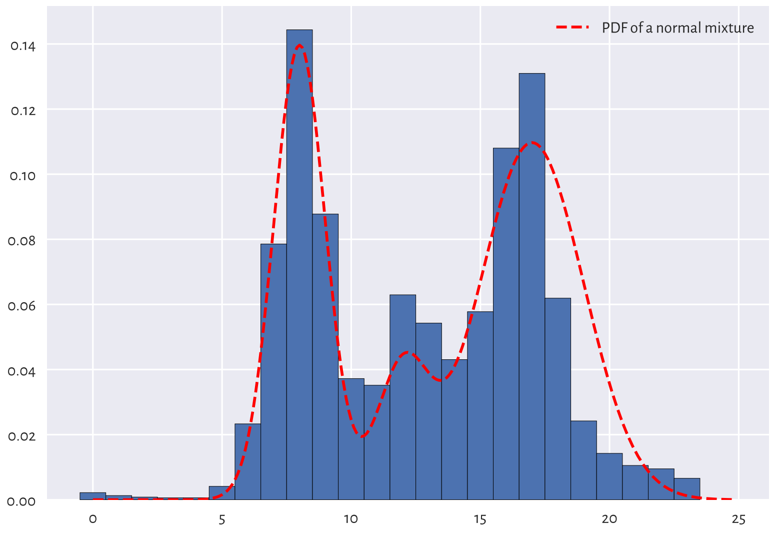


Figure7.11: Histogram for the `peds` dataset and a guesstimated mixture of three normal distributions

---

Important: It will frequently be the case in data wrangling that more complex entities (models, methods) will be arising as combinations of simpler (primitive) components. This is why we should spend a great deal of time studying the *fundamentals*.

Note: Some data clustering techniques (in particular, the k-means algorithm that we briefly discuss later in this course) could be used to split a data sample into disjoint chunks corresponding to different mixture components.

Also, it might be the case that the mixture components can in fact be explained by another categorical variable which divides the dataset into natural groups, compare Chapter 13.

7.4 Generating Pseudorandom Numbers

A probability distribution is useful not only for describing a dataset. It also enables us to perform many experiments on data that we don't currently have, but we might obtain in the future, to test various scenarios and hypotheses.

To do this, we can generate a random sample of independent (not related to each other) observations.

7.4.1 Uniform Distribution

When most people say *random*, they implicitly mean *uniformly distributed*. For example:

```
np.random.rand(5)
## array([0.69646919, 0.28613933, 0.22685145, 0.55131477, 0.71946897])
```

gives 5 observations sampled independently from the uniform distribution on the unit interval, i.e., U(0, 1).

The same with **`scipy`**, but this time the support will be *(-10, 15)*.

```
scipy.stats.uniform.rvs(-10, scale=25, size=5)  # from -10 to -10+25
## array([ 0.5776615 , 14.51910496,  7.12074346,  2.02329754, -0.19706205])
```

Which we could as well have generated by manually shifting and scaling the output the random number generator on the unit interval using the formula `np.random.rand(5)*25-10`.

7.4.2 Not Exactly Random

Actually, we are generating numbers using a computer, which is purely deterministic, hence we shall refer to them as *pseudorandom* or random-like ones (albeit they are non-distinguishable from truly random, when subject to rigorous tests for randomness).

To prove it, we can set the initial state of the number generator (the *seed*) to some number and see what values are output:

```
np.random.seed(123)  # set seed
np.random.rand(5)
## array([0.69646919, 0.28613933, 0.22685145, 0.55131477, 0.71946897])
```

and then set the seed once again to the same number and see how "random" the next values are:

```
np.random.seed(123)  # set seed
np.random.rand(5)
## array([0.69646919, 0.28613933, 0.22685145, 0.55131477, 0.71946897])
```

This enables us to perform completely *reproducible* numerical experiments, and this is a very good feature: truly scientific inquiries should lead to identical results in the same conditions.

Note: If we don't set the seed manually, it will be initialised based on the current wall time, which is different every… time – therefore the numbers will *seem* random to us.

Many Python packages that we will be using in the future, including **pandas** and **sklearn** rely on **numpy**'s random number generator, thus we'll be calling **numpy.random.seed** to make them predictable.

Additionally, many of them (e.g., **sklearn.model_selection.train_test_split** or **pandas.DataFrame.sample**) are equipped with the `random_state` argument, which can *temporarily* change the seed (for just one call to that function). For instance:

```
scipy.stats.uniform.rvs(size=5, random_state=123)
## array([0.69646919, 0.28613933, 0.22685145, 0.55131477, 0.71946897])
```

This gives the same sequence as above.

7.4.3 Sampling from Other Distributions

Of course, generating data from other distributions is possible too; there are many **rvs** methods implemented in **scipy.stats**. For example, here is a sample from N(100, 16):

```
scipy.stats.norm.rvs(100, 16, size=3, random_state=50489)
## array([113.41134015,  46.99328545, 157.1304154 ])
```

Important: Conclusions based on simulated data are trustworthy, because they cannot be manipulated. Or can they?

The pseudorandom number generator's seed used above, `50489`, is quite suspicious. It might suggest that someone wanted to *prove* some point (in this case, the violation of the 3σ rule).

This is why we recommend sticking to one and only seed most of the time, e.g., `123`, or – when performing simulations – setting consecutive seeds for each iteration, `1`, `2`, …

Exercise 7.6 *Generate 1000 pseudorandom numbers from the log-normal distribution.*

Note: (*) Having a good pseudorandom number generator from the uniform distribution on the unit interval is crucial, because sampling from other distributions usually involves transforming independent U(0, 1) variates.

For instance, samples following any continuous cumulative distribution function F can be constructed by means of *inverse transform sampling*:

1. Generate a sample $x_1, \dots, x_n$ independently from U(0, 1);
2. Transform each x_i by applying the quantile function, $y_i = F^{-1}(x_i)$;
3. Now $y_1, \dots, y_n$ follows the CDF F.

For more topics on random number generation, see [[Gen03], [RC04]].

Exercise 7.7 *(*) Generate 1000 pseudorandom numbers from the log-normal distribution using inverse transform sampling.*

Exercise 7.8 *(**) Generate 1000 pseudorandom numbers from the distribution mixture discussed in Section 7.3.4.*

7.4.4 Natural Variability

Note that even a sample which we know that was generated from a specific distribution will deviate from it, sometimes considerably. Such effects usually disappear[81] when the the availability of data increases, but definitely will be visible for small sample sizes.

For example, Figure 7.12 depicts the histogram of 9 different samples of size 100, all drawn independently from the normal distribution N(0, 1).

[81] Compare the Fundamental Theorem of Statistics a.k.a. the Glivenko–Cantelli theorem.

```
x = np.linspace(-4, 4, 1001)
fx = scipy.stats.norm.pdf(x)  # PDF of N(0, 1)
for i in range(9):
    plt.subplot(3, 3, i+1)
    sample = scipy.stats.norm.rvs(size=100, random_state=i+1)
    sns.histplot(sample, stat="density", bins=10)
    plt.plot(x, fx, "r--")
    plt.ylabel(None)
    plt.xlim(-4, 4)
    plt.ylim(0, 0.6)
plt.legend()
plt.show()
```

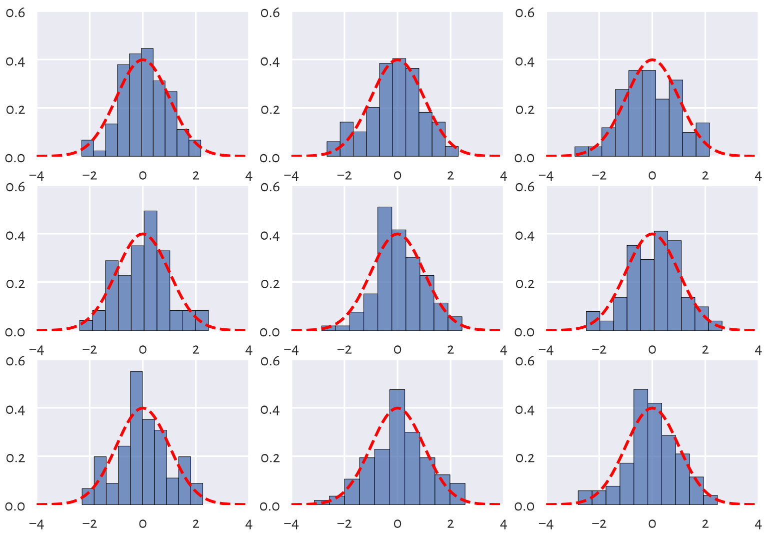


Figure7.12: All samples are normally distributed!

These is some ruggedness in the bars' sizes that a naïve observer might try to interpret as something meaningful; a competent data scientist must train their eye to ignore such impurities that are only due to random effects (but be always ready do detect those which are worth attention).

More specialist tools such as hypothesis tests enable us to tell if the deviation from the theoretical distribution is statistically significant or not. This is why, after finishing this introductory course, we should be yearning for more maths.

Exercise 7.9 *Repeat the above experiment for samples of size 10, 1000, and 10000.*

7.5 Exercises

Exercise 7.10 *Why is the notion of the mean income confusing the general public?*

Exercise 7.11 *When manually setting the seed of a random number generator makes sense?*

Exercise 7.12 *Given a log-normally distributed sample x, how can we turn it to a normally distributed one, i.e., `y=f(x)`, with **f** being... what?*

Exercise 7.13 *What is the 3σ rule for normally distributed data?*

Part III

Multidimensional Data

8

Multidimensional Numeric Data at a Glance

From the perspective of structured datasets, a vector often represents n independent measurements of the same quantitative or qualitative property, e.g., heights of n different patients, incomes in n randomly chosen households, or age category brackets of n runners (we deal with categorical data in Chapter 12).

More generally, these are all instances of a bag of n points on the real line. By far (assuming we have solved all the suggested exercises) we should have become quite fluent with the methods for processing such one-dimensional arrays.

Let's increase the level of complexity by allowing each of the n entities be described by possibly more than one feature, let's say m of them. In other words, we will be dealing with n points in an m-dimensional space, $\mathbb{R}^m$.

We can arrange all the observations in a table with n rows and m columns (just like in spreadsheets). Such an object can be expressed with **numpy** as a two-dimensional array which we will refer to as *matrices*. Thanks to matrices, we can all keep n vectors of length m together in a single object and process them all at once (or m vectors of length n, depending how we want to look at it). Very convenient.

Important: Just like vectors, matrices were designed to store data of the same type. In Chapter 11 we will cover *data frames*, which further increase the degree of complexity (and freedom) by not only allowing for mixed data types (e.g., numerical and categorical; this will enable us to perform data analysis in subgroups more easily) but also for the rows and columns be named.

Many data analysis algorithms convert data frames to matrices automatically and deal with them as such, and from the computational side, it is **numpy** which does most of the "mathematical" work. Our ambitions is to go way beyond very basic data wrangling (this is a university-level course after all), and that is why before proceeding with **pandas**, we are learning the former in detail.

8.1 Creating Matrices

8.1.1 Reading CSV Files

Tabular data are often stored and distributed in a very portable plain-text format called CSV (comma-separated values) or variants thereof.

numpy.loadtxt, which we have already been using, supports them quite well as long as they do not feature column names (comment lines are accepted, though). Unfortunately, for most CSV files the opposite is the case, and hence we suggest relying rely on a corresponding function from the **pandas** package (anticipating what's going to happen in Chapter 11):

```
body = pd.read_csv("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/nhanes_adult_female_bmx_2020.csv",
    comment="#")
body = body.to_numpy()  # data frames will be covered later
```

Note that we have converted the data frame to a **numpy** array by calling the **to_numpy** method. Here is a preview of a few first rows:

```
body[:6, :]  # 6 first rows, all columns
## array([[ 97.1, 160.2,  34.7,  40.8,  35.8, 126.1, 117.9],
##        [ 91.1, 152.7,  33.5,  33. ,  38.5, 125.5, 103.1],
##        [ 73. , 161.2,  37.4,  38. ,  31.8, 106.2,  92. ],
##        [ 61.7, 157.4,  38. ,  34.7,  29. , 101. ,  90.5],
##        [ 55.4, 154.6,  34.6,  34. ,  28.3,  92.5,  73.2],
##        [ 62. , 144.7,  32.5,  34.2,  29.8, 106.7,  84.8]])
```

This is an extended version of the National Health and Nutrition Examination Survey (NHANES[82] dataset), where we are given body measurements of adult females, in the following order:

1. weight (kg),
2. standing height (cm),
3. upper arm length (cm),
4. upper leg length (cm),
5. arm circumference (cm),

[82] https://wwwn.cdc.gov/nchs/nhanes/search/datapage.aspx

6. hip circumference (cm),
7. waist circumference (cm).

Unfortunately, **numpy** matrices do not support column naming, so these should be noted down separately elsewhere. **pandas** data frames will have that capability, but – as we have said above – from the algebraic side, they are not as convenient for the purpose of scientific computing as matrices.

What we are dealing with is still a **numpy** array:

```
type(body)
## <class 'numpy.ndarray'>
```

however, this time a two-dimensional one:

```
body.ndim
## 2
```

which means that the `shape` slot is now a tuple of length 2:

```
body.shape
## (4221, 7)
```

The above gave the number of rows and columns, respectively.

8.1.2 Enumerating Elements

numpy.array can create a two-dimensional array based on a list of lists or vector-like objects of identical lengths. Each of them will constitute a separate row of the resulting matrix.

For example:

```
np.array([
    [ 1,  2,  3,  4 ],
    [ 5,  6,  7,  8 ],
    [ 9, 10, 11, 12 ]
])
## array([[ 1,  2,  3,  4],
##        [ 5,  6,  7,  8],
##        [ 9, 10, 11, 12]])
```

gives a 3-by-4 (3×4) matrix,

```
np.array([ [1], [2], [3] ])
## array([[1],
##        [2],
##        [3]])
```

yields a 3-by-1 one (we call it a *column vector*, but it is a special matrix — we'll soon learn that shapes can make a significant difference), and

```
np.array([ [1, 2, 3, 4] ])
## array([[1, 2, 3, 4]])
```

produces a 1-by-4 array (a *row vector*).

Note: Note that an ordinary vector (a 1-dimensional array) only uses a single pair of square brackets:

```
np.array([1, 2, 3, 4])
## array([1, 2, 3, 4])
```

8.1.3 Repeating Arrays

The previously mentioned **numpy.tile** and **numpy.repeat** can also generate some nice matrices. For instance,

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

repeats a row vector rowwisely (i.e., over axis `0` – the first one).

Replicating a column vector columnwisely (i.e., over axis `1` – the second one) is possible as well:

```
np.repeat([[1], [2], [3], [4]], 3, axis=1)
## array([[1, 1, 1],
##        [2, 2, 2],
```

(continues on next page)

(continued from previous page)

```
##         [3, 3, 3],
##         [4, 4, 4]])
```

Exercise 8.1 *How to obtain matrices of the following kinds programmatically?*

$$\begin{bmatrix} 1 & 2 \\ 1 & 2 \\ 1 & 2 \\ 3 & 4 \\ 3 & 4 \\ 3 & 4 \\ 3 & 4 \end{bmatrix}, \quad \begin{bmatrix} 1 & 2 & 1 & 2 & 1 & 2 & 1 & 2 \\ 1 & 2 & 1 & 2 & 1 & 2 & 1 & 2 \\ 1 & 2 & 1 & 2 & 1 & 2 & 1 & 2 \end{bmatrix}, \quad \begin{bmatrix} 1 & 1 & 1 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 4 & 4 & 4 & 4 \end{bmatrix}.$$

8.1.4 Stacking Arrays

numpy.column_stack and **numpy.row_stack** take a tuple of array-like objects and bind them column- or rowwisely to form a new matrix:

```
np.column_stack(([10, 20], [30, 40], [50, 60]))  # a tuple of lists
## array([[10, 30, 50],
##        [20, 40, 60]])
np.row_stack(([10, 20], [30, 40], [50, 60]))
## array([[10, 20],
##        [30, 40],
##        [50, 60]])
np.column_stack((
    np.row_stack(([10, 20], [30, 40], [50, 60])),
    [70, 80, 90]
))
## array([[10, 20, 70],
##        [30, 40, 80],
##        [50, 60, 90]])
```

Exercise 8.2 *Perform similar operations using* ***numpy.append***, ***numpy.vstack***, ***numpy.hstack***, ***numpy.concatenate***, *and (*)* ***numpy.c_***.

Exercise 8.3 *Using* ***numpy.insert***, *and a new row/column at the beginning, end, and in the middle of an array. Note that this function returns a new array.*

8.1.5 Other Functions

Many built-in functions allow for generating arrays of arbitrary shapes (not only vectors). For example:

```
np.random.seed(123)
np.random.rand(3, 4)  # not: rand((3, 4))
## array([[0.69646919, 0.28613933, 0.22685145, 0.55131477],
##        [0.71946897, 0.42310646, 0.9807642 , 0.68482974],
##        [0.4809319 , 0.39211752, 0.34317802, 0.72904971]])
```

The same with **scipy**:

```
np.random.seed(123)
scipy.stats.uniform.rvs(0, 1, size=(2, 5))
## array([[0.69646919, 0.28613933, 0.22685145, 0.55131477, 0.71946897],
##        [0.42310646, 0.9807642 , 0.68482974, 0.4809319 , 0.39211752]])
```

Note that the way we specify the output shapes might differ across functions and packages, therefore – as usual – it is always best to refer to the their documentation.

Exercise 8.4 *Check out the following functions:* ***numpy.eye****,* ***numpy.diag****,* ***numpy.zeros****,* ***numpy.ones****, and* ***numpy.empty****.*

8.2 Reshaping Matrices

Let's take an example 3-by-4 matrix:

```
A = np.array([
    [ 1,  2,  3,  4 ],
    [ 5,  6,  7,  8 ],
    [ 9, 10, 11, 12 ]
])
```

Internally, a matrix is represented using a *long* flat vector where elements are stored in the row-major order (sometimes referred to as a C-style array, as opposed to Fortran-style which is used in, e.g., R):

```
A.size  # total number of elements
## 12
```

(continues on next page)

(continued from previous page)

```
A.ravel()  # the underlying array
## array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

It is the `shape` slot that is causing the 12 elements be treated if they were arranged on a 3 by 4 grid, for example in different algebraic computations and during the printing thereof. It can be modified anytime without modifying the underlying array:

```
A.shape = (4, 3)
A
## array([[ 1,  2,  3],
##        [ 4,  5,  6],
##        [ 7,  8,  9],
##        [10, 11, 12]])
```

This way, we have obtained a different *view* on the same data.

For convenience, there is also the **`reshape`** method that returns a modified version of the object it is applied on:

```
A.reshape(-1, 6)
## array([[ 1,  2,  3,  4,  5,  6],
##        [ 7,  8,  9, 10, 11, 12]])
```

Here, "-1" means that **`numpy`** must deduce by itself how many rows we actually want in the result (12 elements are supposed to be arranged in 6 columns, so the maths behind it is not rocket science).

This way, generating row or column vectors is very easy:

```
np.linspace(0, 1, 5).reshape(1, -1)
## array([[0.  , 0.25, 0.5 , 0.75, 1.  ]])
np.array([9099, 2537, 1832]).reshape(-1, 1)
## array([[9099],
##        [2537],
##        [1832]])
```

We should also note that reshaping is not the same as matrix *transpose*, which also changes the order of elements in the underlying array:

```
A  # to recall
## array([[ 1,  2,  3],
##        [ 4,  5,  6],
##        [ 7,  8,  9],
##        [10, 11, 12]])
A.T  # transpose of A
## array([[ 1,  4,  7, 10],
##        [ 2,  5,  8, 11],
##        [ 3,  6,  9, 12]])
```

We see that the rows became columns and vice versa.

Note: (*) Higher-dimensional arrays are also possible. For example,

```
np.arange(24).reshape(2, 4, 3)
## array([[[ 0,  1,  2],
##         [ 3,  4,  5],
##         [ 6,  7,  8],
##         [ 9, 10, 11]],
##
##        [[12, 13, 14],
##         [15, 16, 17],
##         [18, 19, 20],
##         [21, 22, 23]]])
```

Is an array of "depth" 2, "height" 4, and "width" 3; we can see it as two 4-by-3 matrices stacked together. Theoretically, they can be useful for representing contingency tables for products of many factors, but in our application areas we are used to sticking with long data frames instead, see Section 11.7 (more aesthetic display, better handling of sparse data).

8.3 Mathematical Notion

Here is some standalone mathematical notation that we shall be employing in the course of this course. A matrix with n rows and m columns (an n by m, $n \times m$-matrix) $\mathbf{X}$ can be written as

$$\mathbf{X} = \left[\begin{array}{cccc} x_{1,1} & x_{1,2} & \cdots & x_{1,m} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,m} \end{array} \right].$$

Mathematically, we denote this as $\mathbf{X} \in \mathbb{R}^{n\times m}$. Looking at the above, if this makes us think of how data are displayed in spreadsheets, we are correct, because the latter were inspired by the former.

We see that $x_{i,j} \in \mathbb{R}$ denotes the element in the i-th row (e.g., the i-th *observation*) and the j-th column (e.g., the j-th *feature* or *variable*), for every $i = 1,\dots,n$, $j = 1,\dots,m$.

Important: Matrices like $\mathbf{X}$ are a convenient means of representing many different kinds of data:

- n points in an m dimensional space (like n observations for which there are m measurements/features recorded, where each row describes a different object);
- m time series sampled at n points in time (e.g., m different stocks on n consecutive days);
- a single kind of measurement for data in m groups, each consisting of n subjects (e.g., heights of n males and n females); here, the order of elements in each column does not usually matter as observations are not *paired*; there is no relationship between $x_{i,j}$ and $x_{i,k}$ for $j \neq k$; a matrix is used merely as a convenient container for storing a few unrelated vectors of identical sizes; we will be dealing with a more generic case of possibly nonhomogeneous groups in Chapter 13;
- two-way contingency tables (see Section 12.2.2), where an element $x_{i,j}$ gives the number of occurrences of items at the i-th level of the first categorical variable and, at the same time, being at the j-th level of the second variable (e.g., blue-eyed *and* blonde-haired);
- graphs and other relationships between objects, e.g., $x_{i,j} = 0$ might denote that the i-th object is not connected with the j-th one and $x_{k,l} = 0.2$ that there is a weak connection between k and l;
- images, where $x_{i,j}$ represents the intensity of a colour component (e.g., red, green, blue or shades of grey or hue, saturation, brightness; compare Section 18.4) of a pixel in the $(n-i+1)$-th row and the j-th column.

In this course, we are most most often dealing with the first case, i.e., n independent points in an m dimensional space, $\mathbb{R}^m$, which we refer to as *multidimensional data*. The above body dataset is a typical example of such data.

Note: In practice, quite often more complex and less-structured data can be mapped to a tabular form. For instance, a set of audio recordings can be described by measuring overall loudness, timbre, and danceability of each song. Also, a collection of documents can be described by means of the degrees of belongingness to some automatically discovered topics (e.g., someone said that Joyce's *Ulysses* is 80% travel literature, 70% comedy, and 50% heroic fantasy, but let's not take it for granted).

8.3.1 Row and Column Vectors

Additionally, will sometimes use the following notation to emphasise that $\mathbf{X}$ consists of n rows:

$$\mathbf{X} = \left[\begin{array}{c} \mathbf{x}_{1,\cdot} \\ \mathbf{x}_{2,\cdot} \\ \vdots \\ \mathbf{x}_{n,\cdot} \end{array} \right].$$

Here, $\mathbf{x}_{i,\cdot}$ is a *row vector* of length m, i.e., a $(1 \times m)$-matrix:

$$\mathbf{x}_{i,\cdot} = \left[\begin{array}{cccc} x_{i,1} & x_{i,2} & \cdots & x_{i,m} \end{array} \right].$$

Also, recall that we are used to denoting a *vector* of length m with $\boldsymbol{x} = (x_1, \dots, x_m)$. A vector is a 1-dimensional array (not a 2-dimensional one), hence a slightly different notation in the case where ambiguity can lead to some trouble.

Important: In Chapter 9 we will see that often numpy automatically treats length-m vectors as if they were row vectors, i.e., matrices of size 1×m. For the column vector behaviour, we will need to reshape them manually.

Alternatively, we can specify the m columns

$$\mathbf{X} = \left[\begin{array}{cccc} \mathbf{x}_{\cdot,1} & \mathbf{x}_{\cdot,2} & \cdots & \mathbf{x}_{\cdot,m} \end{array} \right],$$

where $\mathbf{x}_{\cdot,j}$ is a *column vector* of length n, i.e., an $(n \times 1)$-matrix:

$$\mathbf{x}_{\cdot,j} = \left[\begin{array}{cccc} x_{1,j} & x_{2,j} & \cdots & x_{n,j} \end{array}\right]^T = \left[\begin{array}{c} x_{1,j} \\ x_{2,j} \\ \vdots \\ x_{n,j} \end{array}\right],$$

where $\cdot^T$ denotes the transpose of a given matrix (thanks to which we can save some vertical space, we don't want this book be 1000 pages long, do we?).

8.3.2 Transpose

More generally, the *transpose* of a matrix $\mathbf{X} \in \mathbb{R}^{n\times m}$ is an $(m \times n)$-matrix $\mathbf{Y}$ given by:

$$\mathbf{Y} = \mathbf{X}^T = \left[\begin{array}{cccc} x_{1,1} & x_{2,1} & \cdots & x_{m,1} \\ x_{1,2} & x_{2,2} & \cdots & x_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ x_{1,n} & x_{2,n} & \cdots & x_{m,n} \end{array}\right],$$

i.e., $y_{i,j} = x_{j,i}$.

Exercise 8.5 *Compare the display of an example matrix* `A` *and its transpose* `A.T` *above.*

8.3.3 Identity and Other Diagonal Matrices

$\mathbf{I}$ denotes the *identity matrix*, being a square $n \times n$ (with n most often clear from the context) matrix with 0s everywhere except on the main diagonal, where 1s lie.

```
np.eye(5)
## array([[1., 0., 0., 0., 0.],
##        [0., 1., 0., 0., 0.],
##        [0., 0., 1., 0., 0.],
##        [0., 0., 0., 1., 0.],
##        [0., 0., 0., 0., 1.]])
```

The identity matrix is a neutral element of matrix multiplication.

More generally, any diagonal matrix, $\mathrm{diag}(a_1, \dots, a_n)$, can be constructed from a given sequence of elements by calling:

```
np.diag([1, 2, 3, 4])
## array([[1, 0, 0, 0],
```

(continues on next page)

(continued from previous page)

```
##        [0, 2, 0, 0],
##        [0, 0, 3, 0],
##        [0, 0, 0, 4]])
```

8.4 Visualising Multidimensional Data

Let us go back to our body dataset:

```
body[:6, :]  # preview
## array([[ 97.1, 160.2,  34.7,  40.8,  35.8, 126.1, 117.9],
##        [ 91.1, 152.7,  33.5,  33. ,  38.5, 125.5, 103.1],
##        [ 73. , 161.2,  37.4,  38. ,  31.8, 106.2,  92. ],
##        [ 61.7, 157.4,  38. ,  34.7,  29. , 101. ,  90.5],
##        [ 55.4, 154.6,  34.6,  34. ,  28.3,  92.5,  73.2],
##        [ 62. , 144.7,  32.5,  34.2,  29.8, 106.7,  84.8]])
```

This is an example of tabular ("structured") data. The important property it that the elements in each row describe the same person; we can reorder all the columns at the same time (change the order of participants), but sorting a single column and leaving the others unchanged will be semantically invalid.

```
body.shape
## (4221, 7)
```

Mathematically, we consider the above as a set of 4221 points in a 7-dimensional space. Let us discuss how we can try visualising different natural *projections* thereof.

8.4.1 2D Data

A *scatterplot* can be used to visualise one variable against another one.

```
plt.scatter(body[:, 1], body[:, 3], c="#00000011")
plt.xlabel("standing height (cm)")
plt.ylabel("upper leg length (cm)")
plt.show()
```

Figure 8.1 depicts upper leg length (y axis) vs (versus; against; as a function of) standing

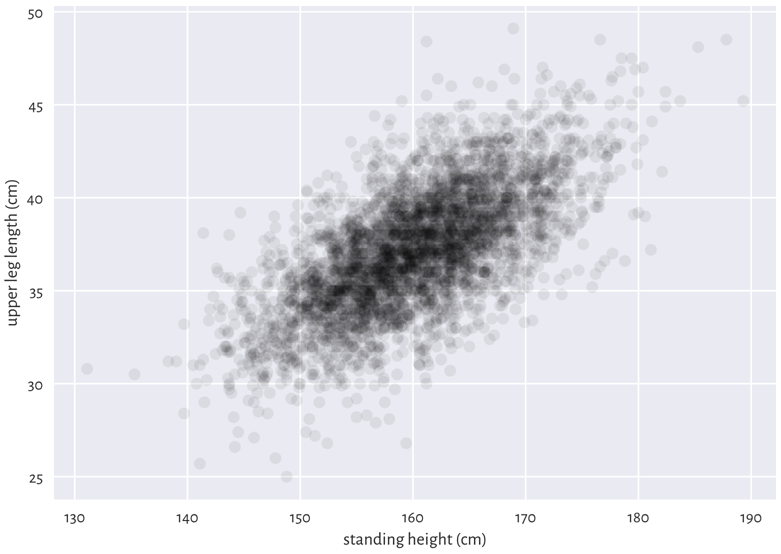


Figure8.1: An example scatterplot

height (x axis) in the form of a point cloud with (x, y) coordinates like `(body[i, 1], body[i, 3])`.

For instance, here are the exact coordinates of the point corresponding to the person of the smallest height:

```
body[np.argmin(body[:, 1]), [1, 3]]
## array([131.1,  30.8])
```

and here is the one with the greatest upper leg length:

```
body[np.argmax(body[:, 3]), [1, 3]]
## array([168.9,  49.1])
```

As the points are plentiful, normally we cannot easily see *where* the majority of them is located. However, to remedy this, we applied the simple trick of plotting the points using a semi-transparent colour. Here, the colour specifier was of the form `#rrggbbaa`, giving the intensity of the red, green, blue, and alpha (opaqueness) channel in series of two hexadecimal digits (between `00` = 0 and `ff` = 255).

Overall, the plot reveals that there is a *general tendency* of small heights and small upper leg lengths to occur frequently together. The same with larger pairs. In Chapter 10 we will explore some measures of correlation that enable us to quantify the degree of association between variable pairs.

8.4.2 3D Data and Beyond

If we have more than 2 variables to visualise, we might be tempted to use, e.g., a 3-dimensional scatterplot like the one in Figure 8.2.

```
fig = plt.figure()
ax = fig.add_subplot(projection="3d")
ax.scatter(body[:, 1], body[:, 3], body[:, 0], color="#00000011")
ax.view_init(elev=30, azim=20, vertical_axis="y")
ax.set_xlabel("standing height (cm)")
ax.set_ylabel("upper leg length (cm)")
ax.set_zlabel("weight (kg)")
plt.show()
```

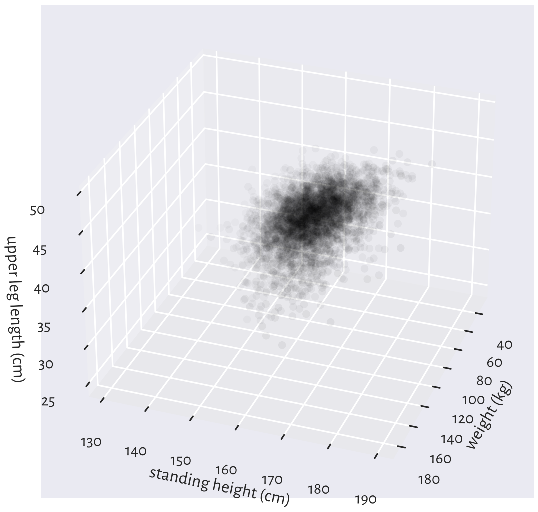


Figure8.2: A three-dimensional scatterplot reveals almost nothing

However, infrequently will such a 3D plot provide us with readable results: we are projecting a three-dimensional reality onto a two-dimensional screen or page. Some information must inherently be lost. Also, what we see is relative to the position of the virtual camera.

Exercise 8.6 *(*) Try finding an* interesting *elevation and azimuth angle by playing with the arguments passed to the* **`mpl_toolkits.mplot3d.axes3d.Axes3D.view_init`** *function. Also, depict arm circumference, hip circumference, and weight.*

Note: (*) Sometimes there might be facilities available to create an interactive scatterplot (e.g., running the above from the Python's console actually enables this), where the virtual camera can be freely repositioned. This can give some insight into our data. Also, there are means of creating animated sequences, where we can fly over the data scene. Some people find it cool, others find in annoying, but the biggest problem therewith is that they cannot be included in printed material. However, if we are only targeting the display for the Web (this includes mobile devices), we can try some Python libraries[83] that output HTML+CSS+JavaScript code to be rendered by a browser engine.

Example 8.7 *Instead of drawing a 3D plot, it might be better to play with different marker colours (or sometimes sizes: think of them as bubbles). A suitable colour map[84], can be used to distinguish between low and high values of an additional variable, as in Figure 8.3.*

```
from matplotlib import cm
plt.scatter(
    body[:, 4],    # x
    body[:, 5],    # y
    c=body[:, 0], # "z" - colours
    cmap=cm.get_cmap("copper"),  # colour map
    alpha=0.5  # opaqueness level between 0 and 1
)
plt.xlabel("arm circumference (cm)")
plt.ylabel("hip circumference (cm)")
plt.axis("equal")
plt.rcParams["axes.grid"] = False
cbar = plt.colorbar()
plt.rcParams["axes.grid"] = True
cbar.set_label("weight (kg)")
plt.show()
```

[83] https://wiki.python.org/moin/NumericAndScientific/Plotting
[84] https://matplotlib.org/stable/tutorials/colors/colormaps.html

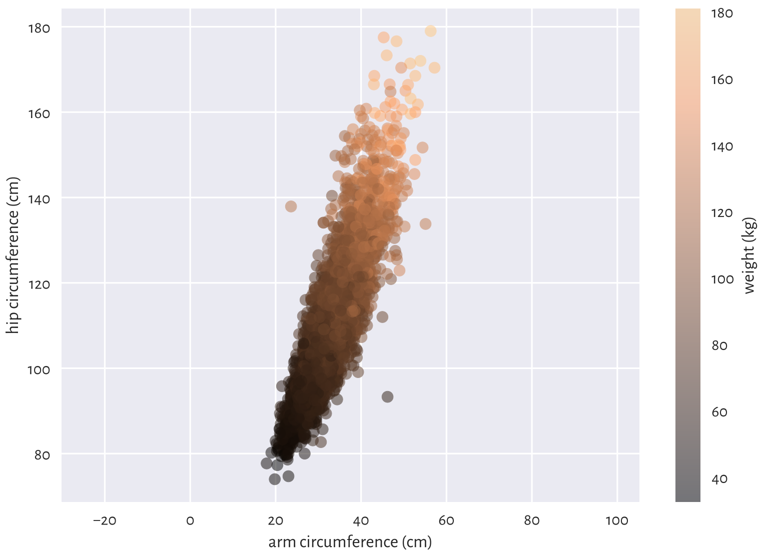


Figure8.3: A two-dimensional scatter plot displaying 3 variables

We can see some tendency for the weight be greater as both the arm and the hip circumferences increase.

Exercise 8.8 *Play around with different colour pallettes*[85].

However, be wary that ca. every 1 in 12 men (8%) and 1 in 200 women (0.5%) has colour vision deficiencies, especially in the red-green or blue-yellow spectrum, thence some diverging colour maps might be worse than others.

A piece of paper is 2-dimensional. We only have height and width. The world around us is 3-dimensional, we thus also understand the notion of depth. As far as the case of more-dimensional data is concerned, well, suffice it to say that we are 3-dimensional creatures and any attempts towards visualising them will simply not work, don't even trip.

Luckily, this is where mathematics comes to our rescue. With some more knowledge

[85] https://matplotlib.org/stable/tutorials/colors/colormaps.html

and intuitions, and this book lets us gain them, it will be as easy as imagining a generic m-dimensional space, and then assuming that, say, m=7 or 42.

Pardon yours truly an old joke.

But, really, jokes aside, this is exactly why data science relies on automated methods for knowledge/pattern discovery – so that we are able to identify, describe, and analyse the structures that might be present in the data, but cannot be perceived with our imperfect senses.

Note: Linear and nonlinear dimensionality reduction techniques can be applied to visualise some aspects of high-dimensional data in form of 2D (or 3D) plots. In particular, the principal component analysis (PCA) finds an *interesting* angle at which to look at the data (so that most variance is along the first, and then second, etc., axis).

8.4.3 Scatterplot Matrix (Pairplot)

As a countermeasure, we may try depicting all (or most – ones we deem interesting) pairs of variables in the same drawing in form of a scatterplot matrix, see Figure 8.4.

```
sns.pairplot(data=pd.DataFrame(
    body[:, [0, 1, 4, 5]],
    columns=[
        "weight (kg)", "standing height (cm)",
        "arm circumference (cm)", "hip circumference (cm)"
    ]
))
# plt.show()  # not needed :/
```

As depicting a variable against itself is uninteresting (exercise: what would that be?), we have included histograms on the main diagonal to see how the one-dimensional projections are distributed (the *marginal distributions*).

In Chapter 10 we will be interested in describing and even modelling the possible relationships between variables. A scatterplot matrix can be a good tool for identifying interesting combinations of columns in our datasets.

Exercise 8.9 *Draw a scatterplot matrix for the fcps_chainlink*[86] *dataset. Note that we are only observing different 2-dimensional projections of this more complex domain – only those that are along the main axes. Create a 3D scatterplot to reveal the true shapes.*

[86] https://raw.githubusercontent.com/gagolews/teaching_data/master/clustering/fcps_chainlink.csv

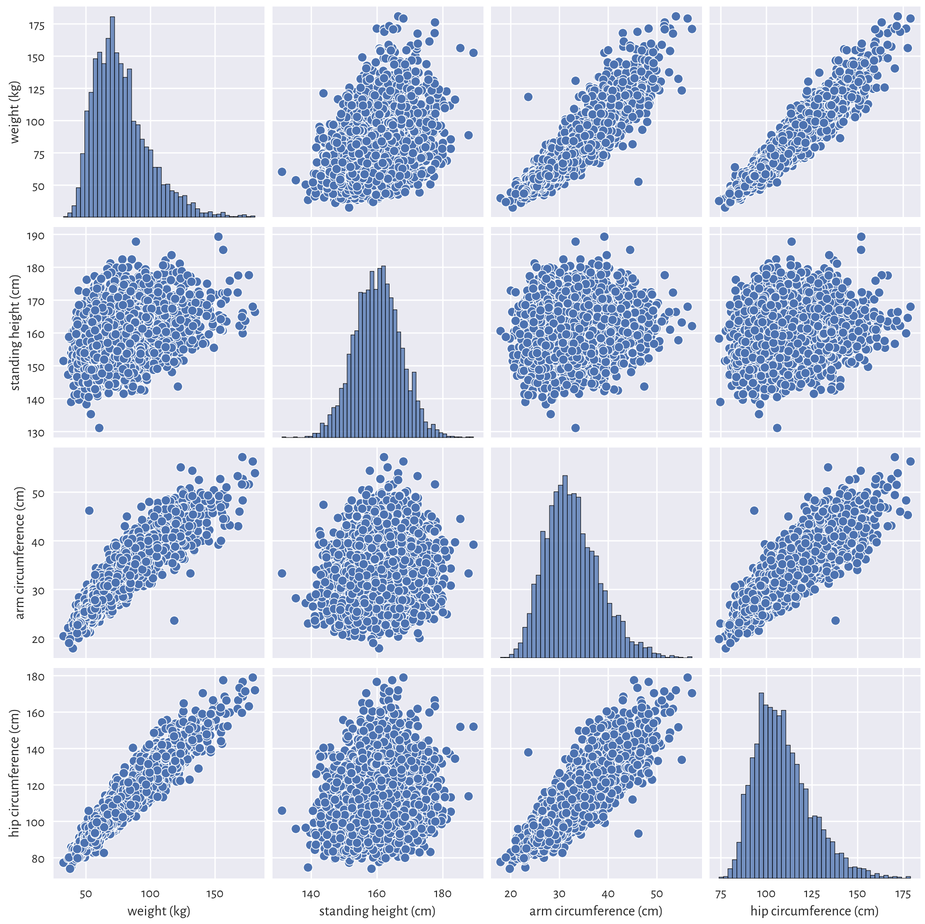


Figure8.4: Scatterplot matrix for selected columns in the `body` dataset

Exercise 8.10 *(*) Use* **`matplotlib.pyplot.subplot`** *and other functions we have learned in the previous part to create a scatterplot matrix manually. Draw weight, arm circumference, and hip circumference on a logarithmic scale.*

8.5 Exercises

Exercise 8.11 *What is the difference between `[1, 2, 3]`, `[[1, 2, 3]]` and `[[1], [2], [3]]` in the context of array creation?*

Exercise 8.12 *If `A` is a matrix with 5 rows and 6 columns, what is the difference between `A.reshape(6, 5)` and `A.T`?*

Exercise 8.13 *If `A` is a matrix with 5 rows and 6 columns, what is the meaning of: `A.reshape(-1)`, `A.reshape(3, -1)`, `A.reshape(-1, 3)`, `A.reshape(-1, -1)`, `A.shape = (3, 10)`, and `A.shape = (-1, 3)`?*

Exercise 8.14 *List some methods to add a new row to an existing matrix.*

Exercise 8.15 *How to visualise 3-dimensional data?*

9

Transforming, Aggregating, and Filtering Multidimensional Data

Below we will study how the vector operations that we discussed in Chapter 6 can be extended to matrices. In many cases, we'll end up applying the same transform either on every matrix element separately, or on each row or column. They are all a brilliant example of the *write less, do more* principle in practice.

9.1 Vectorised Mathematical Functions

Applying vectorised functions such as **numpy.round**, **numpy.log**, and **numpy.exp** returns an array of the same shape, with all elements transformed accordingly.

```
A = np.array([
    [0.2, 0.6, 0.4, 0.4],
    [0.0, 0.2, 0.4, 0.7],
    [0.8, 0.8, 0.2, 0.1]
])  # example matrix that we'll be using below
```

For example:

```
np.square(A)
## array([[0.04, 0.36, 0.16, 0.16],
##        [0.  , 0.04, 0.16, 0.49],
##        [0.64, 0.64, 0.04, 0.01]])
```

takes the square of every element.

More generally, we will be denoting such operations with:

$$f(\mathbf{X}) = \left[\begin{array}{cccc} f(x_{1,1}) & f(x_{1,2}) & \cdots & f(x_{1,m}) \\ f(x_{2,1}) & f(x_{2,2}) & \cdots & f(x_{2,m}) \\ \vdots & \vdots & \ddots & \vdots \\ f(x_{n,1}) & f(x_{n,2}) & \cdots & f(x_{n,m}) \end{array}\right].$$

9.2 Aggregation Functions (Componentwise)

Aggregation functions (e.g., **numpy.mean**, **numpy.quantile**) can summarise:

- all data into a single number (`axis=None`, being the default),
- data in each row (`axis=1`),
- data in each column (`axis=0`).

Here are the examples corresponding to the above cases:

```
np.mean(A)
## 0.39999999999999997
np.mean(A, axis=1)
## array([0.4  , 0.325, 0.475])
np.mean(A, axis=0)
## array([0.33333333, 0.53333333, 0.33333333, 0.4       ])
```

Important: Let us repeat, `axis=1` does not mean that we get the column means (even though columns constitute the 2nd axis, and we count starting at 0). It denotes the axis *along* which the matrix is sliced. Sadly, even yours truly sometimes does not get it right.

Exercise 9.1 *Given the* `nhanes_adult_female_bmx_2020`[87] *dataset, compute the mean, standard deviation, minimum, and maximum of each body measurement.*

[87] https://raw.githubusercontent.com/gagolews/teaching_data/master/marek/nhanes_adult_female_bmx_2020.csv

9.3 Arithmetic, Logical, and Comparison Operations

Recall that for vectors, binary operators such as `` `+` ``, `` `*` ``, `` `==` ``, `` `<=` ``, and `` `&` `` and similar elementwise functions (e.g., `numpy.minimum`) can be applied if both inputs are of the same length, for example:

```
np.array([1, 10, 100, 1000]) * np.array([7, -6, 2, 8])
## array([   7,  -60,  200, 8000])
```

Alternatively, one input can be a scalar:

```
np.array([1, 10, 100, 1000]) * -3
## array([   -3,    -30,   -300, -3000])
```

More generally, a set of rules referred in the `numpy` manual to as *broadcasting*[88] describes how this package handles arrays of different shapes.

Important: Generally, for two matrices, their column/row numbers much match or be equal to 1. Also, if one operand is a 1-dimensional array, it will be promoted to a row vector.

Let us explore all the possible cases.

9.3.1 Matrix vs Scalar

If one operand is a scalar, then it is going to be propagated over all matrix elements, for example:

```
A = np.array([
    [0.2, 0.6, 0.4, 0.4],
    [0.0, 0.2, 0.4, 0.7],
    [0.8, 0.8, 0.2, 0.1]
])  # example matrix
```

And now:

[88] https://numpy.org/devdocs/user/basics.broadcasting.html

```
(-1)*A
## array([[-0.2, -0.6, -0.4, -0.4],
##        [-0. , -0.2, -0.4, -0.7],
##        [-0.8, -0.8, -0.2, -0.1]])
```

changes the sign of every element, which is, mathematically, an instance of multiplying a matrix $\mathbf{X}$ by a scalar c

$$c\mathbf{X} = \left[\begin{array}{cccc} cx_{1,1} & cx_{1,2} & \cdots & cx_{1,m} \\ cx_{2,1} & cx_{2,2} & \cdots & cx_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ cx_{n,1} & cx_{n,2} & \cdots & cx_{n,m} \end{array}\right].$$

Furthermore:

```
A**2
## array([[0.04, 0.36, 0.16, 0.16],
##        [0.  , 0.04, 0.16, 0.49],
##        [0.64, 0.64, 0.04, 0.01]])
```

takes the square of each element (this is not the same as matrix-multiply by itself which we cover in the matrix algebra section).

Also:

```
A >= 0.25
## array([[False,  True,  True,  True],
##        [False, False,  True,  True],
##        [ True,  True, False, False]])
```

compares each element to 0.25.

9.3.2 Matrix vs Matrix

For two matrices of identical sizes, we will be acting on the corresponding elements:

```
B = np.tri(A.shape[0], A.shape[1])  # just an example
B  # a lower triangular 0-1 matrix
## array([[1., 0., 0., 0.],
##        [1., 1., 0., 0.],
##        [1., 1., 1., 0.]])
```

And now:

```
A * B
## array([[0.2, 0. , 0. , 0. ],
##        [0. , 0.2, 0. , 0. ],
##        [0.8, 0.8, 0.2, 0. ]])
```

multiplies each $a_{i,j}$ by the corresponding $b_{i,j}$.

This extends on the idea from algebra that given $\mathbf{A}$, $\mathbf{B}$ with n rows and m columns each, the result of $+$ (or $-$) would be for instance:

$$\mathbf{A}+\mathbf{B} = \left[\begin{array}{cccc} a_{1,1}+b_{1,1} & a_{1,2}+b_{1,2} & \cdots & a_{1,m}+b_{1,m} \\ a_{2,1}+b_{2,1} & a_{2,2}+b_{1,2} & \cdots & a_{2,m}+b_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1}+b_{n,1} & a_{n,2}+b_{n,2} & \cdots & a_{n,m}+b_{n,m} \end{array}\right].$$

Thanks to the matrix-matrix and matrix-scalar operations we can perform various tests on an per-element basis, e.g.,

```
(A >= 0.25) & (A <= 0.75)  # logical matrix & logical matrix
## array([[False,  True,  True,  True],
##        [False, False,  True,  True],
##        [False, False, False, False]])
```

9.3.3 Matrix vs Any Vector

An $n \times m$ matrix can also be combined with an $n \times 1$ column vector:

```
A * np.array([1, 10, 100]).reshape(-1, 1)
## array([[ 0.2,   0.6,   0.4,   0.4],
##        [ 0. ,   2. ,   4. ,   7. ],
##        [80. ,  80. ,  20. ,  10. ]])
```

The above propagated the column vector over all columns (left to right).

Similarly, combining with an $1 \times m$ row vector:

```
A + np.array([1, 2, 3, 4]).reshape(1, -1)
## array([[1.2, 2.6, 3.4, 4.4],
##        [1. , 2.2, 3.4, 4.7],
##        [1.8, 2.8, 3.2, 4.1]])
```

recycles the row vector over all rows (top to bottom).

If one operand is a 1-dimensional array or a list of length m, it will be treated as a row vector.

```
np.round(A - np.mean(A, axis=0), 3)
## array([[-0.133,  0.067,  0.067, -0.   ],
##        [-0.333, -0.333,  0.067,  0.3  ],
##        [ 0.467,  0.267, -0.133, -0.3  ]])
```

This resulted in each column mean's being 0. An explicit `.reshape(1, -1)` was not necessary.

Mathematically, although it is not necessarily a standard notation, we will allow adding and subtracting row vectors from matrices of compatible sizes:

$$\mathbf{X}+\mathbf{t}=\left[\begin{array}{cccc} x_{1,1}+t_1 & x_{1,2}+t_2 & \dots & x_{1,m}+t_m \\ x_{2,1}+t_1 & x_{2,2}+t_2 & \dots & x_{2,m}+t_m \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1}+t_1 & x_{n,2}+t_2 & \dots & x_{n,m}+t_m \end{array}\right].$$

This corresponds to shifting (translating) every row in the matrix.

However, subtracting the row means already requires some extra labour:

```
A - np.mean(A, axis=1).reshape(-1, 1)
## array([[-0.2  ,  0.2  ,  0.   ,  0.   ],
##        [-0.325, -0.125,  0.075,  0.375],
##        [ 0.325,  0.325, -0.275, -0.375]])
```

Note that `A - np.mean(A, axis=1)` would raise an exception.

Exercise 9.2 *Standardise, normalise, and min-max scale each column in the* `nhanes_adult_female_bm` `csv`[89] *dataset using a single line of code.*

9.3.4 Row Vector vs Column Vector (*)

As a bonus, let us quickly mention that a row vector combined with a column vector results in an operation's being performed on each *combination* of *all* pairs of elements in the two arrays (i.e., the cross-product; not just the *corresponding* pairs).

```
np.arange(1, 8).reshape(1, -1) * np.array([1, 10, 100]).reshape(-1, 1)
## array([[  1,   2,   3,   4,   5,   6,   7],
```

(continues on next page)

[89] https://raw.githubusercontent.com/gagolews/teaching_data/master/marek/nhanes_adult_female_bmx_2020.csv

(continued from previous page)

```
##        [ 10,  20,  30,  40,  50,  60,  70],
##        [100, 200, 300, 400, 500, 600, 700]])
```

Exercise 9.3 *(**) Check out that* **`numpy.where`** *relies on similar shape broadcasting rules as the binary operators we have discussed here, however, not with regards to all the 3 arguments.*

9.4 Other Row and Column Transforms (*)

Some functions that we have already discussed in the previous part of this course are equipped with the `axis` argument, which allows to process each row or column independently, for example:

```
np.sort(A, axis=1)
## array([[0.2, 0.4, 0.4, 0.6],
##        [0. , 0.2, 0.4, 0.7],
##        [0.1, 0.2, 0.8, 0.8]])
```

sorts every row (separately). Moreover:

```
scipy.stats.rankdata(A, axis=0)
## array([[2. , 2. , 2.5, 2. ],
##        [1. , 1. , 2.5, 3. ],
##        [3. , 3. , 1. , 1. ]])
```

computes the ranks of elements in each column.

Also note that a few functions have the default argument `axis=-1`, which means that they are applied along the last (i.e., columns in the matrix case) axis:

```
np.diff(A)  # axis=1 here
## array([[ 0.4, -0.2,  0. ],
##        [ 0.2,  0.2,  0.3],
##        [ 0. , -0.6, -0.1]])
```

However, the aforementioned **`numpy.mean`** is amongst the many exceptions to this rule.

Compare the above with:

```
np.diff(A, axis=0)
## array([[-0.2, -0.4,  0. ,  0.3],
##        [ 0.8,  0.6, -0.2, -0.6]])
```

which gives the iterated differences for each column separately (along the rows).

If a function (built-in or custom) in not equipped with the `axis` argument and – instead – it was designed to work with individual vectors, we can propagate it over all the rows or columns by calling **`numpy.apply_along_axis`**.

For instance, here is another (have we solved the suggested exercise?) way to compute the column z-scores:

```
def standardise(x):
    return (x-np.mean(x))/np.std(x)

np.round(np.apply_along_axis(standardise, 0, A), 2)
## array([[-0.39,  0.27,  0.71, -0.  ],
##        [-0.98, -1.34,  0.71,  1.22],
##        [ 1.37,  1.07, -1.41, -1.22]])
```

9.5 Matrix Indexing

Recall that for 1-dimensional arrays we have 4 possible choices of an indexer (i.e., where performing filtering like `x[i]`):

- scalar (extracts a single element),
- slice (selects a regular subsequence, e.g., every 2nd element or the first 6 items; returns a *view* on existing data – it does not make an independent copy of the subsetted elements),
- integer vector (selects elements at given indices),
- logical vector (select elements that correspond to `True` in the indexer).

Matrices are two-dimensional arrays, and hence subsetting thereof will require two indexes: we will thus be writing `A[i, j]` to select rows given by `i` and columns given by `j`.

Both `i` and `j` can be one of the four above types, so we at least 10 different cases to consider (skipping the symmetric ones).

Important: Generally:

- each scalar indexer reduces the dimensionality of the subsetted object by 1;
- slice-slice and slice-scalar indexing returns a view on the existing array, so we need to be careful when modifying the resulting object;
- usually, indexing returns a submatrix (subblock), which is a combination of elements at given rows and columns, but two flat-vector indexers are vectorised elementwisely instead;
- indexing with two integer or logical vectors at the same time should be avoided.

Let us look at all the possible scenarios in greater detail.

9.5.1 Slice-Based Indexing

Our favourite example matrix again:

```
A = np.array([
    [0.2, 0.6, 0.4, 0.4],
    [0.0, 0.2, 0.4, 0.7],
    [0.8, 0.8, 0.2, 0.1]
])
```

Indexing based on two slices selects a submatrix:

```
A[::2, 3:]
## array([[0.4],
##        [0.1]])
```

gives every second row and skips first three columns. Note that the result is still a matrix.

An empty slice selects all elements on the corresponding axis, therefore:

```
A[:, ::-1]  # all rows, reversed columns
## array([[0.4, 0.4, 0.6, 0.2],
##        [0.7, 0.4, 0.2, 0. ],
##        [0.1, 0.2, 0.8, 0.8]])
```

simply reverses the order of columns.

9.5.2 Scalar-Based Indexing

Indexing by a scalar selects a given row or column, reducing the dimensionality of the output object.

```
A[1, :]
## array([0. , 0.2, 0.4, 0.7])
```

selects the 2nd row and gives a flat vector.

```
A[0, -1]
## 0.4
```

yields the element in the first row and last column.

However, we can always use the **reshape** method to convert the resulting object back to a matrix.

9.5.3 Mixed Boolean/Integer Vector and Scalar/Slice Indexers

A logical and integer vector-like object can also be used for element selection. If the other indexer is a slice or a scalar, the result is quite predictable, for instance:

```
A[ [0, -1, 0], ::-1 ]
## array([[0.4, 0.4, 0.6, 0.2],
##        [0.1, 0.2, 0.8, 0.8],
##        [0.4, 0.4, 0.6, 0.2]])
```

selects the first, the last, and the first row again and reverses the order of columns.

```
A[ A[:, 0] > 0.1, : ]
## array([[0.2, 0.6, 0.4, 0.4],
##        [0.8, 0.8, 0.2, 0.1]])
```

selects rows such that in the first column the values are greater than 0.1.

```
A[np.mean(A, axis=1) > 0.35, : ]
## array([[0.2, 0.6, 0.4, 0.4],
##        [0.8, 0.8, 0.2, 0.1]])
```

selects the rows whose mean is greater than 0.35.

```
A[np.argsort(A[:, 0]), : ]
## array([[0. , 0.2, 0.4, 0.7],
##        [0.2, 0.6, 0.4, 0.4],
##        [0.8, 0.8, 0.2, 0.1]])
```

orders the matrix with respect to the values in the first row (all rows permuted in the same way, together).

Exercise 9.4 *In the* `nhanes_adult_female_bmx_2020`[90] *dataset, select all the participants whose heights is within their mean ± 2 standard deviations.*

9.5.4 Two Vectors as Indexers (*)

With two vectors (logical or integer) things are a tad more horrible, as in this case not only some form *shape broadcasting* comes into play but also all the headache-inducing exceptions listed in the perhaps not the most clearly written Advanced Indexing[91] section of the **numpy** manual. Cheer up, however, things in **pandas** are much worse (see Section 11.5).

For the sake of our maintaining sanity, in practice it is best to stick only to the scenarios below and be extra careful when using two vector indexers.

For two flat integer indexers, we pick elementwisely:

```
A[ [0, -1, 0, 2, 0], [1, 2, 0, 2, 1] ]
## array([0.6, 0.2, 0.2, 0.2, 0.6])
```

yields `A[0, 1]`, `A[-1, 2]`, `A[0, 0]`, `A[2, 2]`, and `A[0, 1]`.

In order to select a submatrix using integer indexes, it is best to make sure that the first indexer is a column vector, and the second is a row vector (or some object like the two said ones, e.g., compatible lists of lists). Further, if indexing involves logical vectors, it is best to convert them to integer ones first (e.g., by calling **numpy.where**).

The above transformations can be done automatically via the **numpy.ix_** function, which is always the safest choice:

```
A[ np.ix_(np.mean(A, axis=1) > 0.35, [0, 2, 3, 0]) ]
## array([[0.2, 0.4, 0.4, 0.2],
##        [0.8, 0.2, 0.1, 0.8]])
```

[90] https://raw.githubusercontent.com/gagolews/teaching_data/master/marek/nhanes_adult_female_bmx_2020.csv

[91] https://numpy.org/doc/stable/user/basics.indexing.html

Alternatively, we can always apply indexing twice instead (which we will anyway be forced to do in **pandas** whenever selecting rows by number and columns by names is required).

```
A[np.mean(A, axis=1) > 0.45, :][:, [0, 2, 3, 0]]
## array([[0.8, 0.2, 0.1, 0.8]])
```

9.5.5 Views on Existing Arrays (*)

Only indexing involving two slices or a slice and a scalar returns a view[92] on an existing array.

For example:

```
B = A[:, ::2]
B
## array([[0.2, 0.4],
##        [0. , 0.4],
##        [0.8, 0.2]])
```

Now `B` and `A` share memory. Therefore, by modifying `B` in-place, e.g.,

```
B *= -1
```

the changes will be visible in `A` as well:

```
A
## array([[-0.2,  0.6, -0.4,  0.4],
##        [-0. ,  0.2, -0.4,  0.7],
##        [-0.8,  0.8, -0.2,  0.1]])
```

This is time and memory efficient, but might lead to some unexpected results if we are not focused enough. We have been warned.

9.5.6 Adding and Modifying Rows or Columns

With slice/scalar-based indexers, rows/columns/individual elements can be replaced by new content in a natural way:

[92] https://numpy.org/devdocs/user/basics.copies.html

```
A[:, 0] = A[:, 0]**2
```

With **numpy** arrays, however, brand new rows or columns cannot be added using the index operator. Instead, the whole array needs to be created from scratch using, for example, one of the functions discussed in Section 8.1.4.

Here is an example where we add a new column to the A matrix being a function of the first column:

```
A = np.column_stack((A, np.sqrt(A[:, 0])))
A
## array([[ 0.04,  0.6 , -0.4 ,  0.4 ,  0.2 ],
##        [ 0.  ,  0.2 , -0.4 ,  0.7 ,  0.  ],
##        [ 0.64,  0.8 , -0.2 ,  0.1 ,  0.8 ]])
```

9.6 Exercises

Exercise 9.5 *Does* ***numpy.mean(A, axis=0)*** *compute rowwise or columnwise means?*

Exercise 9.6 *How does shape broadcasting work? List the most common pairs of shape cases when performing arithmetic operations.*

Exercise 9.7 *What are the possible matrix indexing schemes and how do they behave?*

Exercise 9.8 *Which kinds of matrix indexers return a view on an existing array?*

Exercise 9.9 *(*) How to select a submatrix comprised of the first and the last row and the first and the last column?*

10

Exploring Relationships Between Variables

Note: This part is going to be significantly extended and more chapters will follow in the future.

In particular, we will introduce some basic methods related to:

- matrix algebra
- basic principles of numerical optimisation,
- aggregation of multidimensional data, detecting outliers,
- regression analysis,
- dimensionality reduction,
- multivariate density estimation,
- generating data from multivariate distributions.

Consider the National Health and Nutrition Examination Survey (NHANES study) excerpt again:

```
body = pd.read_csv("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/nhanes_adult_female_bmx_2020.csv",
    comment="#")
body = body.to_numpy()  # data frames will be covered later
body.shape
## (4221, 7)
body[:6, :]  # 6 first rows, all columns
## array([[ 97.1, 160.2,  34.7,  40.8,  35.8, 126.1, 117.9],
##        [ 91.1, 152.7,  33.5,  33. ,  38.5, 125.5, 103.1],
##        [ 73. , 161.2,  37.4,  38. ,  31.8, 106.2,  92. ],
##        [ 61.7, 157.4,  38. ,  34.7,  29. , 101. ,  90.5],
```

(continues on next page)

(continued from previous page)

```
##        [ 55.4, 154.6,  34.6,  34. ,  28.3,  92.5,  73.2],
##        [ 62. , 144.7,  32.5,  34.2,  29.8, 106.7,  84.8]])
```

We thus have *n*=4221 participants and 7 different features describing them, in this order:

1. weight (kg),
2. standing height (cm),
3. upper arm length (cm),
4. upper leg length (cm),
5. arm circumference (cm),
6. hip circumference (cm),
7. waist circumference (cm).

We expect the data in columns to be *related* to each other (e.g., a taller person *usually tends to* weight more). This is why in this chapter we are interested in quantifying the degree of association between the variables.

10.1 Measuring Correlation

Scatterplots let us identify some simple patterns or structure in data. From Figure 8.4, we can note that higher hip circumferences *tend to* occur more often together with higher arm circumferences and that the latter does not really tell us anything about height.

Let's explore some basic means for measuring (expressing as a single number) the degree of association between a set of pairs of points.

10.1.1 Pearson's Linear Correlation Coefficient

First, the Pearson's *linear correlation* coefficient:

$$r(\boldsymbol{x},\boldsymbol{y}) = \frac{1}{n}\sum_{i=1}^{n} \frac{x_i-\bar{x}}{s_x}\,\frac{y_i-\bar{y}}{s_y},$$

with s_x, s_y denoting the standard deviations and $\bar{x},\bar{y}$ being the means of $\boldsymbol{x} = (x_1,\dots,x_n)$ and $\boldsymbol{y} = (y_1,\dots,y_n)$, respectively.

Note: Look carefully: we are computing pairwise products of standardised versions of the two vectors. It is a normalised measure of how they *vary* together (co-variance).

Here is how we can compute it manually:

```
x = body[:, 4]  # arm circumference
y = body[:, 5]  # hip circumference
x_std = (x-np.mean(x))/np.std(x)
y_std = (y-np.mean(y))/np.std(y)
np.sum(x_std*y_std)/len(x)
## 0.8680627457873239
```

And here is a built-in function that implements the same formula:

```
scipy.stats.pearsonr(x, y)[0]
## 0.8680627457873241
```

To get more insight, we'll illustrate some interesting cases using the following function that draws a scatter plot and prints out Pearson's r (and Spearman's ρ which we discuss below – let's ignore it by then):

```
def plot_corr(x, y):
    r = scipy.stats.pearsonr(x, y)[0]
    ρ = scipy.stats.spearmanr(x, y)[0]
    plt.scatter(x, y, label=f"r = {r:.3}\nρ = {ρ:.3}")
    plt.legend()
    pass
```

Here are the basic properties of Pearson's r:

1. $r(\boldsymbol{x},\boldsymbol{y}) = r(\boldsymbol{y},\boldsymbol{x})$ (symmetric);
2. $|r(\boldsymbol{x},\boldsymbol{y})| \le 1$ (bounded from below by -1 and from above by 1);
3. $r(\boldsymbol{x},\boldsymbol{y}) = 1$ if and only if $\boldsymbol{y} = a\boldsymbol{x} + b$ for some $a > 0$ and b, (reaches the maximum when one variable is an increasing linear function of the other one);
4. $r(\boldsymbol{x},-\boldsymbol{y}) = -r(\boldsymbol{x},\boldsymbol{y})$ (negative scaling (reflection) of one variable changes the sign of the coefficient);
5. $r(\boldsymbol{x},a\boldsymbol{y}+b) = r(\boldsymbol{x},\boldsymbol{y})$ for any $a > 0$ and b (invariant to translation and scaling of inputs that does not change the sign of elements).

Note that the above implies that $r(\boldsymbol{x},\boldsymbol{y}) = -1$ if and only if $\boldsymbol{y} = a\boldsymbol{x}+b$ for some $a < 0$ and b (reaches the minimum when one variable is a decreasing linear function of the other one) Furthermore, a variable is trivially perfectly correlated with itself $r(\boldsymbol{x},\boldsymbol{x}) = 1$.

Hence, we get perfect *linear correlation* (-1 or 1) when one variable is a scaled and shifted version (linear function) of the other variable, see Figure 10.1.

```
np.random.seed(123)
x = np.random.rand(100)
plt.subplot(121)
plot_corr(x, -0.5*x+3)  # negative slope
plt.axis("equal")
plt.subplot(122)
plot_corr(x, 3*x+10)  # positive slope
plt.axis("equal")
plt.show()
```

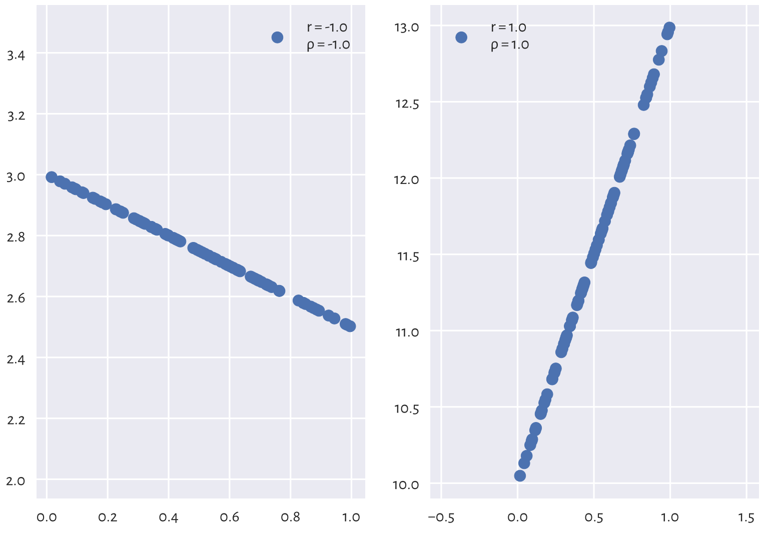


Figure10.1: Perfect linear correlation (negative and positive)

Note that negative correlation means that when one variable increases, the other one decreases (like: a car's braking distance vs velocity).

If two variables are *more or less* linear functions of themselves, the correlations will be close to -1 or 1, with the degree of association diminishing as the linear relationship becomes less and less present, see Figure 10.2.

```
x = np.random.rand(100)
y = 0.5*x
e = np.random.randn(len(x))  # random white noise (of mean 0)
plt.subplot(221)
plot_corr(x, y)
plt.subplot(222)
plot_corr(x, y+0.05*e)  # add some noise
plt.subplot(223)
plot_corr(x, y+0.1*e)   # more noise
plt.subplot(224)
plot_corr(x, y+0.25*e)  # even more noise
plt.show()
```

Note again that the arm and hip circumferences enjoy quite high positive degree of linear correlation.

Exercise 10.1 *Draw a series of similar plots but for the case of negatively correlated point pairs.*

Important: As a rule of thumb, linear correlation degree of 0.9 or greater (or -0.9 or smaller) is quite decent. Between -0.8 and 0.8 we probably shouldn't be talking about two variables being linearly correlated at all. Some textbooks are more lenient, but we have higher standards. In particular, it is not uncommon in social sciences to consider 0.6 a decent degree of correlation, but this is like building on sand. If a dataset at hand does not provide us with strong evidence, it is our ethical duty to refrain ourselves from publishing unjustified statements.

We should stress that correlation close to 0 does not necessarily mean that two variables are not related to each other, although definitely for two independent variables we expect the correlation coefficient be approximately equal to 0. It is a *linear* correlation coefficient, so we are only quantifying these types of relationships. See Figure 10.3 for an illustration of this fact.

```
plt.subplot(221)
plot_corr(x, np.random.rand(100))  # independent (not correlated)
```

(continues on next page)

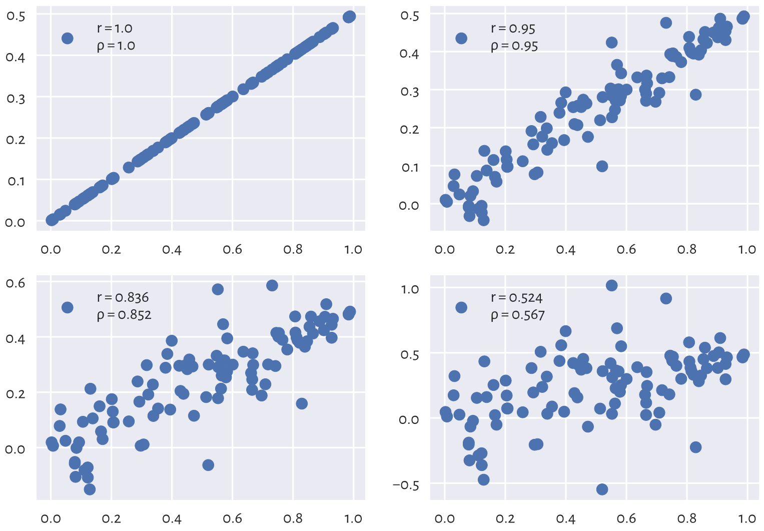


Figure10.2: Linear correlation coefficients for data with different amout of noise

(continued from previous page)

```
plt.subplot(222)
plot_corr(x, (2*x-1)**2-1)  # quadratic dependence
plt.subplot(223)
plot_corr(x, np.abs(2*x-1))  # another form of dependence
plt.subplot(224)
plot_corr(x, 0.25*np.sin(8*np.pi*x))  # another
plt.show()
```

What is more, sometimes we can detect *false* correlations – when data are functionally dependent, the relationship is not linear, but it kind of looks like linear. Refer to Figure 10.4 for some examples.

```
plt.subplot(221)
plot_corr(x, 0.7*np.sin(2*np.pi*x))
plt.subplot(222)
plot_corr(x, np.log(x+1))
```

(continues on next page)

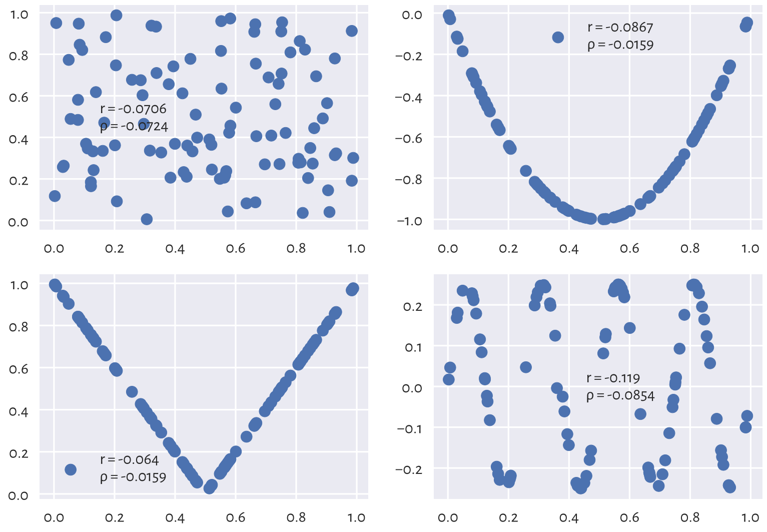


Figure10.3: Are all of these really uncorrelated?

(continued from previous page)

```
plt.subplot(223)
plot_corr(x, np.exp(x**2))
plt.subplot(224)
plot_corr(x, 1/x)
plt.show()
```

No single measure is perfect – we are trying to compress $2n$ data points into a single number — it is obvious that there will be many different datasets, sometimes very diverse, that will yield the same correlation value.

Important: Note that high correlation degree (either positive or negative) does not mean that there is any *casual* relationship between the two variables ("correlation is not causation"). We cannot say that having large arm circumference affects hip size or the other way around. There might be some *latent* variable that influences these two (e.g., maybe also related to weight?).

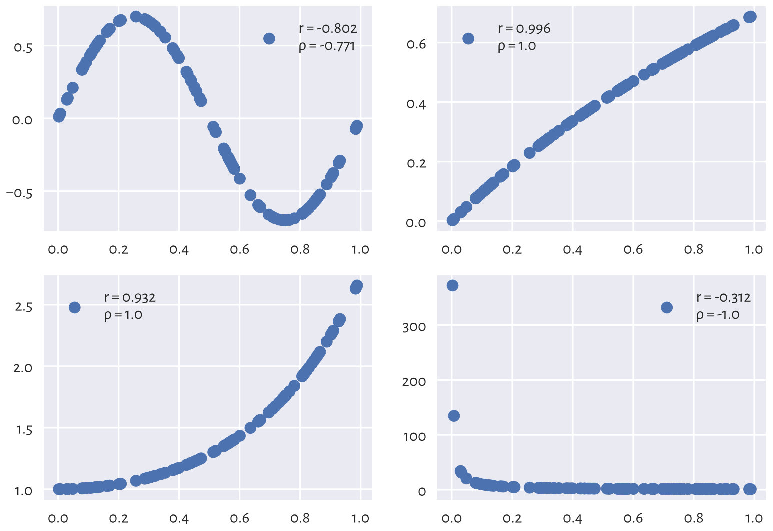


Figure10.4: Example non-linear relationships

Exercise 10.2 *Quite often, medical advice is formulated based on correlations and similar association-measuring tools. We should know how to interpret them, as it is never a true cause-effect relationship; rather, it's all about detecting common patterns in larger populations. For instance, in "obesity increases the likelihood of lower back pain and diabetes" we does not say that one necessarily* implies *another or that if you are not obese, there is no risk of getting the two said conditions. It might also work the other way around, as lower back pain may lead to less exercise and then weight gain. Reality is complex. Find similar patterns related to other sets of conditions. Which are stronger than others?*

Note: (*) Measuring correlations can aid in constructing regression models, where we would like to identify the transformation that expresses one variable as a function of one or more other ones (when we say that y can be modelled by $ax + b$, regression analysis with identify the concrete a and b coefficients). In particular, we would like to include some variables that are correlated with the modelled variable but avoid modelling with the features that are highly correlated with each other, because they do not bring anything interesting to the table and can cause the solution to be numerically unstable.

10.1.2 Correlation Heatmap

Calling **`numpy.corrcoef`**`(body.T)` (note the matrix transpose) allows for determining the linear correlation coefficients between all pairs of variables.

We can nicely depict them on a heatmap, see Figure 10.5.

```
from matplotlib import cm
order = [4, 5, 6, 0, 2, 1, 3]
cols = np.array(["weight", "height", "arm len",
    "leg len", "arm circ", "hip circ", "waist circ"])
C = np.corrcoef(body.T)
sns.heatmap(
    C[np.ix_(order, order)],
    xticklabels=cols[order],
    yticklabels=cols[order],
    annot=True, fmt=".2f", cmap=cm.get_cmap("copper")
)
plt.show()
```

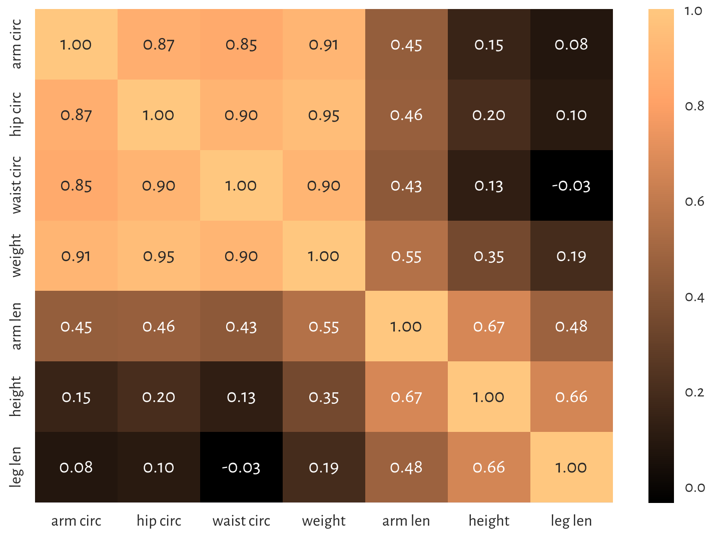


Figure10.5: A correlation heatmap

Note that we have ordered the columns to reveal some naturally occurring variable *clusters*: for instance, arm, hip, waist circumference and weight are all quite strongly correlated.

Of course, we have 1.0s on the main diagonal because a variable is trivially correlated with itself. Also, note that this heatmap is symmetric which is due to the property $r(\boldsymbol{x}, \boldsymbol{y}) = r(\boldsymbol{y}, \boldsymbol{x})$.

Example 10.3 *(*) To fetch the row and column index of the most correlated pair of variables (either positively or negatively), we should first take the upper (or lower) triangle of the correlation matrix (see* **`numpy.triu`** *or* **`numpy.tril`***) to ignore the irrelevant and repeating items:*

```
Cu = np.triu(np.abs(C), 1)
np.round(Cu, 2)
## array([[0.  , 0.35, 0.55, 0.19, 0.91, 0.95, 0.9 ],
##        [0.  , 0.  , 0.67, 0.66, 0.15, 0.2 , 0.13],
##        [0.  , 0.  , 0.  , 0.48, 0.45, 0.46, 0.43],
##        [0.  , 0.  , 0.  , 0.  , 0.08, 0.1 , 0.03],
##        [0.  , 0.  , 0.  , 0.  , 0.  , 0.87, 0.85],
##        [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.9 ],
##        [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]])
```

and then find the location of the maximum:

```
np.unravel_index(np.argmax(Cu), Cu.shape)
## (0, 5)
```

Thus, weight and hip circumference is the most strongly correlated.

10.1.3 Linear Correlation Coefficients on Transformed Data

Pearson's coefficient can of course also be applied on nonlinearly transformed versions of variables, e.g., logarithms (remember incomes?), squares, square roots, etc.

Let's consider an excerpt from the the 2020 CIA World Factbook[93], where we have data on gross domestic product per capita (based on purchasing power parity) and life expectancy at birth in many countries.

```
world = pd.read_csv("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/world_factbook_2020_subset1.csv",
    comment="#")
```

(continues on next page)

[93] https://www.cia.gov/library/publications/the-world-factbook/docs/rankorderguide.html

(continued from previous page)

```
world = world.to_numpy()
world[:6, :]  # preview
## array([[ 2000. ,    52.8],
##        [12500. ,    79. ],
##        [15200. ,    77.5],
##        [11200. ,    74.8],
##        [49900. ,    83. ],
##        [ 6800. ,    61.3]])
```

Figure 10.6 depicts these data on a scatterplot.

```
plt.subplot(121)
plot_corr(world[:, 0], world[:, 1])
plt.xlabel("per capita GDP PPP")
plt.ylabel("life expectancy (years)")
plt.subplot(122)
plot_corr(np.log(world[:, 0]), world[:, 1])
plt.xlabel("log(per capita GDP PPP)")
plt.yticks()
plt.show()
```

Computing Pearson's r between these two indicates a quite weak linear correlation:

```
scipy.stats.pearsonr(world[:, 0], world[:, 1])[0]
## 0.656471945486374
```

However, already the logarithm of GDP is slightly more strongly linearly correlated with life expectancy:

```
scipy.stats.pearsonr(np.log(world[:, 0]), world[:, 1])[0]
## 0.8066505089380016
```

which means that modelling our data via $\boldsymbol{y} = a \log \boldsymbol{x} + b$ can be an idea worth considering.

10.1.4 Spearman's Rank Correlation Coefficient

Sometimes we might be interested in measuring the degree of any kind of *monotonic* correlation – to what extent one variable is an increasing or decreasing function of another one (linear, logarithmic, quadratic over the positive domain, etc.).

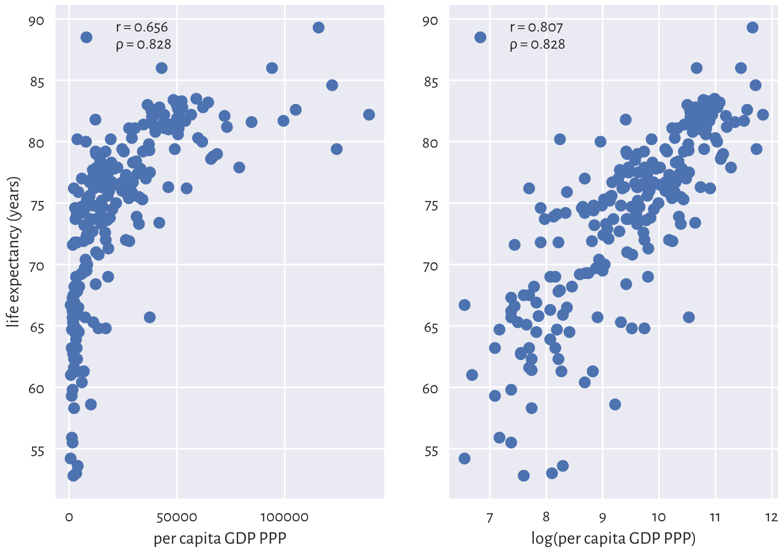


Figure10.6: Scatterplots for life expectancy vs gross domestic product (purchasing power parity) on linear (left-) and log-scale (righthand side)

Spearman's rank correlation coefficient is frequently used in such a scenario:

$$\rho(\boldsymbol{x},\boldsymbol{y}) = r(R(\boldsymbol{x}),R(\boldsymbol{y}))$$

which is[95] the Pearson coefficient computed over vectors of the corresponding ranks of all the elements in $\boldsymbol{x}$ and $\boldsymbol{y}$ (denoted with $R(\boldsymbol{x})$ and $R(\boldsymbol{y})$).

Hence, the two following calls are equivalent:

```
scipy.stats.spearmanr(world[:, 0], world[:, 1])[0]
## 0.8275220380818622
scipy.stats.pearsonr(
    scipy.stats.rankdata(world[:, 0]),
    scipy.stats.rankdata(world[:, 1])
)[0]
## 0.8275220380818621
```

[95] If a method Y is nothing else than X on transformed data, we should not consider it a totally new method.

Let us point out that this measure is invariant with respect to monotone transformations of the input variables (up to the sign):

```
scipy.stats.spearmanr(np.log(world[:, 0]), -np.sqrt(world[:, 1]))[0]
## -0.8275220380818622
```

Exercise 10.4 *We have included the ρs in all the outputs generated by our **`plot_corr`** functions. Review all the figures listed above.*

Exercise 10.5 *Apply **`numpy.corrcoef`** and **`scipy.stats.rankdata`** (with an appropriate `axis` argument) to compute the Spearman correlation matrix for all the variable pairs in `body`. Draw it on a heatmap.*

Exercise 10.6 *(*) Draw the scatterplots of the ranks of columns in the `world` and `body` datasets.*

10.2 Further Reading

(*) Kendall's τ[94] is another interesting rank correlation coefficient.

10.3 Exercises

Exercise 10.7 *Give some ways to visualise 3-dimensional data.*

Exercise 10.8 *How to and why set point opaqueness/transparency when drawing a scatter plot?*

Exercise 10.9 *What does "correlation is not causation" mean?*

Exercise 10.10 *What does linear correlation of 0.9 mean?*

Exercise 10.11 *What does rank correlation of 0.9 mean?*

Exercise 10.12 *What does linear correlation of 0.0 mean?*

Exercise 10.13 *How is the Spearman's coefficient related to the Pearson's one?*

[94] https://en.wikipedia.org/wiki/Kendall_rank_correlation_coefficient

Part IV

Heterogeneous Data

11

Introducing Data Frames

numpy arrays are an extremely versatile tool for performing data analysis and other numerical computations computations of various kinds. Although theoretically possible otherwise, in practice we only store elements of the same type therein, most often numbers.

pandas[96] [[McK17]] is amongst over one hundred thousand[97] of open source packages and repositories that use **numpy** to provide additional data wrangling functionality. It was originally written by Wes McKinney and was heavily inspired by the `data.frame`[115] objects in S and R as well as tables in relational (think: SQL) databases and spreadsheets.

It is customary to load this package with the following alias:

```
import pandas as pd
```

The **pandas** package delivers a few classes, of which the most important are:

- `DataFrame` – for representing tabular data (matrix-like) with columns of possibly different types, in particular a mix of numerical and categorical variables,
- `Series` – vector-like objects for representing individual columns,
- `Index` – vector-like (usually) objects for labelling individual rows and columns of `DataFrame`s and items in `Series` objects,

together with many methods for:

- transforming/aggregating/processing data, also in groups determined by categorical variables or products thereof,
- reshaping (e.g., from wide to long format) and joining datasets,
- importing/exporting data from/to various sources and formats, e.g., CSV and HDF5 files or relational databases,

[96] https://pandas.pydata.org/
[97] https://libraries.io/pypi/numpy
[115] Data frames were first introduced in the 1991 version of the S language [[CH91]].

- handling missing data,

which we introduce in this and further chapters.

11.1 Creating Data Frames

Important: Let us repeat: **pandas** is built on top of **numpy** and most objects therein can be processed by **numpy** functions as well. Other functions (e.g., in **seaborn** or **sklearn**) will be accepting both `DataFrame` and `ndarray` objects, but will be converting the former to the latter automatically to enable data processing using fast C/C++/Fortran routines. What we have learnt so far still applies. But there is of course more, hence this part.

Data frames can be created, amongst others, using the `DataFrame` class constructor, which can be fed, for example, with a **numpy** matrix:

```
np.random.seed(123)
pd.DataFrame(
    np.random.rand(4, 3),
    columns=["a", "b", "c"]
)
##           a         b         c
## 0  0.696469  0.286139  0.226851
## 1  0.551315  0.719469  0.423106
## 2  0.980764  0.684830  0.480932
## 3  0.392118  0.343178  0.729050
```

Note that rows and columns are labelled (and how readable that is).

A dictionary of vector-like objects of equal lengths is another common option:

```
np.random.seed(123)
pd.DataFrame(dict(
    u=np.round(np.random.rand(5), 2),
    v=[True, True, False, False, True],
    w=["A", "B", "C", "D", "E"],
    x=["spam", "spam", "bacon", "spam", "eggs"]
))
##       u      v  w      x
```

(continues on next page)

(continued from previous page)

```
## 0  0.70   True  A   spam
## 1  0.29   True  B   spam
## 2  0.23  False  C  bacon
## 3  0.55  False  D   spam
## 4  0.72   True  E   eggs
```

This illustrates the possibility of having columns of different types.

Exercise 11.1 *Check out the* **`pandas.DataFrame.from_dict`** *and* **`pandas.DataFrame.from_records`** *methods in the documentation*[98]*. Use them to create some example data frames.*

Further, data frames can be read from files in different formats, for instance, CSV:

```
body = pd.read_csv("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/nhanes_adult_female_bmx_2020.csv",
    comment="#")
body.head()  # display first few rows (5 by default)
##     BMXWT  BMXHT  BMXARML  BMXLEG  BMXARMC  BMXHIP  BMXWAIST
## 0    97.1  160.2     34.7    40.8     35.8   126.1     117.9
## 1    91.1  152.7     33.5    33.0     38.5   125.5     103.1
## 2    73.0  161.2     37.4    38.0     31.8   106.2      92.0
## 3    61.7  157.4     38.0    34.7     29.0   101.0      90.5
## 4    55.4  154.6     34.6    34.0     28.3    92.5      73.2
```

Reading from URLs and local files (compare Section 14.6.1) is of course supported.

Exercise 11.2 *Check out the other* **`pandas.read_*`** *functions in the* **`pandas`** *documentation – some of which we shall be discussing later.*

11.1.1 Data Frames are Matrix-Like

Data frames are modelled through **`numpy`** matrices, hence we can feel quite at home with them.

For example, given:

```
np.random.seed(123)
df = pd.DataFrame(dict(
    u=np.round(np.random.rand(5), 2),
    v=[True, True, False, False, True],
```

(continues on next page)

[98] https://pandas.pydata.org/docs/

(continued from previous page)

```
    w=["A", "B", "C", "D", "E"],
    x=["spam", "spam", "bacon", "spam", "eggs"]
))
df
##       u      v  w      x
## 0  0.70   True  A   spam
## 1  0.29   True  B   spam
## 2  0.23  False  C  bacon
## 3  0.55  False  D   spam
## 4  0.72   True  E   eggs
```

it is easy to fetch the number of rows and columns:

```
df.shape
## (5, 4)
```

or the type of each column:

```
df.dtypes
## u    float64
## v       bool
## w     object
## x     object
## dtype: object
```

Recall that arrays are equipped with the `dtype` slot.

11.1.2 Series

There is a separate class for storing individual data frame columns: it is called `Series`.

```
s = df.loc[:, "u"]  # extract the `u` column; alternatively: d.u
s
## 0    0.70
## 1    0.29
## 2    0.23
## 3    0.55
## 4    0.72
## Name: u, dtype: float64
```

Note that a data frame with one column is printed out slightly differently:

```
s.to_frame()
##       u
## 0  0.70
## 1  0.29
## 2  0.23
## 3  0.55
## 4  0.72
```

Indexing will of course be discussed later.

Important: It is crucial to know when we are dealing with a `Series` and when with a `DataFrame` object, because each of them defines a slightly different set of methods.

For example:

```
s.mean()
## 0.49800000000000005
```

refers to **`pandas.Series.mean`** (which returns a scalar), whereas

```
df.mean(numeric_only=True)
## u    0.498
## v    0.600
## dtype: float64
```

uses **`pandas.DataFrame.mean`** (which yields a `Series`).

Exercise 11.3 *Look up the two methods in the* **`pandas`** *manual. Note that their argument list is slightly different.*

`Series` are wrappers around **`numpy`** arrays.

```
s.values
## array([0.7 , 0.29, 0.23, 0.55, 0.72])
```

Most importantly, **`numpy`** functions can be called directly on them:

```
np.mean(s)
## 0.49800000000000005
```

Hence, what we have covered in the part dealing with vector processing still holds for data frame columns as well (there will be more, stay tuned).

`Series` can also be *named*.

```
s.name
## 'u'
```

This is convenient when we convert from/to a data frame.

```
s.rename("spam").to_frame()
##    spam
## 0  0.70
## 1  0.29
## 2  0.23
## 3  0.55
## 4  0.72
```

11.1.3 Index

Another important class is `Index`, which is used for encoding the row and column names in a data frame:

```
df.index  # row labels
## RangeIndex(start=0, stop=5, step=1)
```

The above represents a sequence (0, 1, 2, 3, 4).

```
df.columns  # column labels
## Index(['u', 'v', 'w', 'x'], dtype='object')
```

Also, we can label the individual elements in `Series` objects:

```
s.index
## RangeIndex(start=0, stop=5, step=1)
```

A quite frequent operation that we'll be applying to make a data frame column act as a vector of row labels is implemented in the `set_index` method for data frames:

```
df2 = df.set_index("x")
df2
##            u      v  w
## x
## spam    0.70   True  A
## spam    0.29   True  B
```

(continues on next page)

(continued from previous page)

```
## bacon  0.23  False  C
## spam   0.55  False  D
## eggs   0.72   True  E
```

Note that this `Index` object is named:

```
df2.index.name
## 'x'
```

which is handy when we decide that we want to convert the vector of row labels back to a standalone column:

```
df2.reset_index()
##        x     u      v  w
## 0   spam  0.70   True  A
## 1   spam  0.29   True  B
## 2  bacon  0.23  False  C
## 3   spam  0.55  False  D
## 4   eggs  0.72   True  E
```

There is also an option to get rid of the current `.index` and to replace it with the default label sequence, i.e., 0, 1, 2, …:

```
df2.reset_index(drop=True)
##       u      v  w
## 0  0.70   True  A
## 1  0.29   True  B
## 2  0.23  False  C
## 3  0.55  False  D
## 4  0.72   True  E
```

Important: In due course, we will be calling `df.reset_index(drop=True)` quite frequently, sometimes more than once in a single chain of commands.

Exercise 11.4 *Use the* ***`pandas.DataFrame.rename`*** *method to change the name of the* u *column in* `df` *to* `spam`*.*

Also, a *hierarchical* index – one that is comprised of more than two levels – is possible:

```
df.sort_values("x", ascending=False).set_index(["x", "v"])
##                  u  w
## x     v
## spam  True    0.70  A
##       True    0.29  B
##       False   0.55  D
## eggs  True    0.72  E
## bacon False   0.23  C
```

Note that a group of three consecutive `spam`s does not have the labels repeated – this is for increased readability.

Important: We will soon see that hierarchical indexes might arise after aggregating data in groups (via the **groupby** method).

For example[99]:

```
nhanes = pd.read_csv("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/nhanes_p_demo_bmx_2020.csv",
    comment="#")
res = nhanes.groupby(["RIAGENDR", "DMDEDUC2"])["BMXBMI"].mean()
res  # BMI by gender and education
## RIAGENDR  DMDEDUC2
## 1         1.0         28.765244
##           2.0         28.534737
##           3.0         29.637793
##           4.0         30.323158
##           5.0         28.689736
##           9.0         28.080000
## 2         1.0         30.489032
##           2.0         31.251247
##           3.0         30.969200
##           4.0         31.471476
##           5.0         28.891357
##           7.0         30.200000
##           9.0         33.950000
## Name: BMXBMI, dtype: float64
```

[99] https://wwwn.cdc.gov/Nchs/Nhanes/2017-2018/P_DEMO.htm

But let us fret not, as there is always:

```
res.reset_index()
##     RIAGENDR  DMDEDUC2     BMXBMI
## 0          1       1.0  28.765244
## 1          1       2.0  28.534737
## 2          1       3.0  29.637793
## 3          1       4.0  30.323158
## 4          1       5.0  28.689736
## 5          1       9.0  28.080000
## 6          2       1.0  30.489032
## 7          2       2.0  31.251247
## 8          2       3.0  30.969200
## 9          2       4.0  31.471476
## 10         2       5.0  28.891357
## 11         2       7.0  30.200000
## 12         2       9.0  33.950000
```

11.2 Aggregating

Here is an example data frame:

```
np.random.seed(123)
d = pd.DataFrame(dict(
    u = np.round(np.random.rand(5), 2),
    v = np.round(np.random.randn(5), 2),
    w = ["spam", "bacon", "spam", "eggs", "sausage"]
), index=["a", "b", "c", "d", "e"])
d
##       u     v        w
## a  0.70  0.32     spam
## b  0.29 -0.05    bacon
## c  0.23 -0.20     spam
## d  0.55  1.98     eggs
## e  0.72 -1.62  sausage
```

First, all **numpy** functions can be applied directly on individual columns (i.e., objects of type `Series`), as the latter are, after all, vector-like.

```
np.quantile(d.loc[:, "u"], [0, 0.5, 1])
## array([0.23, 0.55, 0.72])
```

For more details on the **loc**[...]-type indexing, see below. By then, the meaning of the above should be clear from the context.

Most **numpy** functions also work if they are fed with data frames, but we will need to extract the numeric matrix columns manually.

```
np.quantile(d.loc[:, ["u", "v"]], [0, 0.5, 1], axis=0)
## array([[ 0.23, -1.62],
##        [ 0.55, -0.05],
##        [ 0.72,  1.98]])
```

Sometimes the results will automatically be coerced to a `Series` object with `index` slot set appropriately:

```
np.mean(d.loc[:, ["u", "v"]], axis=0)
## u    0.498
## v    0.086
## dtype: float64
```

Many operations, for convenience, have also been implemented as methods in the `Series` and `DataFrame` classes, e.g., **mean**, **median**, **min**, **max**, **quantile**, **var**, **std** (with `ddof=1` by default, interestingly), and **skew**.

```
d.loc[:, ["u", "v"]].mean(numeric_only=True)
## u    0.498
## v    0.086
## dtype: float64
d.loc[:, ["u", "v"]].quantile([0, 0.5, 1], numeric_only=True)
##         u     v
## 0.0  0.23 -1.62
## 0.5  0.55 -0.05
## 1.0  0.72  1.98
```

Also note the **describe** method, which returns a few statistics at the same time.

```
d.describe()
##                u          v
```

(continues on next page)

(continued from previous page)

```
## count  5.000000  5.000000
## mean   0.498000  0.086000
## std    0.227969  1.289643
## min    0.230000 -1.620000
## 25%    0.290000 -0.200000
## 50%    0.550000 -0.050000
## 75%    0.700000  0.320000
## max    0.720000  1.980000
```

Note: (*) Let us stress that above we see the corrected for bias (but still only asymptotically unbiased) version of standard deviation (`ddof=1`), given by $\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$, compare Chapter 5.

Exercise 11.5 *Check out the* ***`pandas.DataFrame.agg`*** *method that can apply all aggregating operations from a given list of functions. Write a call equivalent to* *`d.describe()`*.

11.3 Transforming

By applying the already well-known vectorised mathematical **`numpy`** functions. we can transform each data cell and return an object of the same type as the input one.

```
np.exp(d.loc[:, "u"])
## a    2.013753
## b    1.336427
## c    1.258600
## d    1.733253
## e    2.054433
## Name: u, dtype: float64
np.exp(d.loc[:, ["u", "v"]])
##           u         v
## a  2.013753  1.377128
## b  1.336427  0.951229
## c  1.258600  0.818731
## d  1.733253  7.242743
## e  2.054433  0.197899
```

When applying the binary arithmetic, comparison, and logical operators on an object of class `Series` and a scalar or a **`numpy`** vector, the operations are performed elementwisely – a style with which we are already familiar.

For instance, here is a standardised version of the `u` column:

```
u = d.loc[:, "u"]
(u - np.mean(u)) / np.std(u)
## a    0.990672
## b   -1.020098
## c   -1.314357
## d    0.255025
## e    1.088759
## Name: u, dtype: float64
```

For two objects of class `Series`, the operators are vectorised somewhat differently from their **`numpy`** counterparts: they are applied on the pairs of elements having *identical labels* (i.e., labelwisely). Still, if we are transforming the columns that come from the same data frame (and thus having identical `index` slots; this is the most common scenario), there will be no surprises, because this is equivalent to elementwise action. For instance:

```
d.loc[:, "u"] > d.loc[:, "v"]
## a     True
## b     True
## c     True
## d    False
## e     True
## dtype: bool
```

For transforming many numerical columns at once, it is a good idea either to convert them to a numeric matrix explicitly and then use the basic **`numpy`** functions:

```
uv = d.loc[:, ["u", "v"]].values
uv = (uv-np.mean(uv, axis=0))/np.std(uv, axis=0)
uv
## array([[ 0.99067229,  0.20286225],
##        [-1.0200982 , -0.11790285],
##        [-1.3143573 , -0.24794275],
##        [ 0.25502455,  1.64197052],
##        [ 1.08875866, -1.47898717]])
```

or to use the **`pandas.DataFrame.apply`** method which invokes a given function on each column separately:

```
def standardise(x): return (x-np.mean(x))/np.std(x)
uv = d.loc[:, ["u", "v"]].apply(standardise)
uv
##           u         v
## a  0.990672  0.202862
## b -1.020098 -0.117903
## c -1.314357 -0.247943
## d  0.255025  1.641971
## e  1.088759 -1.478987
```

Note that, in both cases, we can then write:

```
d.loc[:, ["u", "v"]] = uv
```

to replace the old content with the transformed one.

Also, new columns can be added based on transformed versions of the existing ones:

```
d.loc[:, "uv_squared"] = (d.loc[:, "u"] * d.loc[:, "v"])**2
d
##           u         v        w  uv_squared
## a  0.990672  0.202862     spam    0.040389
## b -1.020098 -0.117903    bacon    0.014465
## c -1.314357 -0.247943     spam    0.106201
## d  0.255025  1.641971     eggs    0.175346
## e  1.088759 -1.478987  sausage    2.592938
```

11.4 Filtering `Series` Objects

Recall that each `DataFrame` and `Series` object is equipped with a slot called `index`, which is an object of class `Index` (or subclass thereof), giving the row and element labels, respectively. It turns out that we may apply the *index* operator, `[...]`, to subset these objects not only through the *indexers* known from the **numpy** part (e.g., numerical ones, i.e., by position) but also ones that pinpoint the items via their labels. That is quite a lot of index-ing.

Let us study different forms thereof in very detail.

In this section we play with two example objects of class `Series`:

```
np.random.seed(123)
b = pd.Series(np.round(np.random.uniform(0,1,10),2))
b.index =  np.random.permutation(np.r_[0:10])
b
## 2    0.70
## 1    0.29
## 8    0.23
## 7    0.55
## 9    0.72
## 4    0.42
## 5    0.98
## 6    0.68
## 3    0.48
## 0    0.39
## dtype: float64
c = b.copy()
c.index =  list("abcdefghij")
c
## a    0.70
## b    0.29
## c    0.23
## d    0.55
## e    0.72
## f    0.42
## g    0.98
## h    0.68
## i    0.48
## j    0.39
## dtype: float64
```

They consist of the same values, in the same order, but have different labels (`.index` slots). In particular, `b`'s labels are integers which *do not* match the physical element positions (where `0` would denote the first element, etc.).

Important: For **`numpy`** vectors, we had four different indexing schemes: by scalar (extracts an element at given position), slice, integer vector, and logical vector. `Series` objects are *additionally* labelled, therefore they can *additionally* be accessed taking into account the contents of the `.index` slot.

11.4.1 Do Not Use [...] Directly

Applying the index operator, `[...]`, directly on `Series` is generally not a good idea:

```
b[0]
## 0.39
b[ [0] ]
## 0    0.39
## dtype: float64
```

both do not select the first item, but the item labelled `0`.

However:

```
b[0:1]  # slice - 0 only
## 2    0.7
## dtype: float64
```

and

```
c[0]  # there is no label `0`
## 0.7
```

fall back to position-based indexing.

Confusing? Well, with some self-discipline, the solution is easy. Namely:

Important: To stay away from potential problems, we should never apply `[...]` directly on `Series` objects.

11.4.2 `loc[...]`

To avoid ambiguity, which is what we want, we should be referring to **`Series.loc`**`[...]` and **`Series.iloc`**`[...]` for label- and position-based filtering, respectively.

And thus:

```
b.loc[0]
## 0.39
```

returns the element labelled `0`. On the other hand, `c.loc[0]` will raise a `KeyError`, because `c` consists of character labels only.

Next, we can use lists of labels to select a *subset*.

```
b.loc[ [0, 1, 0] ]
## 0    0.39
## 1    0.29
## 0    0.39
## dtype: float64
c.loc[ ["j", "b", "j"] ]
## j    0.39
## b    0.29
## j    0.39
## dtype: float64
```

The result is always of type `Series`.

Slicing behaves differently as the range is *inclusive* at both sides:

```
b.loc[1:7]
## 1    0.29
## 8    0.23
## 7    0.55
## dtype: float64
b.loc[0:4:-1]
## 0    0.39
## 3    0.48
## 6    0.68
## 5    0.98
## 4    0.42
## dtype: float64
c.loc["d":"g"]
## d    0.55
## e    0.72
## f    0.42
## g    0.98
## dtype: float64
```

return all elements between the two indicated labels.

Be careful that if there are repeated labels, then we will be returning all the matching items:

```
d = pd.Series([1, 2, 3, 4], index=["a", "b", "a", "c"])
```

(continues on next page)

(continued from previous page)

```
d.loc["a"]
## a    1
## a    3
## dtype: int64
```

The result is not a scalar but a `Series` object.

11.4.3 `iloc[...]`

Here are some examples of position-based indexing:

```
b.iloc[0]  # the same: c.iloc[0]
## 0.7
```

returns the first element.

```
b.iloc[1:7]  # the same: b.iloc[1:7]
## 1    0.29
## 8    0.23
## 7    0.55
## 9    0.72
## 4    0.42
## 5    0.98
## dtype: float64
```

returns the 2nd, 3rd, …, 7th element (not including `b.iloc[7]`, i.e., the 8th one).

11.4.4 Logical Indexing

Indexing using a logical vector-like object is also available.

We usually will be using **`loc`**`[...]` with either a `Series` object of identical `.index` slot as the subsetted object or a logical **`numpy`** vector.

```
b.loc[(b > 0.4) & (b < 0.6)]
## 7    0.55
## 4    0.42
## 3    0.48
## dtype: float64
```

For **`iloc`**`[...]`, the indexer must be unlabelled, like in, e.g., `b.loc[i.values]`.

11.5 Filtering Data Frames

11.5.1 loc[...] and iloc[...]

For data frames, iloc and loc can be used too, but they now require two arguments, serving as row and column selectors.

For example:

```
np.random.seed(123)
d = pd.DataFrame(dict(
    u = np.round(np.random.rand(5), 2),
    v = np.round(np.random.randn(5), 2),
    w = ["spam", "bacon", "spam", "eggs", "sausage"],
    x = [True, False, True, False, True]
), index=["a", "b", "c", "d", "e"])
```

And now:

```
d.loc[d.loc[:, "u"] > 0.5, "u":"w"]
##         u     v        w
## a    0.70  0.32     spam
## d    0.55  1.98     eggs
## e    0.72 -1.62  sausage
```

selects the rows where the values in the u column are greater than 0.5 and then returns all columns between u and w.

Furthermore,

```
d.iloc[:3, :].loc[:, ["u", "w"]]
##         u      w
## a    0.70   spam
## b    0.29  bacon
## c    0.23   spam
```

fetches the first 3 rows (by position – iloc is necessary) and then selects two indicated columns.

Important: We can write d.u as a shorter version of d.loc[:, "u"]. This improves the readability in contexts such as:

```
d.loc[(d.u >= 0.5) & (d.u <= 0.7), ["u", "w"]]
##        u     w
## a   0.70  spam
## d   0.55  eggs
```

However, this accessor is not universal: we can check this out by considering a data frame featuring a column named, e.g., `mean`.

Exercise 11.6 *Use* ***`pandas.DataFrame.drop`*** *to select all columns but* `v` *in* `d`*.*

Exercise 11.7 *Use* ***`pandas.Series.isin`*** *(amongst others) to select all rows with* `spam` *and* `bacon` *on the* `d`*'s menu.*

Exercise 11.8 *In the* `tips`[100] *dataset, select data on male customers where total bill was in the* [10, 20] *interval. Also, select data from Saturday and Sunday with tip greater than $5 from* `tips`*.*

11.5.2 Adding Rows and Columns

`loc[...]` can also used to add new columns to an existing data frame:

```
d.loc[:, "y"] = d.loc[:, "u"]**2  # or d.loc[:, "y"] = d.u**2
d
##        u     v        w      x       y
## a   0.70  0.32     spam   True  0.4900
## b   0.29 -0.05    bacon  False  0.0841
## c   0.23 -0.20     spam   True  0.0529
## d   0.55  1.98     eggs  False  0.3025
## e   0.72 -1.62  sausage   True  0.5184
```

Note that notation like "`d.new_column = ...`" will not work. As we have said, `loc` and `iloc` are universal, other accessors – not so much.

Exercise 11.9 *Use* ***`pandas.DataFrame.insert`*** *to add a new column not necessarily at the end of* `d`*.*

Exercise 11.10 *Use* ***`pandas.DataFrame.append`*** *to add a few more rows to* `d`*.*

11.5.3 Random Sampling

As a simple application of what we have covered so far, let us consider the case of randomly sampling a number of rows from an existing data frame.

[100] https://raw.githubusercontent.com/gagolews/teaching_data/master/other/tips.csv

For the most basic use cases, we can use the **pandas.DataFrame.sample** method. It includes scenarios such as:

- randomly select 5 rows, without replacement,
- randomly select 20% rows, with replacement,
- randomly rearrange all the rows.

For example:

```
body = pd.read_csv("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/nhanes_adult_female_bmx_2020.csv",
    comment="#")
body.sample(5, random_state=123)  # 5 random rows without replacement
##       BMXWT  BMXHT  BMXARML  BMXLEG  BMXARMC  BMXHIP  BMXWAIST
## 4214   58.4  156.2     35.2    34.7     27.2    99.5      77.5
## 3361   73.7  161.0     36.5    34.5     29.0   107.6      98.2
## 3759   61.4  164.6     37.5    40.4     26.9    93.5      84.4
## 3733  120.4  158.8     33.5    34.6     40.5   147.2     129.3
## 1121  123.5  157.5     35.5    29.0     50.5   143.0     136.4
```

Note the random_state argument which controls the seed of the random number generator so that we get reproducible results. Alternatively, we could call **numpy.random.seed**.

Exercise 11.11 *Show how the 3 aforementioned scenarios can be implemented manually using* ***iloc**[...] and* ***numpy.random.permutation*** *or* ***numpy.random.choice****.*

In machine learning practice, as we shall see later, we are used to training and evaluating machine learning models on different (mutually disjoint) subsets of the whole data frame.

For instance, we might be interested in performing the so-called *train/test split*, where 80% (or 60% or 70%) of the randomly selected rows would constitute the first new data frame and the remaining 20% (or 40% or 30%) would go to the second one.

Given a data frame like:

```
x = body.head(10)  # this is just an example
x
##    BMXWT  BMXHT  BMXARML  BMXLEG  BMXARMC  BMXHIP  BMXWAIST
## 0   97.1  160.2     34.7    40.8     35.8   126.1     117.9
## 1   91.1  152.7     33.5    33.0     38.5   125.5     103.1
## 2   73.0  161.2     37.4    38.0     31.8   106.2      92.0
## 3   61.7  157.4     38.0    34.7     29.0   101.0      90.5
```

(continues on next page)

(continued from previous page)

```
## 4    55.4  154.6      34.6    34.0     28.3    92.5      73.2
## 5    62.0  144.7      32.5    34.2     29.8   106.7      84.8
## 6    66.2  166.5      37.5    37.6     32.0    96.3      95.7
## 7    75.9  154.5      35.4    37.6     32.7   107.7      98.7
## 8    77.2  159.2      38.5    40.5     35.7   102.0      97.5
## 9    91.6  174.5      36.1    45.9     35.2   121.3     100.3
```

one way to perform the aforementioned split is to generate a random permutation of the set of row indices:

```
np.random.seed(123)  # reproducibility matters
idx = np.random.permutation(x.shape[0])
idx
## array([4, 0, 7, 5, 8, 3, 1, 6, 9, 2])
```

And then to pick the first 80% of these indices to construct the data frame number one:

```
k = int(x.shape[0]*0.8)
x.iloc[idx[:k], :]
##     BMXWT  BMXHT  BMXARML  BMXLEG  BMXARMC  BMXHIP  BMXWAIST
## 4    55.4  154.6      34.6    34.0     28.3    92.5      73.2
## 0    97.1  160.2      34.7    40.8     35.8   126.1     117.9
## 7    75.9  154.5      35.4    37.6     32.7   107.7      98.7
## 5    62.0  144.7      32.5    34.2     29.8   106.7      84.8
## 8    77.2  159.2      38.5    40.5     35.7   102.0      97.5
## 3    61.7  157.4      38.0    34.7     29.0   101.0      90.5
## 1    91.1  152.7      33.5    33.0     38.5   125.5     103.1
## 6    66.2  166.5      37.5    37.6     32.0    96.3      95.7
```

and the remaining ones to generate the second dataset:

```
x.iloc[idx[k:], :]
##     BMXWT  BMXHT  BMXARML  BMXLEG  BMXARMC  BMXHIP  BMXWAIST
## 9    91.6  174.5      36.1    45.9     35.2   121.3     100.3
## 2    73.0  161.2      37.4    38.0     31.8   106.2      92.0
```

Exercise 11.12 *In the* `wine_quality_all`[101] *dataset, leave out all but the white wines. Partition the resulting data frame randomly into three data frames:* `wines_train` *(60% of the rows),* `wines_validate` *(another 20% of the rows), and* `wines_test` *(the remaining 20%).*

[101] https://raw.githubusercontent.com/gagolews/teaching_data/master/other/wine_quality_all.csv

Exercise 11.13 *Consider the white wines dataset again. Write a function* **`kfold`** *which takes a data frame* `x` *and an integer* `k>1` *on input. Return a list of data frames resulting in randomly splitting* `x` *into* `k` *disjoint chunks of equal (or almost equal if that is not possible) sizes.*

11.5.4 Hierarchical Indices (*)

Consider the following `DataFrame` object with a hierarchical index:

```
np.random.seed(123)
d = pd.DataFrame(dict(
    year = sorted([2023, 2024, 2025]*4),
    quarter = ["Q1", "Q2", "Q3", "Q4"]*3,
    data = np.round(np.random.rand(12), 2)
)).set_index(["year", "quarter"])
d
##                data
## year quarter
## 2023 Q1        0.70
##      Q2        0.29
##      Q3        0.23
##      Q4        0.55
## 2024 Q1        0.72
##      Q2        0.42
##      Q3        0.98
##      Q4        0.68
## 2025 Q1        0.48
##      Q2        0.39
##      Q3        0.34
##      Q4        0.73
```

The index has both levels named, but this is purely for aesthetic reasons.

Indexing using **`loc`**`[...]` by default relates to the first level of the hierarchy:

```
d.loc[2023, :]
##           data
## quarter
## Q1        0.70
## Q2        0.29
## Q3        0.23
## Q4        0.55
```

(continues on next page)

(continued from previous page)

```
d.loc[[2023, 2025], :]
##               data
## year quarter
## 2023 Q1       0.70
##      Q2       0.29
##      Q3       0.23
##      Q4       0.55
## 2025 Q1       0.48
##      Q2       0.39
##      Q3       0.34
##      Q4       0.73
```

To access deeper levels, we can use tuples as indexers:

```
d.loc[(2023, "Q1"), :]
## data    0.7
## Name: (2023, Q1), dtype: float64
d.loc[[(2023, "Q1"), (2024, "Q3")], :]
##               data
## year quarter
## 2023 Q1       0.70
## 2024 Q3       0.98
```

In certain scenarios, though, it will probably be much easier to subset a hierarchical index by using **reset_index** and **set_index** creatively (together with **loc**[...] and **pandas.Series.isin**, etc.).

Also, we should note that the `:` operator for slicing can be used only *directly* within square brackets, but we can always use the **slice** constructor to create them from within any context:

```
d.loc[(slice(None), ["Q1", "Q3"]), :]  # `:`, ["Q1", "Q3"]
##               data
## year quarter
## 2023 Q1       0.70
##      Q3       0.23
## 2024 Q1       0.72
##      Q3       0.98
## 2025 Q1       0.48
##      Q3       0.34
```

(continues on next page)

(continued from previous page)

```
d.loc[(slice(None, None, -1), slice("Q2", "Q3")), :]  # ::-1, "Q2":"Q3"
##               data
## year quarter
## 2025 Q3       0.34
##      Q2       0.39
## 2024 Q3       0.98
##      Q2       0.42
## 2023 Q3       0.23
##      Q2       0.29
```

11.6 Sorting

Let's consider another example dataset. Here are the yearly (for 2018) average air quality data[102] in the Australian state of Victoria.

```
air = pd.read_csv("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/air_quality_2018_means.csv",
    comment="#")
air = (
    air.
    loc[air.param_id.isin(["BPM2.5", "NO2"]), :].
    reset_index(drop=True)
)
air
##            sp_name param_id      value
## 0       Alphington   BPM2.5   7.848758
## 1       Alphington      NO2   9.558120
## 2     Altona North      NO2   9.467912
## 3        Churchill   BPM2.5   6.391230
## 4        Dandenong      NO2   9.800705
## 5        Footscray   BPM2.5   7.640948
## 6        Footscray      NO2  10.274531
## 7    Geelong South   BPM2.5   6.502762
## 8    Geelong South      NO2   5.681722
```

(continues on next page)

[102] https://discover.data.vic.gov.au/dataset/epa-air-watch-all-sites-air-quality-hourly-averages-yearly

(continued from previous page)

```
## 9   Melbourne CBD   BPM2.5   8.072998
## 10            Moe   BPM2.5   6.427079
## 11   Morwell East   BPM2.5   6.784596
## 12  Morwell South   BPM2.5   6.512849
## 13  Morwell South      NO2   5.124430
## 14      Traralgon   BPM2.5   8.024735
## 15      Traralgon      NO2   5.776333
```

sort_values is a convenient means to order the rows with respect to one criterion:

```
air.sort_values("value", ascending=False)
##             sp_name param_id      value
## 6         Footscray      NO2  10.274531
## 4         Dandenong      NO2   9.800705
## 1        Alphington      NO2   9.558120
## 2      Altona North      NO2   9.467912
## 9     Melbourne CBD   BPM2.5   8.072998
## 14        Traralgon   BPM2.5   8.024735
## 0        Alphington   BPM2.5   7.848758
## 5         Footscray   BPM2.5   7.640948
## 11     Morwell East   BPM2.5   6.784596
## 12    Morwell South   BPM2.5   6.512849
## 7     Geelong South   BPM2.5   6.502762
## 10              Moe   BPM2.5   6.427079
## 3        Churchill    BPM2.5   6.391230
## 15        Traralgon      NO2   5.776333
## 8     Geelong South      NO2   5.681722
## 13    Morwell South      NO2   5.124430
```

Sorting with respect to more criteria is also possible:

```
air.sort_values(["param_id", "value"], ascending=[True, False])
##             sp_name param_id      value
## 9     Melbourne CBD   BPM2.5   8.072998
## 14        Traralgon   BPM2.5   8.024735
## 0        Alphington   BPM2.5   7.848758
## 5         Footscray   BPM2.5   7.640948
## 11     Morwell East   BPM2.5   6.784596
## 12    Morwell South   BPM2.5   6.512849
## 7     Geelong South   BPM2.5   6.502762
```

(continues on next page)

(continued from previous page)

```
## 10           Moe   BPM2.5   6.427079
## 3      Churchill   BPM2.5   6.391230
## 6      Footscray      NO2  10.274531
## 4      Dandenong      NO2   9.800705
## 1     Alphington      NO2   9.558120
## 2   Altona North      NO2   9.467912
## 15     Traralgon      NO2   5.776333
## 8  Geelong South      NO2   5.681722
## 13 Morwell South      NO2   5.124430
```

Note that here, in each group of identical parameters, we get a decreasing order with respect to the value.

Exercise 11.14 *(*) Compare the ordering with respect to `param_id` and `value` vs `value` and then `param_id`.*

Exercise 11.15 *(**) Perform identical reorderings but using only **`loc[...]`**, **`iloc[...]`**, and **`numpy.argsort`**.*

Note: (*) **`DataFrame.sort_values`** unfortunately uses a non-stable algorithm by default (a modified quicksort). If a data frame is sorted with respect to one criterion, and then we reorder it with respect to another one, tied observations are not guaranteed to be listed in the original order:

```
(pd.read_csv("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/air_quality_2018_means.csv",
    comment="#")
    .sort_values("sp_name")
    .sort_values("param_id")
    .set_index("param_id")
    .loc[["BPM2.5", "NO2"], :]
    .reset_index())
##    param_id        sp_name      value
## 0    BPM2.5  Melbourne CBD   8.072998
## 1    BPM2.5            Moe   6.427079
## 2    BPM2.5      Footscray   7.640948
## 3    BPM2.5   Morwell East   6.784596
## 4    BPM2.5      Churchill   6.391230
## 5    BPM2.5  Morwell South   6.512849
## 6    BPM2.5      Traralgon   8.024735
```

(continues on next page)

(continued from previous page)

```
## 7    BPM2.5     Alphington   7.848758
## 8    BPM2.5   Geelong South   6.502762
## 9       NO2   Morwell South   5.124430
## 10      NO2       Traralgon   5.776333
## 11      NO2   Geelong South   5.681722
## 12      NO2    Altona North   9.467912
## 13      NO2      Alphington   9.558120
## 14      NO2       Dandenong   9.800705
## 15      NO2       Footscray  10.274531
```

We have lost the ordering based on station names in the two subgroups. To switch to a mergesort-like method (timsort), we should pass `kind="stable"`.

```
(pd.read_csv("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/air_quality_2018_means.csv",
    comment="#")
    .sort_values("sp_name")
    .sort_values("param_id", kind="stable")  # !
    .set_index("param_id")
    .loc[["BPM2.5", "NO2"], :]
    .reset_index())
##    param_id         sp_name      value
## 0    BPM2.5      Alphington   7.848758
## 1    BPM2.5       Churchill   6.391230
## 2    BPM2.5       Footscray   7.640948
## 3    BPM2.5   Geelong South   6.502762
## 4    BPM2.5   Melbourne CBD   8.072998
## 5    BPM2.5             Moe   6.427079
## 6    BPM2.5    Morwell East   6.784596
## 7    BPM2.5   Morwell South   6.512849
## 8    BPM2.5       Traralgon   8.024735
## 9       NO2      Alphington   9.558120
## 10      NO2    Altona North   9.467912
## 11      NO2       Dandenong   9.800705
## 12      NO2       Footscray  10.274531
## 13      NO2   Geelong South   5.681722
## 14      NO2   Morwell South   5.124430
## 15      NO2       Traralgon   5.776333
```

11.7 Stacking and Unstacking

Let's discuss some further ways to transform data frames, that benefit from, make sense because of, or are possible due to their being able to store data of different types.

Note that the above `air` dataset is in the so-called *long* format, where all measurements are *stacked* one after/below another. Such a form is quite convenient for the purpose of storing data, but not necessarily for all processing tasks, compare [[Wic14]]. Recall from the matrix part that a more natural way would be to have a single *observation* (e.g., data for a measurement station) in each row.

We can *unstack* the `air` data frame (convert to the *wide* format) quite easily:

```
air.set_index(["sp_name", "param_id"]).unstack().loc[:, "value"]
## param_id         BPM2.5        NO2
## sp_name
## Alphington     7.848758   9.558120
## Altona North        NaN   9.467912
## Churchill      6.391230        NaN
## Dandenong           NaN   9.800705
## Footscray      7.640948  10.274531
## Geelong South  6.502762   5.681722
## Melbourne CBD  8.072998        NaN
## Moe            6.427079        NaN
## Morwell East   6.784596        NaN
## Morwell South  6.512849   5.124430
## Traralgon      8.024735   5.776333
```

Note that the missing values are denoted with `NaN`s (not-a-number) – we have a separate section devoted to their processing given a bit later. Interestingly, we got a hierarchical index in the columns, hence the **`loc`**`[...]` part.

Equivalently:

```
air_wide = air.pivot("sp_name", "param_id", "value").\
    rename_axis(index=None, columns=None)
air_wide
##                  BPM2.5        NO2
## Alphington     7.848758   9.558120
## Altona North        NaN   9.467912
```

(continues on next page)

(continued from previous page)

```
## Churchill       6.391230        NaN
## Dandenong            NaN   9.800705
## Footscray       7.640948  10.274531
## Geelong South   6.502762   5.681722
## Melbourne CBD   8.072998        NaN
## Moe             6.427079        NaN
## Morwell East    6.784596        NaN
## Morwell South   6.512849   5.124430
## Traralgon       8.024735   5.776333
```

The **rename_axis** part is there so as to get rid of the `name` parts of the `index` and `columns` slots.

The other way around, we can use the `stack` method:

```
air_wide.T.rename_axis(index="location", columns="param").\
    stack().rename("value").reset_index()
##     location          param      value
## 0     BPM2.5     Alphington   7.848758
## 1     BPM2.5      Churchill   6.391230
## 2     BPM2.5      Footscray   7.640948
## 3     BPM2.5  Geelong South   6.502762
## 4     BPM2.5  Melbourne CBD   8.072998
## 5     BPM2.5            Moe   6.427079
## 6     BPM2.5   Morwell East   6.784596
## 7     BPM2.5  Morwell South   6.512849
## 8     BPM2.5      Traralgon   8.024735
## 9        NO2     Alphington   9.558120
## 10       NO2   Altona North   9.467912
## 11       NO2      Dandenong   9.800705
## 12       NO2      Footscray  10.274531
## 13       NO2  Geelong South   5.681722
## 14       NO2  Morwell South   5.124430
## 15       NO2      Traralgon   5.776333
```

We used the data frame transpose (**T**) to get a location-major order (less boring an outcome in this context).

11.8 Set-Theoretic Operations

Here are two not at all disjoint sets of imaginary persons:

```
A = pd.read_csv("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/some_birth_dates1.csv",
    comment="#")
A
##                     Name   BirthDate
## 0      Paitoon Ornwimol  26.06.1958
## 1          Antónia Lata  20.05.1935
## 2     Bertoldo Mallozzi  17.08.1972
## 3         Nedeljko Bukv  19.12.1921
## 4         Micha Kitchen  17.09.1930
## 5       Mefodiy Shachar  01.10.1914
## 6          Paul Meckler  29.09.1968
## 7       Katarzyna Lasko  20.10.1971
## 8          Åge Trelstad  07.03.1935
## 9   Duchanee Panomyaong  19.06.1952
```

and

```
B = pd.read_csv("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/some_birth_dates2.csv",
    comment="#")
B
##                      Name   BirthDate
## 0     Hushang Naigamwala  25.08.1991
## 1               Zhen Wei  16.11.1975
## 2          Micha Kitchen  17.09.1930
## 3            Jodoc Alwin  16.11.1969
## 4            Igor Mazał  14.05.2004
## 5        Katarzyna Lasko  20.10.1971
## 6    Duchanee Panomyaong  19.06.1952
## 7        Mefodiy Shachar  01.10.1914
## 8           Paul Meckler  29.09.1968
## 9         Noe Tae-Woong  11.07.1970
## 10          Åge Trelstad  07.03.1935
```

In both datasets, there is a single column whose elements uniquely identify each record (i.e., `Name`). In the language of relational databases, we would call it a *primary key*. In such a case, implementing the set-theoretic operations is relatively easy:

$A \cap B$ (intersection) – the rows that are both in A and in B:

```
A.loc[A.Name.isin(B.Name), :]
##                   Name   BirthDate
## 4        Micha Kitchen  17.09.1930
## 5      Mefodiy Shachar  01.10.1914
## 6         Paul Meckler  29.09.1968
## 7      Katarzyna Lasko  20.10.1971
## 8         Åge Trelstad  07.03.1935
## 9  Duchanee Panomyaong  19.06.1952
```

$A \quad B$ (difference) – the rows that are in A but not in B:

```
A.loc[~A.Name.isin(B.Name), :]
##                 Name   BirthDate
## 0   Paitoon Ornwimol  26.06.1958
## 1       Antónia Lata  20.05.1935
## 2  Bertoldo Mallozzi  17.08.1972
## 3      Nedeljko Bukv  19.12.1921
```

$A \cup B$ (union) – the rows that exist in A or are in B:

```
pd.concat((A, B.loc[~B.Name.isin(A.Name), :]))
##                   Name   BirthDate
## 0     Paitoon Ornwimol  26.06.1958
## 1         Antónia Lata  20.05.1935
## 2    Bertoldo Mallozzi  17.08.1972
## 3        Nedeljko Bukv  19.12.1921
## 4        Micha Kitchen  17.09.1930
## 5      Mefodiy Shachar  01.10.1914
## 6         Paul Meckler  29.09.1968
## 7      Katarzyna Lasko  20.10.1971
## 8         Åge Trelstad  07.03.1935
## 9  Duchanee Panomyaong  19.06.1952
## 0   Hushang Naigamwala  25.08.1991
## 1             Zhen Wei  16.11.1975
## 3          Jodoc Alwin  16.11.1969
## 4           Igor Mazał  14.05.2004
## 9        Noe Tae-Woong  11.07.1970
```

Note that there are no duplicate rows in the output.

Exercise 11.16 *Determine* $(A \cup B) \quad (A \cap B) = (A \quad B) \cup (B \quad A)$ *(symmetric difference).*

Exercise 11.17 *(*) Determine the union, intersection, and difference of the* `wine_sample1`[103] *and* `wine_sample2`[104] *datasets, where there is no column uniquely identifying the observations. Hint: consider using* **`pandas.DataFrame.duplicated`** *or* **`pandas.DataFrame.drop_duplicates`**.

11.9 Joining (Merging)

In database design, it is common to normalise[105] the datasets in order to avoid the duplication of information and pathologies stemming from them.

Example 11.18 *The above air quality parameters are separately described in another data frame:*

```
param = pd.read_csv("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/air_quality_2018_param.csv",
    comment="#")
param.rename(dict(param_std_unit_of_measure="unit"), axis=1)
##   param_id                     param_name   unit      param_short_name
## 0      API      Airborne particle index   none  Visibility Reduction
## 1   BPM2.5  BAM  Particles < 2.5 micron  ug/m3                 PM2.5
## 2       CO              Carbon Monoxide    ppm                    CO
## 3    HPM10                   Hivol PM10  ug/m3                   NaN
## 4      NO2             Nitrogen Dioxide    ppb                   NO2
## 5       O3                        Ozone    ppb                    O3
## 6     PM10     TEOM Particles <10micron  ug/m3                  PM10
## 7   PPM2.5               Partisol PM2.5  ug/m3                   NaN
## 8      SO2               Sulfur Dioxide    ppb                   SO2
```

We could have stored them alongside the `air` *data frame, but that would be a waste of space.*

Also, if we wanted to modify some datum (note, e.g., the annoying double space in `param_name` *for* `BPM2.5`*), we would have to update all the relevant records.*

Instead, we can always look the information up by `param_id` *and join (merge) the two data frames only if we need it.*

[103] https://raw.githubusercontent.com/gagolews/teaching_data/master/other/wine_sample1.csv
[104] https://raw.githubusercontent.com/gagolews/teaching_data/master/other/wine_sample2.csv
[105] https://en.wikipedia.org/wiki/Database_normalization

Let us discuss the possible join operations by studying the two following data sets:

```
A = pd.DataFrame({
    "x": ["a0", "a1", "a2", "a3"],
    "y": ["b0", "b1", "b2", "b3"]
})
A
##     x   y
## 0  a0  b0
## 1  a1  b1
## 2  a2  b2
## 3  a3  b3
```

and:

```
B = pd.DataFrame({
    "x": ["a0", "a2", "a2", "a4"],
    "z": ["c0", "c1", "c2", "c3"]
})
B
##     x   z
## 0  a0  c0
## 1  a2  c1
## 2  a2  c2
## 3  a4  c3
```

They both have one column somewhat *in common*, `x`.

The *inner (natural) join* returns the records that have a match in both datasets:

```
pd.merge(A, B, on="x")
##     x   y   z
## 0  a0  b0  c0
## 1  a2  b2  c1
## 2  a2  b2  c2
```

The *left join* of A with B guarantees to return all the records from A, even those which are not matched by anything in B.

```
pd.merge(A, B, how="left", on="x")
##     x   y    z
## 0  a0  b0   c0
```

(continues on next page)

(continued from previous page)

```
## 1  a1  b1  NaN
## 2  a2  b2   c1
## 3  a2  b2   c2
## 4  a3  b3  NaN
```

The *right join* of A with B is the same as the left join of B with A:

```
pd.merge(A, B, how="right", on="x")
##      x    y   z
## 0  a0   b0  c0
## 1  a2   b2  c1
## 2  a2   b2  c2
## 3  a4  NaN  c3
```

Finally, the *full outer join* is the set-theoretic union of the left and the right join:

```
pd.merge(A, B, how="outer", on="x")
##      x    y    z
## 0  a0   b0   c0
## 1  a1   b1  NaN
## 2  a2   b2   c1
## 3  a2   b2   c2
## 4  a3   b3  NaN
## 5  a4  NaN   c3
```

Exercise 11.19 *Join* `air_quality_2018_value`[106] *with* `air_quality_2018_point`[107] *and* `air_quality`

Exercise 11.20 *Normalise* `air_quality_2018`[109] *yourself so that you get the three data frames mentioned in the previous exercise (*`value`*,* `point`*,* `param`*).*

Exercise 11.21 *(*) In the National Health and Nutrition Examination Survey (NHANES) by the US Centres for Disease Control and Prevention, each participant is uniquely identified by their sequence number (*`SEQN`*), which is mentioned in numerous datasets, including:*

- *demographic variables*[110]*;*

[106] https://raw.githubusercontent.com/gagolews/teaching_data/master/marek/air_quality_2018_value.csv.gz
[107] https://raw.githubusercontent.com/gagolews/teaching_data/master/marek/air_quality_2018_point.csv
[108] https://raw.githubusercontent.com/gagolews/teaching_data/master/marek/air_quality_2018_param.csv
[109] https://raw.githubusercontent.com/gagolews/teaching_data/master/marek/air_quality_2018.csv.gz
[110] https://wwwn.cdc.gov/Nchs/Nhanes/2017-2018/P_DEMO.htm

- *body measures*[111];
- *audiometry*[112];
- *and many more*[113].

Join a few chosen datasets that are to your liking.

11.10 Further Reading

Looking at the list of methods for the `DataFrame` and `Series` classes in the **pandas** package's documentation[114], we can see that they are abundant. Together with the object-oriented syntax, we will often find ourselves appreciating the high readability of even quite complex operation chains such as `data.drop_duplicates().groupby(["year", "month"]).mean().reset_index()`.

Nevertheless, in fact, we should admit that the methods are too plentiful – they make an impression that someone was too generous by including a list of all the possible *verbs* related to data analysis, even if they can be easily expressed as a combination of 2-3 simpler operations. Therefore, in order to prevent the reader from being overloaded with too much new information, below we will be discussing only the most noteworthy features that often appear in data wrangling scenarios.

As strong advocates of minimalism (which is more environmentally friendly and sustainable), not rarely will we be more eager to fall back to the hand-crafted combinations of the more basic (universal) building blocks from **numpy** and **scipy** instead. This is also in line with our putting emphasis on developing *transferable* skills – as Python with **pandas** is not the only combo where we can work with data frames (e.g., base R allows that too).

11.11 Exercises

Exercise 11.22 *How are data frames different from matrices?*

[111] https://wwwn.cdc.gov/Nchs/Nhanes/2017-2018/P_BMX.htm
[112] https://wwwn.cdc.gov/Nchs/Nhanes/2017-2018/AUX_J.htm
[113] https://wwwn.cdc.gov/Nchs/Nhanes/continuousnhanes/default.aspx?BeginYear=2017
[114] https://pandas.pydata.org/pandas-docs/stable/reference/index.html

Exercise 11.23 *What are the use cases of the* `name` *slot in* `Series` *and* `Index` *objects?*

Exercise 11.24 *What is the purpose of* **`set_index`** *and* **`reset_index`***?*

Exercise 11.25 *Why learning* **`numpy`** *is crucial when someone wants to become a proficient user of* **`pandas`***?*

Exercise 11.26 *What is the difference between* **`iloc`**`[...]` *and* **`loc`**`[...]`*?*

Exercise 11.27 *Why applying the index operator* `[...]` *directly on a* `Series` *or* `DataFrame` *object is not necessarily a good idea?*

Exercise 11.28 *What is the difference between* `.index`*,* `Index`*, and* `.columns`*?*

Exercise 11.29 *How to compute the arithmetic mean and median of all the numeric columns in a data frame, using a single line of code?*

Exercise 11.30 *What is a train/test split and how to perform it using* **`numpy`** *and* **`pandas`***?*

Exercise 11.31 *What is the difference between stacking and unstacking? Which one yields a wide (as opposed to long) format?*

Exercise 11.32 *Name different data frame join (merge) operations and explain how they work.*

Exercise 11.33 *How does sorting with respect to more than one criterion work?*

Exercise 11.34 *Name the basic set-theoretic operations on data frames.*

12

Handling Categorical Data

So far we have been mostly dealing with *quantitative* (numeric) data – real numbers on which we can apply various mathematical operations, such as computing the arithmetic mean or taking the square thereof. Of course, not every transformation thereof must always make sense in every context (e.g., multiplying temperatures – what does it mean when we say that it's twice as hot today as compared to yesterday?), but still, the possibilities were plenty.

Qualitative data (also known as categorical data, factors, enumerated types), on the other hand, take a small number of unique values and support a very limited set of admissible operations. Usually, we can only determine where two entities are equal to each other or not (think: eye colour, blood type, or a flag whether a patient is ill).

In datasets involving many features, which we shall cover in Chapter 13, categorical variables are often used for observation *grouping* (e.g., so that we can compute the best and average time for marathoners in each age category or draw boxplots for finish times of men and women separately). Also, they may serve as target variables in statistical classification tasks (e.g., so that we can determine if an email is "spam" or "not spam").

Also, sometimes we might additionally be able to *rank* the observations (Australian school grades are *linearly ordered* like F (fail) < P (pass) < C (credit) < D (distinction) < HD (high distinction), some questionnaires use Likert-type scales such as "strongly disagree" < "disagree" < "neutral" < "agree" < "strongly agree", etc.).

12.1 Representing and Generating Categorical Data

Common ways to represent a categorical variable with k distinct levels is by storing it as:

- a vector of strings,
- a vector of integers between 0 (inclusive) and k (exclusive).

These two are easily interchangeable.

Furthermore, for $k = 2$ (binary data), another convenient representation is by means of logical vectors. This can be extended to a so-called one-hot encoded representation using a logical vector of length k.

Let's consider the data on the original whereabouts of the top 16 marathoners (the 37th PZU Warsaw Marathon dataset again):

```
marathon = pd.read_csv("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/37_pzu_warsaw_marathon_simplified.csv",
    comment="#")
cntrs = np.array(marathon.loc[:, "country"], dtype="str")
cntrs16 = cntrs[:16]
cntrs16
## array(['KE', 'KE', 'KE', 'ET', 'KE', 'KE', 'ET', 'MA', 'PL', 'PL', 'IL',
##        'PL', 'KE', 'KE', 'PL', 'PL'], dtype='<U2')
```

These are two-letter ISO 3166[116] country codes, encoded of course as strings (note the `dtype="str"` argument).

Calling **`numpy.unique`** allows us to determine the set of distinct categories:

```
cat_cntrs16 = np.unique(cntrs16)
cat_cntrs16
## array(['ET', 'IL', 'KE', 'MA', 'PL'], dtype='<U2')
```

Hence, `cntrs16` is a categorical vector of length n=16 (**`len`**`(cntrs16)`) with data assuming one of $k = 5$ different levels (**`len`**`(cat_cntrs16)`).

Important: **`numpy.unique`** sorts the distinct values lexicographically. In other words, they are not listed in the order of appearance, which might be something desirable in certain contexts.

12.1.1 Encoding and Decoding Factors

In order to *encode* a label vector through a set of consecutive nonnegative integers, we pass the `return_inverse=True` argument to **`numpy.unique`**:

[116] https://en.wikipedia.org/wiki/List_of_ISO_3166_country_codes

```
cat_cntrs16, codes_cntrs16 = np.unique(cntrs16, return_inverse=True)
cat_cntrs16
## array(['ET', 'IL', 'KE', 'MA', 'PL'], dtype='<U2')
codes_cntrs16
## array([2, 2, 2, 0, 2, 2, 0, 3, 4, 4, 1, 4, 2, 2, 4, 4])
```

Note that the code sequence 2, 2, 2, 0, … corresponds to the 3rd, 3rd, 3rd, 1st, … level in `cat_cntrs16`, i.e., Kenya, Kenya, Kenya, Ethiopia, ….

The values between 0 and $k - 1 = 4$ can be used to index a given array of length $k = 5$. Hence, in order to *decode* our factor, we can write:

```
cat_cntrs16[codes_cntrs16]
## array(['KE', 'KE', 'KE', 'ET', 'KE', 'KE', 'ET', 'MA', 'PL', 'PL', 'IL',
##        'PL', 'KE', 'KE', 'PL', 'PL'], dtype='<U2')
```

We can use any other set of labels now:

```
np.array(["Ethiopia", "Israel", "Kenya", "Morocco", "Poland"])[codes_cntrs16]
## array(['Kenya', 'Kenya', 'Kenya', 'Ethiopia', 'Kenya', 'Kenya',
##        'Ethiopia', 'Morocco', 'Poland', 'Poland', 'Israel', 'Poland',
##        'Kenya', 'Kenya', 'Poland', 'Poland'], dtype='<U8')
```

This is an instance of the *recoding* of a categorical variable.

Important: Despite the fact that we can *represent* categorical variables using a set of integers, it does not mean that they become instances of a quantitative type. Arithmetic operations thereon do not really make sense.

Note: When we represent categorical data using numeric codes, it is possible to introduce non-occurring levels. Such information can be useful, e.g., we could explicitly indicate that there were no runners from Australia in the top 16.

Exercise 12.1 *(**) Determine the set of unique values in `cntrs16` in the order of appearance (and not sorted lexicographically). Then, encode `cntrs16` using this level set.*

Hint: check out the `return_index` argument to **`numpy.unique`** and the **`numpy.searchsorted`** function.

12.1.2 Categorical Data in pandas

pandas includes[117] a special dtype for storing categorical data. Namely, we can write:

```
cntrs16_series = marathon.iloc[:16, :].loc[:, "country"].astype("category")
cntrs16_series
## 0      KE
## 1      KE
## 2      KE
## 3      ET
## 4      KE
## 5      KE
## 6      ET
## 7      MA
## 8      PL
## 9      PL
## 10     IL
## 11     PL
## 12     KE
## 13     KE
## 14     PL
## 15     PL
## Name: country, dtype: category
## Categories (5, object): ['ET', 'IL', 'KE', 'MA', 'PL']
```

or, equivalently in our case, `pd.Series(cntrs16, dtype="category")`. This yields a `Series` object displayed as if it was represented using string labels, however, in fact it is encoded using the numeric representation. This can be revealed by accessing:

```
cntrs16_series.cat.codes.to_numpy()
## array([2, 2, 2, 0, 2, 2, 0, 3, 4, 4, 1, 4, 2, 2, 4, 4], dtype=int8)
cntrs16_series.cat.categories
## Index(['ET', 'IL', 'KE', 'MA', 'PL'], dtype='object')
```

exactly matching what we have obtained with `numpy.unique`. Most often, however, categorical data in data frames will be stored as ordinary strings.

12.1.3 Binary Data and Logical Vectors

Binary data is a special case of the qualitative setting, where we only have two categories.

[117] https://pandas.pydata.org/pandas-docs/stable/user_guide/categorical.html

For convenience, we usually encode the two classes as integers:

- 0 (or logical `False`, e.g., healthy/fail/off/non-spam/absent/…) and
- 1 (or `True`, e.g., ill/success/on/spam/present/…).

Important: When converting logical to numeric, `False` becomes 0 and `True` becomes 1. Conversely, 0 is converted to `False` and anything else (including -0.326) to `True`.

For example:

```
np.array([True, False, True, True, False]).astype(int)
## array([1, 0, 1, 1, 0])
```

The other way around:

```
np.array([-2, -0.326, -0.000001, 0.0, 0.1, 1, 7643]).astype(bool)
## array([ True,  True,  True, False,  True,  True,  True])
```

or, equivalently:

```
np.array([-2, -0.326, -0.000001, 0.0, 0.1, 1, 7643]) != 0
## array([ True,  True,  True, False,  True,  True,  True])
```

12.1.4 One-Hot Encoding (*)

Let $\boldsymbol{x}$ be vector of n integers in $\{0, \dots, k-1\}$. Its *one-hot* encoded version is a 0-1 (or, equivalently, logical) matrix $\mathbf{R}$ of shape n by k such that $r_{i,j} = 1$ if and only if $x_i = j$.

For example, if $\boldsymbol{x} = (0, 1, 2, 1)$ and $k = 4$, then:

$$\mathbf{R} = \left[\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{array}\right].$$

Such a representation is useful when solving a multiclass classification problem by means of k binary classifiers. For example, if *spam*, *bacon*, and *hot dogs* are on the menu, then *spam* is encoded as $(1, 0, 0)$, i.e., yeah-spam, nah-bacon, and nah-hot dog. We can build three binary classifiers, each specialising in telling whether what it encounters is a given food or something else.

Example 12.2 *Write a function to one-hot encode a given categorical vector.*

Example 12.3 *Write a function to decode a one-hot encoded matrix.*

12.1.5 Binning Numeric Data (Revisited)

Numerical data can be converted to categorical via binning (quantisation). This results in information (precision) loss, however, it also opens some new possibilities. In fact, this is what we needed to do in order to draw all the histograms in Chapter 4. Also, reporting observation counts in each bin instead of raw data enables us to include them in printed reports (in form of a table).

Note: As strong proponents of openness and transparency, we always encourage all entities (government, research, non-for-profits, NGOs, corporations, etc.) to share (e.g., under the Creative Commons CC-BY-SA-4.0 license) unabridged versions of their datasets to enable public scrutiny and getting the most of the possibilities they can bring for the public good.

Of course, sometimes the sharing of unprocessed information can violate the privacy of the subjects. In such a case, it might be a good idea to communicated them in binned form.

Consider the 16 best marathon finish times (in minutes):

```
mins = marathon.loc[:, "mins"].to_numpy()
mins16 = mins[:16]
mins16
## array([129.32, 130.75, 130.97, 134.17, 134.68, 135.97, 139.88, 143.2 ,
##        145.22, 145.92, 146.83, 147.8 , 149.65, 149.88, 152.65, 152.88])
```

`numpy.searchsorted` can be used to determine the interval where each value in `mins` falls.

```
bins = [130, 140, 150]
codes_mins16 = np.searchsorted(bins, mins16)
codes_mins16
## array([0, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 3, 3])
```

By default, the intervals are of the form $(a, b]$ (not including a, including b). Code 0 corresponds to values less than the first bin bound, whereas code 3 – greater than or equal to the last bound:

`pandas.cut` us another interface to the same binning method. It returns a vector-like object of `dtype "category"`, with very readable labels generated automatically (and ordered, see Section 12.4.4):

```
cut_mins16 = pd.Series(pd.cut(mins16, [-np.inf, 130, 140, 150, np.inf]))
cut_mins16
## 0       (-inf, 130.0]
## 1      (130.0, 140.0]
## 2      (130.0, 140.0]
## 3      (130.0, 140.0]
## 4      (130.0, 140.0]
## 5      (130.0, 140.0]
## 6      (130.0, 140.0]
## 7      (140.0, 150.0]
## 8      (140.0, 150.0]
## 9      (140.0, 150.0]
## 10     (140.0, 150.0]
## 11     (140.0, 150.0]
## 12     (140.0, 150.0]
## 13     (140.0, 150.0]
## 14       (150.0, inf]
## 15       (150.0, inf]
## dtype: category
## Categories (4, interval[float64, right]): [(-inf, 130.0] < (130.0, 140.0]
 ↪< (140.0, 150.0] <
##                                              (150.0, inf]]
cut_mins16.cat.categories.astype("str")
## Index(['(-inf, 130.0]', '(130.0, 140.0]', '(140.0, 150.0]', '(150.0, inf]
 ↪'], dtype='object')
```

Exercise 12.4 *(*) Check out the* ***numpy.histogram_bin_edges*** *function which tries to determine some informative interval bounds automatically based on a range of simple heuristics. Also, note that* ***numpy.linspace*** *and* ***numpy.geomspace*** *which we have covered in Chapter 4 can be useful for generating equidistant bounds on linear and logarithmic scale, respectively.*

Example 12.5 *(**) We can create a set of the corresponding categories manually, for example, as follows:*

```
bins2 = np.r_[-np.inf, bins, np.inf]
cat_mins16 = np.array(
    [f"({bins2[i-1]}, {bins2[i]}]" for i in range(1, len(bins2))]
)
cat_mins16
## array(['(-inf, 130.0]', '(130.0, 140.0]', '(140.0, 150.0]',
##        '(150.0, inf]'], dtype='<U14')
```

Recall from Section 6.4.4 that list comprehensions are a convenient substitute for a **for** *loop and the* **list.append** *method. Recoding based on the above yields:*

```
cat_mins16[codes_mins16]
## array(['(-inf, 130.0]', '(130.0, 140.0]', '(130.0, 140.0]',
##        '(130.0, 140.0]', '(130.0, 140.0]', '(130.0, 140.0]',
##        '(130.0, 140.0]', '(140.0, 150.0]', '(140.0, 150.0]',
##        '(140.0, 150.0]', '(140.0, 150.0]', '(140.0, 150.0]',
##        '(140.0, 150.0]', '(140.0, 150.0]', '(150.0, inf]', '(150.0, inf]
↪'],
##       dtype='<U14')
```

12.1.6 Generating Pseudorandom Labels

It is worth knowing that **numpy.random.choice** allows us to create a pseudorandom sample of with categories picked with any probabilities:

```
np.random.seed(123)
np.random.choice(
    a=["spam", "bacon", "eggs", "tempeh"],
    p=[   0.7,     0.1,   0.15,     0.05],
    replace=True,
    size=16
)
## array(['spam', 'spam', 'spam', 'spam', 'bacon', 'spam', 'tempeh', 'spam',
##        'spam', 'spam', 'spam', 'bacon', 'spam', 'spam', 'spam', 'bacon'],
##       dtype='<U6')
```

Hence, if we generate a sufficiently large sample, we will expect "spam" to occur ca. 70% times, and "tempeh" to be drawn in 5% of the cases, etc.

12.2 Frequency Distributions

12.2.1 Counting

For arbitrary categorical data, we can call:

```
cat_cntrs16, counts_cntrs16 = np.unique(cntrs16, return_counts=True)
cat_cntrs16, counts_cntrs16
## (array(['ET', 'IL', 'KE', 'MA', 'PL'], dtype='<U2'), array([2, 1, 7, 1,
↪5]))
```

to get both the set of unique categories and the *corresponding* number of occurrences. For instance, there were 7 runners from Kenya amongst the top 16.

Also, note that if we *already* have an array of integer codes between 0 and $k-1$, there is no need to call **numpy.unique**, as **numpy.bincount** can return the number of times each code appears therein.

```
np.bincount(codes_cntrs16)
## array([2, 1, 7, 1, 5])
```

Of course, a vector of counts can easily be turned into a vector of proportions (fractions):

```
counts_cntrs16 / np.sum(counts_cntrs16)
## array([0.125 , 0.0625, 0.4375, 0.0625, 0.3125])
```

Hence, almost 31.25% of the top runners were from Poland (it is a marathon in Warsaw after all…). Note that we can of course multiply the above by 100 to get the percentages.

We can output a nice *frequency table* by storing both objects in a single data frame (or a `Series` object with an appropriate vector of row labels):

```
table = pd.DataFrame({
    "Country": cat_cntrs16,
    "%": 100 * counts_cntrs16 / np.sum(counts_cntrs16)
})
table
##   Country      %
## 0      ET  12.50
## 1      IL   6.25
## 2      KE  43.75
## 3      MA   6.25
## 4      PL  31.25
```

Example 12.6 *(*) In Section 15.5 we will discuss* ***IPython.display.Markdown*** *as a means to embed arbitrary Markdown code inside IPython/Jupyter reports. In the meantime, let us just note that nicely formatted tables can be created from data frames by calling:*

```
import IPython.display
IPython.display.Markdown(table.to_markdown(index=False))
```

Country	%
ET	12.5
IL	6.25
KE	43.75
MA	6.25
PL	31.25

pandas.Series.value_counts is even more convenient, as it returns a `Series` object equipped with a readable index (element labels):

```
marathon.iloc[:16, :].loc[:, "country"].value_counts()
## KE    7
## PL    5
## ET    2
## MA    1
## IL    1
## Name: country, dtype: int64
```

Note that by default data are ordered with respect to the counts, decreasingly.

Exercise 12.7 *In Chapter 4 we have mentioned the **numpy.histogram** function which applies the binning of a numeric vector and then counts the number of occurrences. This is merely a helper function: the same result can be obtained by means of the more basic **numpy.searchsorted**, **numpy.bincount**, and **numpy.histogram_bin_edges**. Apply **numpy.histogram** on the whole 37_pzu_warsaw_marathon_mins dataset.*

Exercise 12.8 *Using **numpy.argsort**, sort counts_cntrs16 increasingly together with the corresponding items in cat_cntrs16.*

12.2.2 Two-Way Contingency Tables: Factor Combinations

Some datasets may feature many categorical columns, each having possibly different levels. Let us now consider a different slice of the `marathon` dataset, this time describing the top 1000 runners:

```
m1000 = marathon.iloc[:1000, :].copy()
m1000.loc[:, "age"] = m1000.loc[:, "category"].str.slice(1)
```

(continues on next page)

(continued from previous page)

```
m1000 = m1000.loc[:, ["sex", "age", "country"]]
m1000.head()
##   sex age country
## 0   M  20      KE
## 1   M  20      KE
## 2   M  20      KE
## 3   M  20      ET
## 4   M  30      KE
```

The three columns are: sex, age (in 10-year brackets), and country. We can of course analyse the data distribution in each column individually, however, some interesting patterns might also arise when we study the *combinations* of levels of different variables.

Here are the levels of the sex and age variables:

```
np.unique(m1000.loc[:, "sex"])
## array(['F', 'M'], dtype=object)
np.unique(m1000.loc[:, "age"])
## array(['20', '30', '40', '50', '60'], dtype=object)
```

We thus have 10 different possible combinations thereof.

A two-way *contingency table* is a matrix which gives the number of occurrences of each pair of values:

```
v = m1000.loc[:, ["sex", "age"]].value_counts().unstack(fill_value=0)
v
## age   20   30   40  50  60
## sex
## F     17   23    8   1   0
## M    158  433  278  73   9
```

Hence, for example, there were 9 men aged 60-69 amongst the top 1000 runners. Nice.

Note that the *marginal* (one dimensional) frequency distributions can be recreated by computing the rowwise and columnwise sums.

```
np.sum(v, axis=1)
## sex
## F     49
## M    951
```

(continues on next page)

(continued from previous page)

```
## dtype: int64
np.sum(v, axis=0)
## age
## 20    175
## 30    456
## 40    286
## 50     74
## 60      9
## dtype: int64
```

12.2.3 Combinations of Even More Factors

`pandas.DataFrame.value_counts` can also be used with a combination of more than 2 categorical variables:

```
m1000.loc[:, ["sex", "age", "country"]].value_counts().\
    rename("count").reset_index()
##    sex age country  count
## 0    M  30      PL    391
## 1    M  40      PL    251
## 2    M  20      PL    142
## 3    M  50      PL     65
## 4    F  30      PL     17
## ..  ..  ..     ...    ...
## 61   M  30      CY      1
## 62   M  30      CZ      1
## 63   F  20      ET      1
## 64   M  30      FR      1
## 65   F  20      CA      1
##
## [66 rows x 4 columns]
```

Of course, the display will be in the *long* format (compare Section 11.7) here, as high-dimensional arrays are not nicely printable.

12.3 Visualising

Methods for visualising categorical data are by no means fascinating (unless we use them as grouping variables in more complex datasets, but this is a topic that we cover in Chapter 13).

12.3.1 Bar Plots

Bar plots are self-explanatory and hence will do the trick most of the time, see Figure 12.1.

```
ind = np.arange(len(cat_cntrs16))
plt.bar(ind, height=counts_cntrs16)
plt.xticks(ind, cat_cntrs16)
plt.show()
```

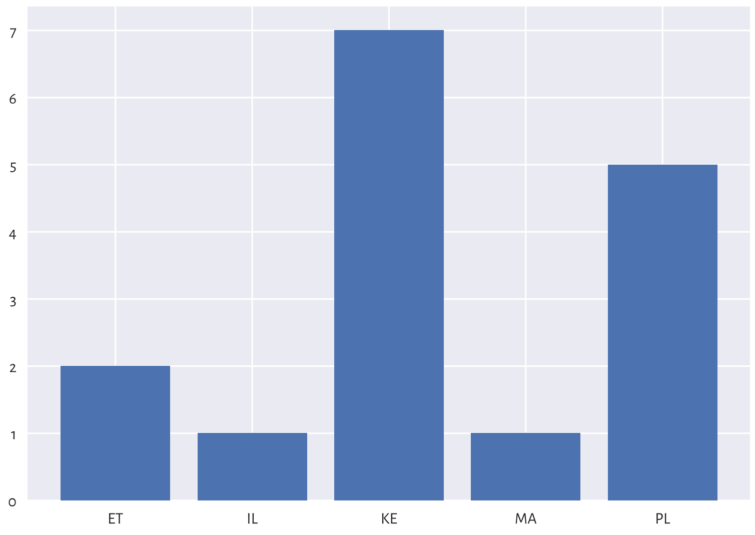


Figure12.1: Bar plot for the top 16 marathoners' countries

The `ind` vector gives the x-coordinates of the bars, here: consecutive integers. By calling **`matplotlib.pyplot.xticks`** we assign them readable labels.

Exercise 12.9 *Assign a different colour to each bar.*

Exercise 12.10 *Draw a bar plot which features percentages instead of counts, so that the total bar height is 100%.*

Exercise 12.11 *Print the frequency table and draw a bar plot for the top 100 marathoners' countries.*

Exercise 12.12 *(*) Print the frequency table and draw a bar plot all the marathoners (not just the elite ones), ordered from the highest to the lowest counts. There will be too many bars, therefore replace the few last bars with a single one, labelled "All other".*

A bar plot is a versatile tool for visualising the counts also in the two-variable case, see Figure 12.2:

```
v = m1000.loc[:, ["sex", "age"]].value_counts().\
    rename("count").reset_index()
sns.barplot(x="age", hue="sex", y="count", data=v)
plt.show()
```

Exercise 12.13 *(*) Draw a similar chart using **`matplotlib.pyplot.bar`**.*

Exercise 12.14 *(**) Create a stacked bar plot similar to the one in Figure 12.3, where we have horizontal bars for data that have been normalised so that for each sex they sum to 100%.*

12.3.2 Political Marketing and Statistics

Even such a simple plot can be manipulated. For example, presidential elections were held in Poland in 2020. In the second round, Andrzej Duda had won against Rafał Trzaskowski. In Figure 12.4 we have the official results that might be presented by one of the infamous Polish TV conglomerate:

```
plt.bar([1, 2], height=[51.03, 48.97], width=0.25)
plt.xticks([1, 2], ["Duda", "Trzaskowski"])
plt.ylabel("%")
plt.xlim(0, 3)
plt.ylim(48.9, 51.1)
plt.show()
```

Such a great victory! Wait… it was a close vote after all! We should just take a look at the y-axis tick marks.

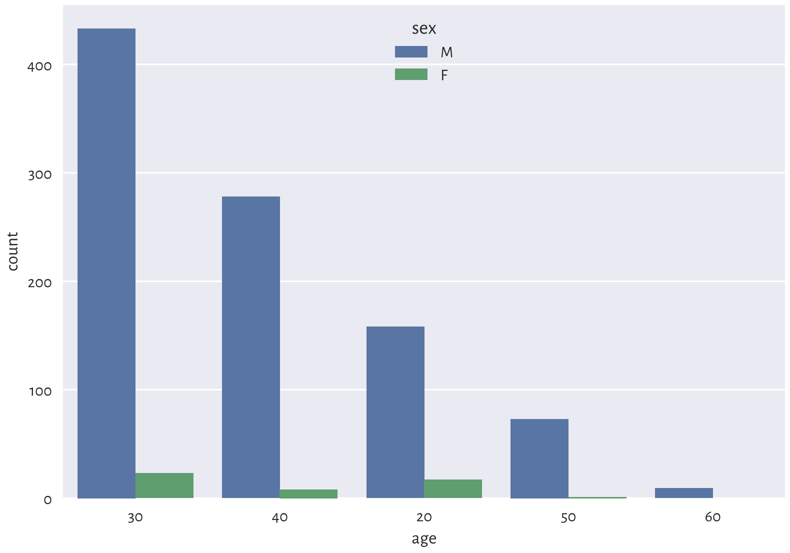


Figure12.2: Number of runners by age category and sex in the top 1000

Another media outlet could have reported it like in Figure 12.5:

```
plt.bar([1, 2], height=[51.03, 48.97], width=0.25)
plt.xticks([1, 2], ["Duda", "Trzaskowski"])
plt.ylabel("%")
plt.xlim(0, 3)
plt.ylim(0, 250)
plt.yticks([0, 100])
plt.show()
```

The moral of the story is:

Important: Always read the y-axis tick marks. And when drawing own bar plots, do not trick the reader; this is unethical.

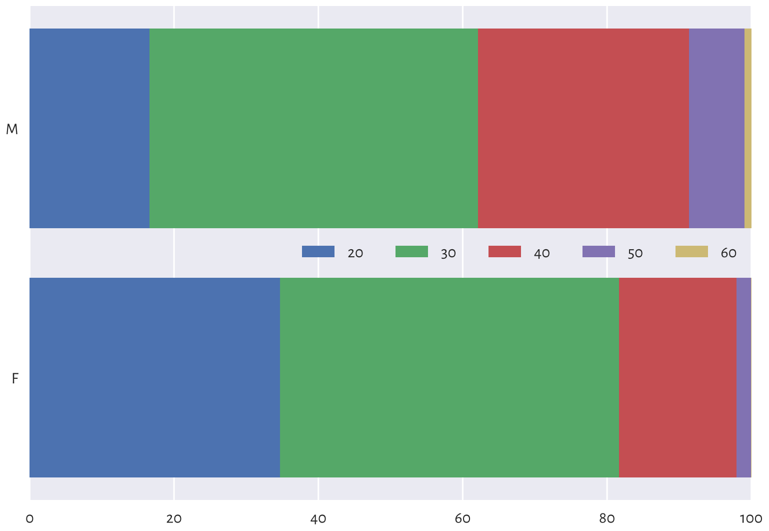


Figure12.3: Example stacked bar plot: Age distribution for different sexes amongst the top 1000 runners

12.3.3 Error Bars (**)

Occasionally, the data we have at hand are based on small surveys and are supposed to serve as estimates of what might be happening in a larger population.

For instance, one of the pre-election polls indicated that c=516 out of n=1017 people would vote for Duda. It is ca. 50.74%, but this number is subject to sampling error.

If we asked a statistician for the error estimates, they could provide us with, for example, 95% confidence intervals for the true proportions (**scipy.stats.binomtest** has a procedure implementing this). For both candidates, they are within ca. ±3.1% from the sample estimates. It would be nice to inform the reader of the figure that we are uncertain about the truth. *Error bars* can serve this purpose, see Figure 12.6.

```
c, n = 516, 1017
plt.bar([1, 2], height=[100*c/n, 100*(n-c)/n], width=0.25, label="Polls")
plt.errorbar([1, 2], [100*c/n, 100*(n-c)/n], yerr=3.1, fmt="r")
plt.plot([1, 2], [51.03, 48.97], 'yx:', label="Final results")
```

(continues on next page)

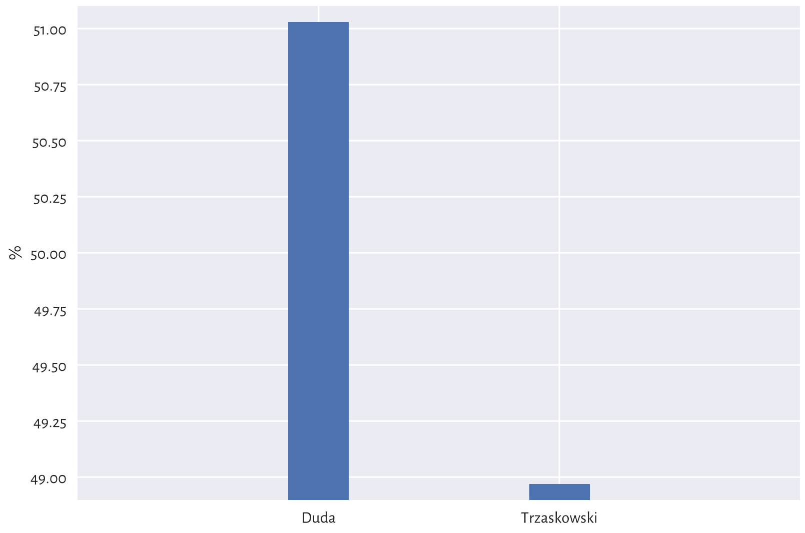


Figure12.4: Flawless victory!

(continued from previous page)

```
plt.xticks([1, 2], ["Duda", "Trzaskowski"])
plt.xlim(0, 3)
plt.ylabel("%")
plt.legend()
plt.show()
```

In order to improve the uncertainty of the estimates, we should of course have to gather more responses, e.g., a sample of size 10,000 would give errors of ±0.1% (in a version of planet Earth where people don't change their minds in the meantime).

To recall, the winner scored 51.03% on the election day, where 20,458,911 valid votes were cast in total. Therefore, in this case, the cited poll was very representative, but the final result could turn the other way.

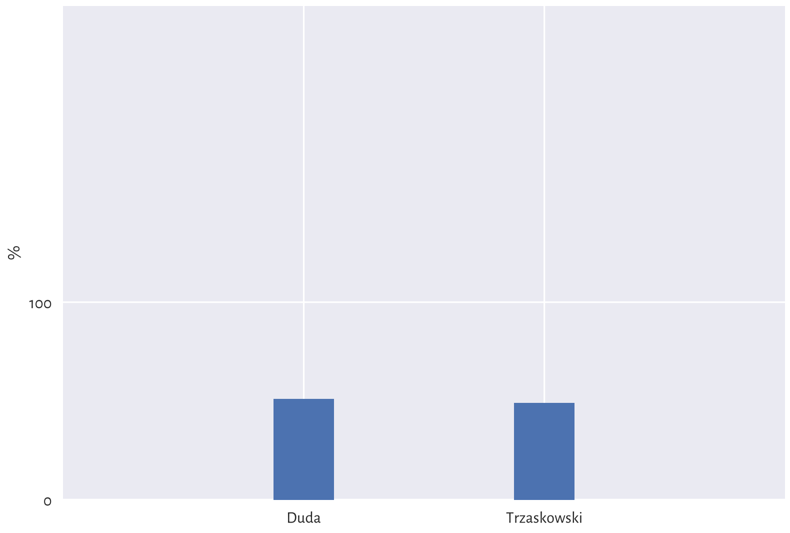


Figure12.5: It was a draw, so close!

12.3.4 Pie Cha… Don't Even Trip

We are definitely not going to discuss the infamous pie charts, because their use in data analysis has been widely criticised for a long time (it's difficult to judge the ratios of areas of their slices). Case closed. Good morning.

12.3.5 Pareto Charts (*)

As a general (empirical) rule, it is usually the case that most instances of something's happening (usually 70–90%) are due to only few causes (10–30%). This is known as the *Pareto rule* (with 80% vs 20% being an often cited rule of thumb).

Example 12.15 *In Chapter 7 we modelled the US cities' population dataset using the Pareto distribution (the very same Pareto, but a different, yet related object). We discovered that only ca. 14% of the settlements (those with 10,000 or more inhabitants) is home to as much as 84% of the population. Hence, we may say that this data domain follows the Pareto rule.*

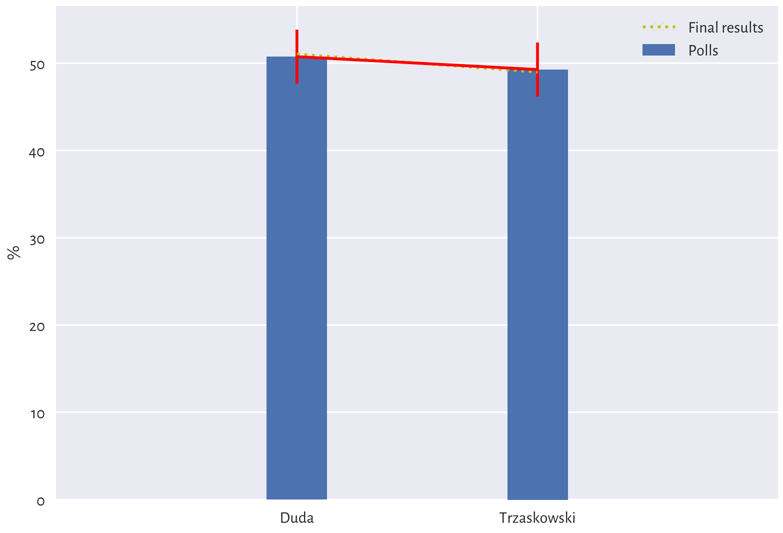


Figure12.6: Pre-election polls, 50.74% wanted to vote for Duda (c=516 out of n=1017, with error ±3.1%)

Here is a dataset[118] fabricated by the Clinical Excellence Commission in New South Wales, Australia, listing the most frequent causes for medication errors:

```
cat_med = np.array([
    "Unauthorised drug", "Wrong IV rate", "Wrong patient", "Dose missed",
    "Underdose", "Wrong calculation","Wrong route", "Wrong drug",
    "Wrong time", "Technique error", "Duplicated drugs", "Overdose"
])
counts_med = np.array([1, 4, 53, 92, 7, 16, 27, 76, 83, 3, 9, 59])
np.sum(counts_med)  # total number of medication errors
## 430
```

Let's order the dataset with respect to the counts, decreasingly:

[118] https://www.cec.health.nsw.gov.au/CEC-Academy/quality-improvement-tools/pareto-charts

```
o = np.argsort(counts_med)[::-1]  # ordering permutation (decreasing)
cat_med = cat_med[o]  # order categories based on counts
counts_med = counts_med[o]  # equivalent to np.sort(counts_med)[::-1]
pd.DataFrame(dict(
    category=cat_med,
    count=counts_med
))  # nicer display
##               category  count
## 0          Dose missed     92
## 1           Wrong time     83
## 2           Wrong drug     76
## 3             Overdose     59
## 4        Wrong patient     53
## 5          Wrong route     27
## 6    Wrong calculation     16
## 7     Duplicated drugs      9
## 8            Underdose      7
## 9        Wrong IV rate      4
## 10     Technique error      3
## 11  Unauthorised drug       1
```

Pareto charts are tools which may aid in visualising the Pareto-ruled datasets. They are based on bar plots, but feature some extras:

- bars are listed in a decreasing order,
- the cumulative percentage curve is added.

The plotting of the Pareto chart is a little tricky, because it involves using two different Y axes (as usual, fine-tuning of the figure and studying the manual of the **matplotlib** package is left as an exercise.)

```
x = np.arange(len(cat_med))  # 0, 1, 2, ...
p = 100.0*counts_med/np.sum(counts_med)  # percentages

fig, ax1 = plt.subplots()
ax1.set_xticks(x-0.5, cat_med, rotation=60)
ax1.set_ylabel("%")
ax1.bar(x, height=p)
ax2 = ax1.twinx()  # creates a new coordinate system with a shared x-axis
ax2.plot(x, np.cumsum(p), "ro-")
ax2.grid(visible=False)
```

(continues on next page)

(continued from previous page)

```
ax2.set_ylabel("cumulative %")

fig.tight_layout()
plt.show()
```

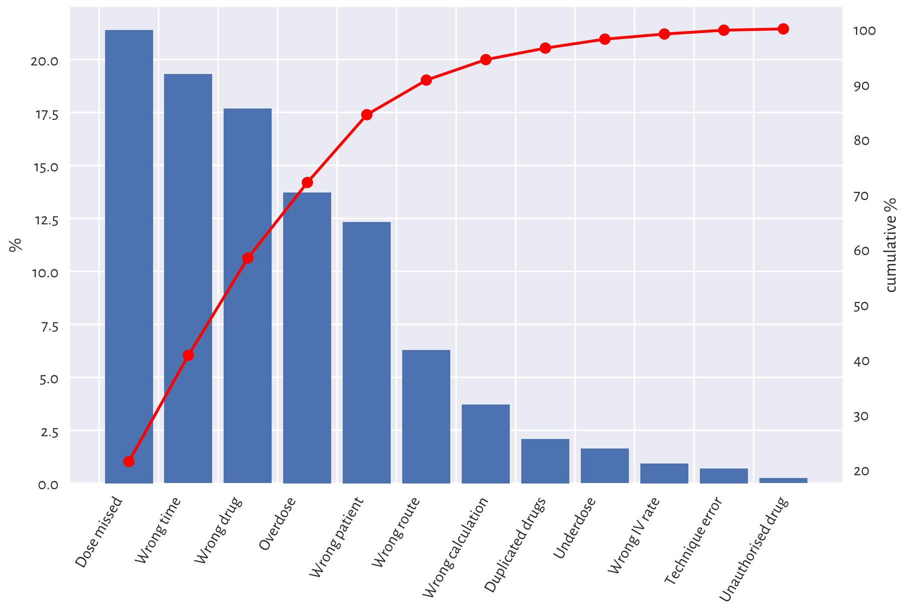


Figure12.7: The most frequent causes for medication errors

From Figure 12.7, we can read that 5 causes (less than 40%) correspond to ca. 85% of the medication errors. More precisely,

```
pd.DataFrame({
    "category": cat_med,
    "cumulative %": np.round(np.cumsum(p), 1)
})
##              category  cumulative %
## 0         Dose missed          21.4
## 1          Wrong time          40.7
## 2          Wrong drug          58.4
```

(continues on next page)

(continued from previous page)

```
## 3            Overdose         72.1
## 4       Wrong patient         84.4
## 5         Wrong route         90.7
## 6   Wrong calculation         94.4
## 7    Duplicated drugs         96.5
## 8           Underdose         98.1
## 9       Wrong IV rate         99.1
## 10    Technique error         99.8
## 11  Unauthorised drug        100.0
```

Note that there is an explicit assumption here that a single error is only due to a single cause. Also, we presume that each medication error has a similar degree of severity.

Policy makers and quality controllers often rely on such simplifications, therefore they most probably are going to be addressing only the top causes. If we have ever wondered why some processes (mal)function the way they do, there is a hint above. However, coming up with something more effective yet so simple at the same time requires much more effort.

12.3.6 Heat Maps

Two-way contingency tables can be depicted by means of a heatmap, where each count affects the corresponding cell's colour intensity, see Figure 12.8.

```
from matplotlib import cm
v = m1000.loc[:, ["sex", "age"]].value_counts().unstack(fill_value=0)
sns.heatmap(v, annot=True, fmt="d", cmap=cm.get_cmap("copper"))
plt.show()
```

12.4 Aggregating

As we have already said, the only operation on categorical data that we can rely on is counting (because we have an equivalence relation on the set of labels and nothing more). Therefore, as far as qualitative data aggregation is concerned, what we are with left is the *mode*, i.e., the most frequently occurring value.

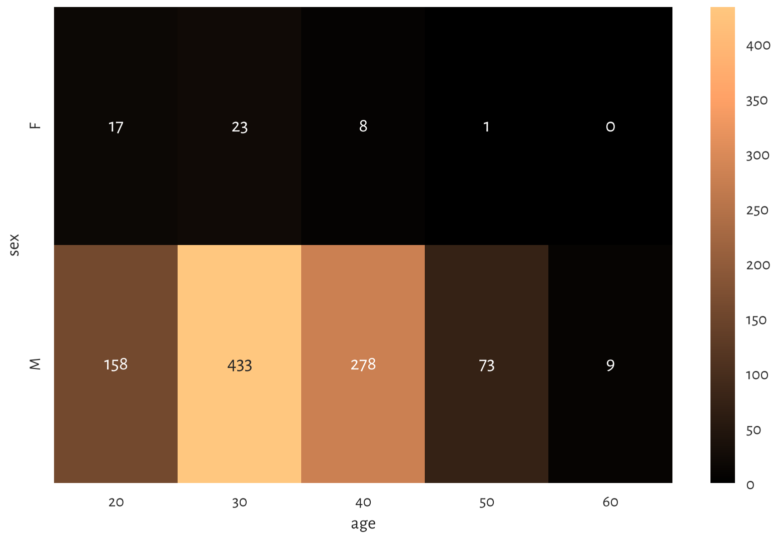


Figure12.8: Heatmap for the top 1000 marathoners' sex and age category

12.4.1 Categorical Data

There is one more measure of central tendency worth mentioning. Namely, a *mode* is a most frequently occurring value.

```
cat_cntrs16, counts_cntrs16 = np.unique(cntrs16, return_counts=True)
cat_cntrs16[np.argmax(counts_cntrs16)]
## 'KE'
```

Recall that if `i` is **`numpy.argmax`**`(counts)` (*argument maximum*, i.e., where is it?), then `counts[i]` is the same as **`numpy.max`**`(counts)` and `cat[i]` is the category with the greatest counts.

Important: A mode might be ambiguous.

For instance, amongst the fastest 22 runners, there is a tie between Kenya and Poland – both meet our definition of a mode:

```
cat_cntrs22, counts_cntrs22 = np.unique(cntrs[:22], return_counts=True)
cat_cntrs22[np.where(counts_cntrs22 == np.max(counts_cntrs22))]
## array(['KE', 'PL'], dtype='<U2')
```

To avoid any bias, it would be best to report both of them as potential mode candidates. Alternatively, we can pick one at random (calling **numpy.random.choice**).

12.4.2 Binary Data as Logical Vectors

Perhaps the most useful arithmetic operation on logical vectors is the sum.

```
cntrs16 = cntrs[:16]
cntrs16  # recall
## array(['KE', 'KE', 'KE', 'ET', 'KE', 'KE', 'ET', 'MA', 'PL', 'PL', 'IL',
##        'PL', 'KE', 'KE', 'PL', 'PL'], dtype='<U2')
np.sum(cntrs16 == "PL")
## 5
```

is the number of elements in `cntrs16` that are equal to `"PL"` (because the sum of 0s and 1s is equal to the number of 1s in the sequence). Note that `(cntrs16 == "PL")` is a logical vector that represents a binary categorical variable with levels: not-Poland (`False`) and Poland (`True`).

If we divide the above result by the length of the vector, we will get the proportion:

```
np.mean(cntrs16 == "PL")
## 0.3125
```

Hence, 31.25% amongst the top 16 runners are from Poland.

Exercise 12.16 *What is the meaning of* ***numpy.all***, ***numpy.any***, ***numpy.min***, ***numpy.max***, ***numpy.cumsum***, *and* ***numpy.cumprod*** *applied on logical vectors?*

Note: (**) Having the 0/1 (or zero/nonzero) vs `False`/`True` correspondence allows us to perform some logical operations using integer arithmetic. In particular, assuming that `p` and `q` are logical values and `a` and `b` are numeric ones, we have, what follows:

- `p+q != 0` means that at least one value is `True` and `p+q == 0` if and only if both are `False`;
- more generally, `p+q == 2` if both elements are `True`, `p+q == 1` if only one is `True` (we call it exclusive-or, XOR), and `p+q == 0` if both are `False`;

- `p*q != 0` means that both values are `True` and `p*q == 0` holds whenever at least one is `False`;
- `1-p` corresponds to negation of `p`;
- `p*a + (1-p)*b` is equal to `a` if `p` is `True` and equal to `b` otherwise.

12.4.3 Binned Numeric Data

Note that generally modes do not work for continuous data, where repeated values are – at least theoretically – highly unlikely (unless someone does not report them with full digit precision). It might make sense to compute it on binned data, though.

Looking at a histogram, the mode is the interval corresponding to the highest bar (hopefully assuming there is only one). If we would like to obtain a single number, we can choose for example the middle of this interval as the mode.

Of course, for numeric data, the mode will heavily depend on the binning (recall that we can also apply logarithmic binning). Thus, the question "what is the most popular income" is overall quite difficult to answer.

Exercise 12.17 *Compute some potentially informative modes for the 2020 UK income*[119] *data. Play around with different numbers of bins for linear and logarithmic binning and see how it affects the mode.*

12.4.4 Ordinal Data (*)

The case where the categories can be linearly ordered, is called *ordinal data*. This gives a few more options as except for the mode, thanks to the existence of order statistics, we can also easily define sample quantiles. However, the standard methods for resolving ties will not work, hence we need to be careful.

For example, median of a sample of student grades (P, P, C, D, HD) is C, but (P, P, C, D, HD, HD) is either C or D - we can choose one at random or just report that the solution is ambiguous.

Another option, of course, is to treat ordinal data as numbers (e.g., F=0, P=1, …, HD=4). In the latter example, the median would be equal to 2.5.

There are some cases, though, where the conversion of labels to consecutive integers is far from optimal – because it gives the impression that the "distance" between different levels is always equal (linear).

[119] https://raw.githubusercontent.com/gagolews/%5Cteaching_data/master/marek/uk_income_simulated_2020.txt

Exercise 12.18 *(**) The grades_results[120] represents the grades (F, P, C, D, HD) of 100 students attending an imaginary course in a virtual Australian university. You can load it in form of an* ordered *categorical* `Series` *by calling:*

```
grades = np.loadtxt("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/grades_results.txt", dtype="str")
grades = pd.Series(pd.Categorical(grades,
    categories=["F", "P", "C", "D", "HD"], ordered=True))
grades
## 0       F
## 1       F
## 2       F
## 3       F
## 4       F
##        ..
## 118    HD
## 119    HD
## 120    HD
## 121    HD
## 122    HD
## Length: 123, dtype: category
## Categories (5, object): ['F' < 'P' < 'C' < 'D' < 'HD']
```

How would you determine the average *grade represented as a number between 0 and 100, taking into account that for a P you need at least 50%, C is given for ≥ 60%, D for ≥ 70%, and HD for only 80% of the points. Come up with a pessimistic, optimistic, and a best-shot estimate and then compare your result to the true corresponding scores listed in the grades_scores[121] dataset.*

12.4.5 Measuring Association (*)

Let us consider Australian Bureau of Statistics' National Health Survey 2018[122] data on the prevalence of certain conditions as a function of age. Here is the extracted contingency table:

```
l = [
    ["Arthritis", "Asthma", "Back problems", "Cancer (malignant neoplasms)",
     "Chronic obstructive pulmonary disease", "Diabetes mellitus",
```

(continues on next page)

[120] https://raw.githubusercontent.com/gagolews/teaching_data/master/marek/grades_results.txt
[121] https://raw.githubusercontent.com/gagolews/teaching_data/master/marek/grades_scores.txt
[122] https://www.abs.gov.au/statistics/health/health-conditions-and-risks/national-health-survey-first-results/2017-18

(continued from previous page)

```
     "Heart, stroke and vascular disease", "Kidney disease",
     "Mental and behavioural conditions", "Osteoporosis"],
    ["15-44", "45-64", "65+"]
]
v = 1000*np.array([
    [ 360.2,     1489.0,      1772.2],
    [1069.7,      741.9,       433.7],
    [1469.6,     1513.3,       955.3],
    [  28.1,      162.7,       237.5],
    [ 103.8,      207.0,       251.9],
    [ 135.4,      427.3,       607.7],
    [  94.0,      344.4,       716.0],
    [  29.6,       67.7,       123.3],
    [2218.9,     1390.6,       725.0],
    [  36.1,      312.3,       564.7],
]).astype(int)
pd.DataFrame(v, index=l[0], columns=l[1])
##                                           15-44    45-64      65+
## Arthritis                                360000  1489000  1772000
## Asthma                                  1069000   741000   433000
## Back problems                           1469000  1513000   955000
## Cancer (malignant neoplasms)              28000   162000   237000
## Chronic obstructive pulmonary disease    103000   207000   251000
## Diabetes mellitus                        135000   427000   607000
## Heart, stroke and vascular disease        94000   344000   716000
## Kidney disease                            29000    67000   123000
## Mental and behavioural conditions       2218000  1390000   725000
## Osteoporosis                              36000   312000   564000
```

Cramer's *V* is one of a few ways to measure the degree of association between two categorical variables. It is equal to 0 (lowest possible value) if the two variables are independent (there is no association between them) and 1 (highest possible value) if they are tied.

```
scipy.stats.contingency.association(v)
## 0.316237999724298
```

The above means that there is small association between age and the prevalence of certain conditions (in other words, it might be the case that some conditions are more prevalent in different age groups than others).

It is defined via:

$$\chi^2 = \sum_{i=1}^{n} \sum_{j=1}^{m} \frac{(v_{i,j} - v_{i,\cdot} v_{\cdot,j}/c)^2}{v_{i,\cdot} v_{\cdot,j}/c},$$

where $c = \sum_{i=1}^{n} \sum_{j=1}^{m} v_{i,j}$, $v_{i,\cdot} = \sum_{j=1}^{m} v_{i,j}$, $v_{\cdot,j} = \sum_{i=1}^{n} v_{i,j}$, and then

$$V = \sqrt{\frac{\chi^2/c}{\min\{n-1, m-1\}}}.$$

Note that $v_{i,j}$ is what we observe and $v_{i,\cdot} v_{\cdot,j}/c$ denotes what we would expect if the two variables were independent.

Example 12.19 *(*) Here is a way to compute Cramer's* V *manually:*

```
e = np.sum(v, axis=1).reshape(-1, 1)*np.sum(v, axis=0).reshape(1, -1)/np.
↪sum(v)
chisq = np.sum((v - e)**2/e)
np.sqrt(chisq/np.sum(v)/(np.min(v.shape)-1))
## 0.316237999724298
```

The result is of course equivalent to the one above.

Exercise 12.20 *(*) Take a look at* Table 19: Comorbidity of selected chronic conditions *of the National Health Survey 2018*[123]*, where we clearly see that many disorders co-occur. Visualise them on some heatmaps and bar plots (including data grouped by sex and age).*

12.5 Exercises

Exercise 12.21 *Does it make sense to compute the arithmetic mean of a categorical variable?*

Exercise 12.22 *Name the basic use cases for categorical data.*

Exercise 12.23 *(*) What is a Pareto chart?*

Exercise 12.24 *How to deal with the case of the mode being nonunique?*

Exercise 12.25 *What is the meaning of* **numpy.mean**`((x > 0) & (x < 1))`*, where* x *is a numeric vector?*

[123] https://www.abs.gov.au/statistics/health/health-conditions-and-risks/national-health-survey-first-results/2017-18

Exercise 12.26 *What is the meaning of the sum and mean for binary data (logical vectors)?*

Exercise 12.27 *List some ways to visualise multidimensional categorical data.*

Exercise 12.28 *(*) How is Cramer's* V *defined and what values does it take?*

13

⨯ Processing Data in Groups

Note: ⨯ This part is going to be significantly extended and more chapters will follow in the future.

In particular, we will introduce some basic methods related to:

- clustering,
- classification (predictive modelling with a categorical variable being the target one).

Let's consider (again) a subset of the US Centres for Disease Control and Prevention National Health and Nutrition Examination Survey data, this time featuring some body measures (P_BMX[124]) and demographics (P_DEMO[125]).

```
nhanes = pd.read_csv("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/nhanes_p_demo_bmx_2020.csv",
    comment="#")
nhanes = (
    nhanes
    .loc[
        (nhanes.loc[:, "DMDBORN4"] <= 2) & (nhanes.loc[:, "RIDAGEYR"] >= 18),
        ["RIDAGEYR", "BMXWT", "BMXHT", "BMXBMI", "RIAGENDR", "DMDBORN4"]
    ]
    .rename({
        "RIDAGEYR": "age",
        "BMXWT": "weight",
        "BMXHT": "height",
        "BMXBMI": "bmival",
        "RIAGENDR": "gender",
        "DMDBORN4": "usborn"
```

(continues on next page)

[124] https://wwwn.cdc.gov/Nchs/Nhanes/2017-2018/P_BMX.htm
[125] https://wwwn.cdc.gov/Nchs/Nhanes/2017-2018/P_DEMO.htm

(continued from previous page)

```
    }, axis=1)
    .dropna()
    .reset_index(drop=True)
)

nhanes.loc[:, "usborn"] = nhanes.loc[:, "usborn"].astype("category").\
    cat.rename_categories(["yes", "no"]).astype("str")

nhanes.loc[:, "gender"] = nhanes.loc[:, "gender"].astype("category").\
    cat.rename_categories(["male", "female"]).astype("str")

nhanes.loc[:, "bmicat"] = pd.cut(nhanes.loc[:, "bmival"],
    bins=[0, 18.5, 25, 30, np.inf],
    labels=["underweight", "normal", "overweight", "obese"]
)

nhanes.head()
##    age  weight  height  bmival  gender usborn      bmicat
## 0   29    97.1   160.2    37.8  female     no       obese
## 1   49    98.8   182.3    29.7    male    yes  overweight
## 2   36    74.3   184.2    21.9    male    yes      normal
## 3   68   103.7   185.3    30.2    male    yes       obese
## 4   76    83.3   177.1    26.6    male    yes  overweight
```

We consider only the adult (at least 18 years old) participants, whose country of birth (the US or not) is well-defined.

We have a mix of categorical (gender, US born-ness, BMI category) and numerical (age, weight, height, BMI) variables. Unless we had encoded qualitative variables as integers, this would not be possible with plain matrices.

In this section, we will treat the qualitative columns as grouping variables, so that we can, e.g., summarise or visualise the data *in each group* separately, because it is likely that data distributions vary across different factor levels. This is much like having many data frames stored in one object.

`nhanes` is thus an example of heterogeneous data at their best.

13.1 The groupby Methods

DataFrame and Series objects are equipped with the **groupby** methods, which assist in performing a wide range of popular operations in data groups defined by one or more data frame columns (compare [[Wic11]]).

They return objects of class DataFrameGroupBy and SeriesGroupby:

```
type(nhanes.groupby("gender"))
## <class 'pandas.core.groupby.generic.DataFrameGroupBy'>
type(nhanes.groupby("gender").height)  # or (...)["height"]
## <class 'pandas.core.groupby.generic.SeriesGroupBy'>
```

Important: DataFrameGroupBy and SeriesGroupBy inherit from (extend) the GroupBy class, hence they have many methods in common. Knowing that they are separate types is useful in exploring the list of possible methods and slots in the **pandas** manual.

Exercise 13.1 *Skim through the documentation[126] of the said classes.*

For example, the size method determines the number of observations in each group:

```
nhanes.groupby("gender").size()
## gender
## female    4514
## male      4271
## dtype: int64
```

This returns an object of type Series.

Another example, this time with grouping with respect to a combination of levels in two qualitative columns:

```
nhanes.groupby(["gender", "bmicat"]).size()
## gender  bmicat
## female  underweight      93
##         normal         1161
##         overweight     1245
```

(continues on next page)

[126] https://pandas.pydata.org/pandas-docs/stable/reference/groupby.html

(continued from previous page)

```
##          obese          2015
## male    underweight      65
##          normal         1074
##          overweight     1513
##          obese          1619
## dtype: int64
```

This yielded a `Series` with a hierarchical index (row labels). We can always use **`reset_index`** to convert it to standalone columns:

```
nhanes.groupby(["gender", "bmicat"]).size().rename("counts").reset_index()
##    gender       bmicat  counts
## 0  female  underweight      93
## 1  female       normal    1161
## 2  female   overweight    1245
## 3  female        obese    2015
## 4    male  underweight      65
## 5    male       normal    1074
## 6    male   overweight    1513
## 7    male        obese    1619
```

Note the **`rename`** part thanks to which we have obtained a readable column name.

Exercise 13.2 Unstack *the above data frame (i.e., convert it from the* long *to the* wide *format).*

Exercise 13.3 *(*) Note the difference between* ***`pandas.GroupBy.count`*** *and* ***`pandas.GroupBy.size`*** *methods.*

13.1.1 Aggregating Data in Groups

The `DataFrameGroupBy` and `SeriesGroupBy` classes are equipped with a number of well-known aggregation functions, for example:

```
nhanes.groupby("gender").mean().reset_index()
##    gender        age     weight      height     bmival
## 0  female  48.956580  78.351839  160.089189  30.489189
## 1    male  49.653477  88.589932  173.759541  29.243620
```

Note that the arithmetic mean was computed only on numeric columns. Further, a few common aggregates are generated by **`describe`**:

```
nhanes.groupby("gender").height.describe().reset_index()
##    gender   count        mean       std    min    25%    50%    75%    max
## 0  female  4514.0  160.089189  7.035483  131.1  155.3  160.0  164.8  189.3
## 1    male  4271.0  173.759541  7.702224  144.6  168.5  173.8  178.9  199.6
```

But we can always apply a custom list of functions using **aggregate**:

```
(
    nhanes.
    loc[:, ["gender", "height", "weight"]].
    groupby("gender").
    aggregate([np.mean, np.median, len]).
    reset_index()
)
##    gender      height                      weight
##                  mean median   len        mean median   len
## 0  female  160.089189  160.0  4514  78.351839   74.1  4514
## 1    male  173.759541  173.8  4271  88.589932   85.0  4271
```

Note that the result's `columns` slot is a hierarchical index.

Note: (*) Own (anonymous) functions can be generated on the fly using the so-called *lambda expressions*:

```
(
    nhanes.
    loc[:, ["gender", "height", "weight"]].
    groupby("gender").
    aggregate(lambda x: (np.max(x)-np.min(x))/2).
    reset_index()
)
##    gender  height  weight
## 0  female    29.1  110.85
## 1    male    27.5  102.90
```

Note that the syntax is **`lambda`** `argument_name: return_value`.

Note: (**) The column names in the output object are generated by reading the applied functions' `__name__` slots, see, e.g., **`print`**`(np.mean.__name__)`.

```
mr = lambda x: (np.max(x)-np.min(x))/2
mr.__name__ = "midrange"
(
    nhanes.
    loc[:, ["gender", "height", "weight"]].
    groupby("gender").
    aggregate([np.mean, mr]).
    reset_index()
)
##    gender      height                weight
##                  mean midrange        mean midrange
## 0  female  160.089189     29.1   78.351839   110.85
## 1    male  173.759541     27.5   88.589932   102.90
```

13.1.2 Transforming Data in Groups

We can easily transform individual columns relative to different data groups by means of the **transform** method for `GroupBy` objects.

```
def standardise(x):
    return (x-np.mean(x, axis=0))/np.std(x, axis=0, ddof=1)

nhanes["height_std"] = (
    nhanes.
    loc[:, ["height", "gender"]].
    groupby("gender").
    transform(standardise)
)
(
    nhanes.
    loc[:, ["gender", "height", "height_std"]].
    groupby("gender").
    aggregate([np.mean, np.std])
)
##             height                height_std
##               mean       std          mean  std
## gender
## female  160.089189  7.035483 -1.353518e-15  1.0
## male    173.759541  7.702224  3.155726e-16  1.0
```

The new column gives the *relative* z-scores: a woman with relative z-score of 0 has height of 160.1 cm, whereas a man with the same z-score has height of 173.8 cm.

Exercise 13.4 *Create a data frame comprised of 5 tallest men and 5 tallest women.*

13.2 Plotting Data in Groups

The **seaborn** package is particularly convenient for plotting grouped data – it's highly interoperable with **pandas**.

13.2.1 Series of Box Plots

For example, Figure 13.1 depicts a boxplot with four boxes side by side:

```
sns.boxplot(x="bmival", y="gender", hue="usborn", data=nhanes)
plt.show()
```

Let us contemplate for a while how easy it is now to compare the BMI distribution in different groups. Note that we have two grouping variables, as specified by the y and hue arguments.

Exercise 13.5 *Create a similar series of violin plots.*

Exercise 13.6 *Add the average BMIs in each group to the above boxplot.*

13.2.2 Series of Bar Plots

In Figure 13.2, on the other hand, we have a bar plot representing a two way contingency table (obtained in a different way than in Chapter 12):

```
sns.barplot(
    y="counts",
    x="gender",
    hue="bmicat",
    data=(
        nhanes.
        groupby(["gender", "bmicat"]).
        size().
        rename("counts").
```

(continues on next page)

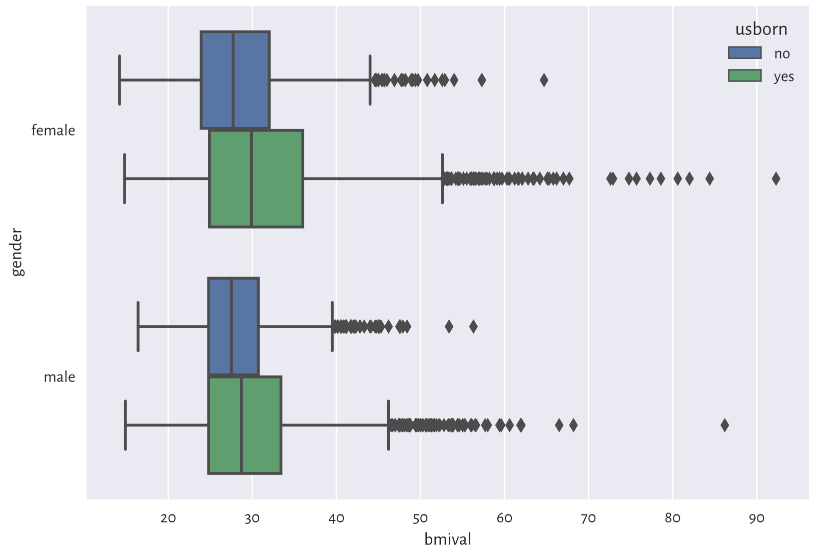


Figure13.1: The distribution of BMIs for different genders and countries of birth

(continued from previous page)

```
        reset_index()
    )
)
plt.show()
```

Exercise 13.7 *Draw a similar bar plot where the bar heights sum to 100% for each gender.*

13.2.3 Scatterplots with Group Information

Scatterplots for grouped data can display category information using points of different shapes or colours, compare Figure 13.3.

```
sns.scatterplot(x="height", y="weight", hue="gender", data=nhanes, alpha=0.4)
plt.show()
```

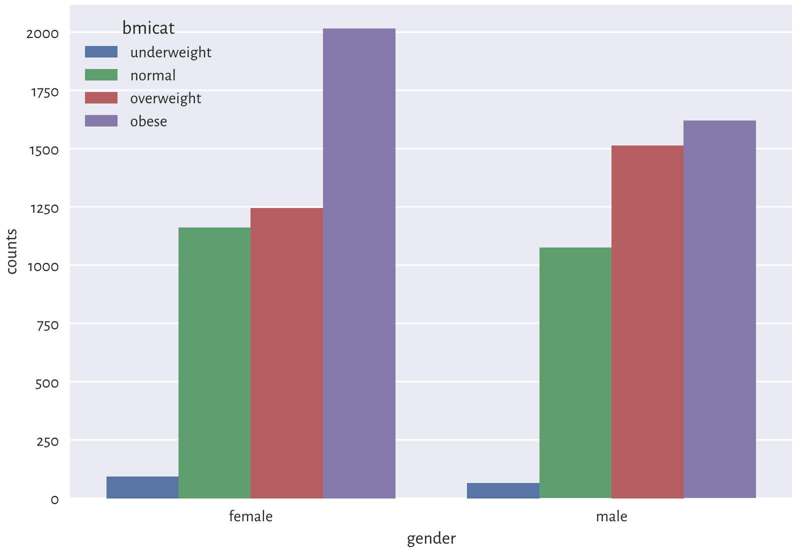


Figure13.2: Number of persons for each gender and BMI category

13.2.4 Grid (Trellis) Plots

Grid plot (also known as trellis, panel, or lattice plots) are a way to visualise data separately for each factor level. All the plots share the same coordinate ranges which makes them easily comparable. For instance, Figure 13.4 depicts a series of histograms for weights grouped by a combination of two categorical variables.

```
grid = sns.FacetGrid(nhanes, col="gender", row="usborn")
grid.map(sns.histplot, "weight", stat="density")
# plt.show()  # not required...
```

Exercise 13.8 *Pass* `hue="bmicat"` *additionally to* ***`seaborn.FacetGrid`***.

Important: Grid plots can feature any kind of data visualisation we have discussed so far (e.g., histograms, bar plots, scatterplots).

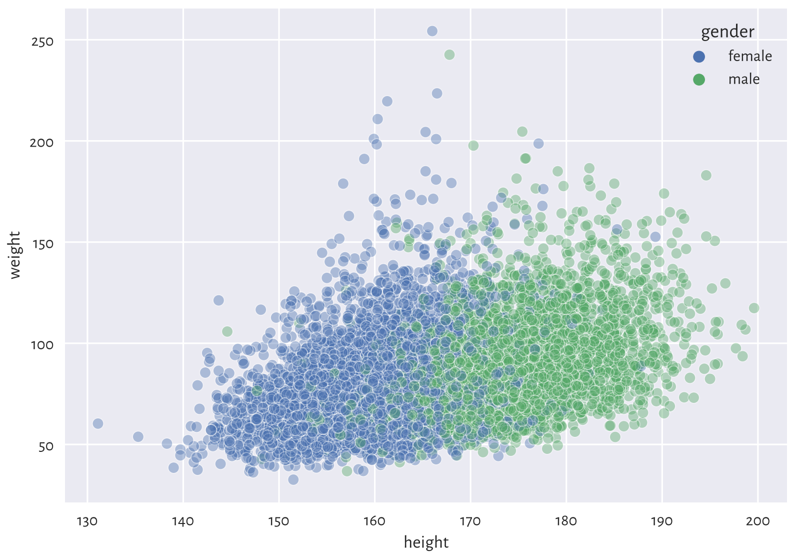


Figure13.3: Weight vs height grouped by gender

Exercise 13.9 *Draw a trellis plot featuring scatterplots of weight vs height grouped by BMI category and gender.*

13.3 Manual Splitting Into Subgroups (**)

Let us demonstrate how easily the splitting can be done manually without the use of **pandas**, which in some cases can be limiting. Calling **numpy.split**(arr, ind) returns a list with arr (being an array-like object, e.g., a matrix, a vector, or a data frame) split rowwisely into **len**(ind)+1 chunks at indices given by ind.

For example:

```
np.split(np.arange(10)*10, [3, 7])
## [array([ 0, 10, 20]), array([30, 40, 50, 60]), array([70, 80, 90])]
```

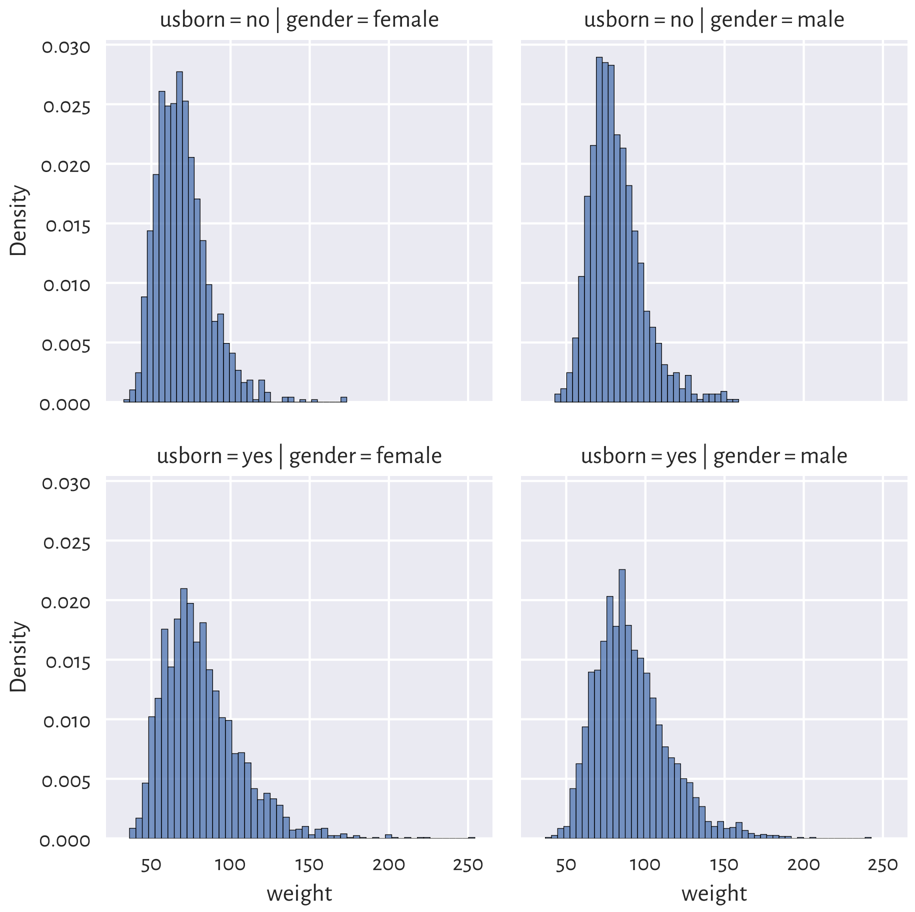


Figure13.4: Distribution of weights for different genders and countries of birth

To split a data frame into groups defined by a categorical column, we can first sort it with respect to the criterion of interest, for instance, the `gender` data:

```
nhanes_srt = nhanes.sort_values("gender", kind="stable")
```

Then, we can use **`numpy.unique`** to fetch the indices of first occurrences of each series of identical labels:

```
levels, where = np.unique(nhanes_srt.loc[:, "gender"], return_index=True)
levels, where
## (array(['female', 'male'], dtype=object), array([   0, 4514]))
```

This can now be used for dividing the sorted data frame into chunks:

```
nhanes_grp = np.split(nhanes_srt, where[1:])
```

We obtained a list of data frames split at rows specified by `where[1:]`. Here is a preview of the first and the last row in each chunk:

```
for i in range(len(levels)):
    print(f"level='{levels[i]}'; preview:")
    print(nhanes_grp[i].iloc[ [0, -1], : ])
    print("")
## level='female'; preview:
##        age  weight  height  bmival  gender usborn bmicat  height_std
## 0       29    97.1   160.2    37.8  female     no  obese    0.015750
## 8781    67    82.8   147.8    37.9  female     no  obese   -1.746744
##
## level='male'; preview:
##        age  weight  height  bmival gender usborn      bmicat  height_std
## 1       49    98.8   182.3    29.7   male    yes  overweight    1.108830
## 8784    74    59.7   167.5    21.3   male     no      normal   -0.812693
```

We can apply any operation on each subgroup we have learned so far, our imagination is the only limiting factor, e.g., aggregate some columns:

```
nhanes_agg = [
    dict(
        level=t.loc[:, "gender"].iloc[0],
        height_mean=np.mean(t.loc[:, "height"]),
        weight_mean=np.mean(t.loc[:, "weight"])
    )
    for t in nhanes_grp
]
nhanes_agg
## [{'level': 'female', 'height_mean': 160.0891891891892, 'weight_mean': 78.
↪35183872396988}, {'level': 'male', 'height_mean': 173.75954109107937,
↪'weight_mean': 88.58993210021072}]
```

Finally, the results may be combined, e.g., to form a data frame:

```
pd.DataFrame(nhanes_agg)
##      level  height_mean  weight_mean
## 0  female   160.089189    78.351839
## 1    male   173.759541    88.589932
```

We see that manual splitting is very powerful but quite tedious in case where we would like to perform the basic operations such as computing some basic aggregates. These scenarios are very common, no wonder why the **pandas** developers came up with an additional, convenience interface in the form of the **pandas.DataFrame.groupby** and **pandas.Series.groupby** methods and the `DataFrameGroupBy` and `SeriesGroupby` classes. However, it is still worth knowing the low-level way to perform splitting in case of more ambitious tasks.

Exercise 13.10 *(**) Using* ***numpy.split*** *and* ***matplotlib.pyplot.boxplot****, draw a box-and-whisker plot of heights grouped by BMI category (four boxes side by side).*

Exercise 13.11 *(**) Using* ***numpy.split****, compute the relative z-scores of the height column separately for each BMI category.*

Note: (**) A simple trick to allow grouping with respect to more than one column is to apply **numpy.unique** on a string vector that combines the levels of the grouping variables, e.g., by concatenating them like `nhanes_srt.gender + "___" + nhanes_srt.bmicat` (assuming that `nhanes_srt` is ordered with respect to these two criteria).

13.4 Exercises

Exercise 13.12 *Name the data type of an object that the* ***DataFrame.groupby*** *method returns.*

Exercise 13.13 *What is the relationship between `GroupBy`, `DataFrameGroupBy`, and `SeriesGroupBy`?*

Exercise 13.14 *What are relative z-scores and how can we compute them?*

14

Accessing Databases

pandas is convenient for working with data that fit into memory and which can be stored in individual CSV files. However, larger information banks, possibly in a shared environment, will often be made available to us via relational (structured) databases such us PostgreSQL or MariaDB, or a wide range of commercial products.

Most commonly, we will be using SQL (structured query language) to express the data chunks we need for analysis, fetching them from the database driver, and importing them to a **pandas** data frame to perform the already well-known operations.

Below we give a quick introduction to the basics of SQL using SQLite[127], which is a lightweight, flat-file, and server-less database management system. Overall, SQLite is sensible for data of even hundreds or thousands of gigabytes in size that fit on a single computer's disk (making it a good choice for playing with our own data science projects or prototyping more complex solutions).

Important: The syntax of SQL is very readable – it is modelled after the natural (English) language. In this introduction, we merely assume that we wish to learn how to read the basic SQL queries, not to write our own. The latter should be covered by a separate course on databases.

14.1 Example Database

In the course of this section, we will be working with a simplified data dump of the Q&A site Travel Stack Exchange[128], which we have downloaded[129] on 2017-10-31.

[127] https://sqlite.org
[128] http://travel.stackexchange.com
[129] https://archive.org/details/stackexchange

Exercise 14.1 *Before continuing, make yourself familiar with the Travel Stack Exchange*[130] *site.*

For example, `Tags` gives, amongst others, topic categories (`TagName`) and how many questions mention them (`Count`):

```
Tags = pd.read_csv("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/travel_stackexchange_com_2017/Tags.csv.gz",
    comment="#")
Tags.head(3)
##    Count  ExcerptPostId  Id    TagName  WikiPostId
## 0    104         2138.0   1   cruising      2137.0
## 1     43          357.0   2  caribbean       356.0
## 2     43          319.0   4  vacations       318.0
```

`Users` provides information on the registered users. In particular, the `Id` column uniquely identifies each member.

```
Users = pd.read_csv("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/travel_stackexchange_com_2017/Users.csv.gz",
    comment="#")
Users.head(3)
##    AccountId   Age             CreationDate  ... Reputation  UpVotes  ↪
 ↪Views
## 0       -1.0   NaN  2011-06-21T15:16:44.253  ...        1.0   2472.0    0.
 ↪0
## 1        2.0  40.0  2011-06-21T20:10:03.720  ...      101.0      1.0   31.
 ↪0
## 2     7598.0  32.0  2011-06-21T20:11:02.490  ...      101.0      1.0   14.
 ↪0
##
## [3 rows x 11 columns]
```

`Badges` recall all rewards given to the users (`UserId`) for their engaging in various praiseworthy activities:

```
Badges = pd.read_csv("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/travel_stackexchange_com_2017/Badges.csv.gz",
    comment="#")
Badges.head(3)
##    Class                     Date  Id           Name  TagBased  UserId
```

(continues on next page)

[130] http://travel.stackexchange.com

(continued from previous page)

```
## 0      3  2011-06-21T20:16:48.910   1  Autobiographer     False       2
## 1      3  2011-06-21T20:16:48.910   2  Autobiographer     False       3
## 2      3  2011-06-21T20:16:48.910   3  Autobiographer     False       4
```

`Posts` (note again that `Id` is the unique identifier of each record in every data frame) lists all the questions and answers (the latter don't have `ParentId` set to `NaN`).

```
Posts = pd.read_csv("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/travel_stackexchange_com_2017/Posts.csv.gz",
    comment="#")
Posts.head(3)
##    AcceptedAnswerId  ...  ViewCount
## 0             393.0  ...      419.0
## 1               NaN  ...     1399.0
## 2               NaN  ...        NaN
##
## [3 rows x 17 columns]
```

`Votes` list all the up-votes (`VoteTypeId` equal to 2) and down-votes (`VoteTypeId` of 3) to all the posts. For privacy reasons, all `UserId`s are missing (which is encoded with a not-a-number, compare Chapter 17).

```
Votes = pd.read_csv("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/travel_stackexchange_com_2017/Votes.csv.gz",
    comment="#")
Votes.head(3)
##    BountyAmount              CreationDate  Id  PostId  UserId  VoteTypeId
## 0           NaN  2011-06-21T00:00:00.000   1       1     NaN           2
## 1           NaN  2011-06-21T00:00:00.000   2       1     NaN           2
## 2           NaN  2011-06-21T00:00:00.000   3       2     NaN           2
```

Exercise 14.2 *See the README*[131] *file for the detailed description of each column.*

[131] https://github.com/gagolews/teaching_data/blob/master/travel_stackexchange_com/README.md

14.2 Exporting Data

Let us establish a connection with the to-be SQLite database. In our case, that will be an ordinary file stored on the computer's disk:

```
import tempfile, os.path
dbfile = os.path.join(tempfile.mkdtemp(), "travel.db")
print(dbfile)
## /tmp/tmp5ry9yg4n/travel.db
```

The above gives the file path (compare Section 14.6.1) where the database is going to be stored. We use a randomly generated filename inside the local filesystem's (we are on Linux) temporary directory, `/tmp`, because this is just a simple exercise and we will not be using this database afterwards.

And now:

```
import sqlite3
conn = sqlite3.connect(dbfile)
```

We are thus "connected". The database might now be queried: new tables can be added, new records inserted, and information fetched.

Important: At the end, we should not forget about the call to `conn.close()`.

We already have our data in form of **pandas** data frames, therefore exporting them to the database is straightforward. All we have to do is to perform a bunch of calls to the **pandas.DataFrame.to_sql** method.

```
Tags.to_sql("Tags", conn, index=False)
Users.to_sql("Users", conn, index=False)
Badges.to_sql("Badges", conn, index=False)
Posts.to_sql("Posts", conn, index=False)
Votes.to_sql("Votes", conn, index=False)
```

Note: (*) It is possible to export data that do not fit into memory by reading them in chunks of considerable, but not too large, sizes. In particular **pandas.read_csv** has the `nrows` argument that lets us read a number of rows from a file buffer; see `help("open")`.

Then, `pandas.DataFrame.to_sql(..., if_exists="append")` can be used to append new rows to an existing table.

Exporting data can of course be done without **pandas** as well, e.g., when they are to be fetched from XML or JSON files (compare Section 14.5) and processed manually, row by row. Intermediate-level SQL users can call `conn.execute("CREATE TABLE t...")`, `conn.executemany("INSERT INTO t VALUES(?, ?, ?)", l)` followed by `conn.commit()` to create a new table (here: named `t`) populated by a list of records (e.g., in form of tuples or **numpy** vectors). For more details, see the manual[132] of the **sqlite3** package.

14.3 Exercises on SQL vs pandas

We can use **pandas** to fetch the results generated by executing any SQL query, for example:

```
pd.read_sql_query("""
    SELECT * FROM Tags LIMIT 3
""", conn)
##     Count  ExcerptPostId  Id    TagName  WikiPostId
## 0     104         2138.0   1   cruising      2137.0
## 1      43          357.0   2  caribbean       356.0
## 2      43          319.0   4  vacations       318.0
```

Fetches all columns (`SELECT *`) and the first 3 rows (`LIMIT 3`) from the `Tags` table.

Exercise 14.3 *For the above and all the following SQL queries, write the equivalent Python code using* ***pandas*** *functions and methods that generates exactly the same result. In each case, there might be more than one equally fine solution.*

Note again that the SQL syntax is very readable, hence no explanations should be necessary for the queries below. In case of any doubt, though, refer to the manual[133].

Example solutions are provided at the end of this section.

Example 14.4 *For instance, here is the reference result:*

```
res1a = pd.read_sql_query("""
```

(continues on next page)

[132] https://docs.python.org/3/library/sqlite3.html
[133] https://sqlite.org/lang.html

(continued from previous page)

```
    SELECT * FROM Tags LIMIT 3
""", conn)
```

Our example equivalent **pandas** *implementation of the above operation might look like:*

```
res1b = Tags.head(3)
```

To verify if the results are identical, we can call:

```
pd.testing.assert_frame_equal(res1a, res1b)
```

Sometimes, however the results generated by **pandas** *will be the same up to the reordering of rows. In such a case, before calling* **pandas.testing.assert_frame_equal***, it might thus be a good idea to call* `DataFrame.sort_values` *on both data frames to sort them with respect to 1 or 2 chosen columns. The good thing about the* **assert_frame_equal** *function is that it ignores small round-off errors introduced by some some operations.*

14.3.1 Filtering

Exercise 14.5 *Fetch all tag data for tags whose name contain* `"europe"` *as substring:*

```
res2a = pd.read_sql_query("""
    SELECT * FROM Tags WHERE TagName LIKE '%europe%'
""", conn)
res2a
##    Count  ExcerptPostId    Id          TagName  WikiPostId
## 0    681          286.0    30           europe       285.0
## 1     27         2384.0   174    eastern-europe      2383.0
## 2     12         6403.0   512  southeast-europe      6402.0
## 3     20         4513.0   669    central-europe      4512.0
## 4     36        16392.0  1950    western-europe     16391.0
```

Hint: use `Tags.TagName.str.contains("europe")` *to obtain the same result with* **pandas***.*

Exercise 14.6 *From* `Tags`*, select two columns* `TagName` *and* `Count` *and rows for which* `TagName` *is equal to one of the three choices provided.*

```
res3a = pd.read_sql_query("""
    SELECT TagName, Count
    FROM Tags
    WHERE TagName IN ('poland', 'australia', 'china')
```

(continues on next page)

(continued from previous page)

```
""", conn)
res3a
##      TagName  Count
## 0      china    443
## 1  australia    411
## 2     poland    139
```

Hint: use **pandas.Series.isin**.

Exercise 14.7 *Select a number of columns from* `Posts` *whose rows fulfil given conditions:*

```
res4a = pd.read_sql_query("""
    SELECT Title, Score, ViewCount, FavoriteCount
    FROM Posts
    WHERE PostTypeId=1 AND
        ViewCount>=10000 AND
        FavoriteCount BETWEEN 35 AND 100
""", conn)
res4a
##                                                Title  Score  ViewCount 
 ↪FavoriteCount
## 0  When traveling to a country with a different c...    136    16838.0   
 ↪       35.0
## 1          How can I do a "broad" search for flights?     95    33554.0   
 ↪       49.0
## 2  Tactics to avoid getting harassed by corrupt p...    156    13220.0   
 ↪       42.0
## 3  Flight tickets: buy two weeks before even duri...    109    49440.0   
 ↪       36.0
## 4  OK we're all adults here, so really, how on ea...    306    73808.0   
 ↪       79.0
## 5  How to intentionally get denied entry to the U...    219    63873.0   
 ↪       53.0
## 6  How do you know if Americans genuinely/literal...    254    30312.0   
 ↪       79.0
## 7  OK, we are all adults here, so what is a bidet...    140    52265.0   
 ↪       38.0
## 8          How to cope with too slow Wi-Fi at hotel?     82    16380.0   
 ↪       41.0
```

14.3.2 Ordering

Exercise 14.8 *Select the* `Title` *and* `Score` *columns from* `Posts` *where* `ParentId` *is missing (i.e., the post is actually a question) and* `Title` *is well-defined; then, sort the result by the* `Score` *column, decreasingly (descending order); finally, return only first 5 rows (e.g., top 5 scoring questions):*

```
res5a = pd.read_sql_query("""
    SELECT Title, Score
    FROM Posts
    WHERE ParentId IS NULL AND Title IS NOT NULL
    ORDER BY Score DESC
    LIMIT 5
""", conn)
res5a
##                                                Title  Score
## 0  OK we're all adults here, so really, how on ea...    306
## 1  How do you know if Americans genuinely/literal...    254
## 2  How to intentionally get denied entry to the U...    219
## 3  Why are airline passengers asked to lift up wi...    210
## 4                         Why prohibit engine braking?    178
```

Hint: use **`pandas.DataFrame.sort_values`** *and* **`numpy.isnan`** *or* **`pandas.isnull`**.

14.3.3 Removing Duplicates

Exercise 14.9 *Get all unique badge names for the user with* `Id=23`.

```
res6a = pd.read_sql_query("""
    SELECT DISTINCT Name
    FROM Badges
    WHERE UserId=23
""", conn)
res6a
##               Name
## 0        Supporter
## 1          Student
## 2          Teacher
## 3          Scholar
## 4             Beta
## 5    Nice Question
## 6           Editor
```

(continues on next page)

(continued from previous page)

```
## 7          Nice Answer
## 8             Yearling
## 9     Popular Question
## 10          Taxonomist
## 11    Notable Question
```

Hint: use **`pandas.DataFrame.drop_duplicates`**.

Exercise 14.10 *For each badge given to the user with* `Id=23`*, get the year it was given and store it in a new column named* `Year`*. Then, select are unique pairs* `(Name, Year)`.

```
res7a = pd.read_sql_query("""
    SELECT DISTINCT
        Name,
        CAST(strftime('%Y', Date) AS FLOAT) AS Year
    FROM Badges
    WHERE UserId=23
""", conn)
res7a
##                  Name    Year
## 0           Supporter  2011.0
## 1             Student  2011.0
## 2             Teacher  2011.0
## 3             Scholar  2011.0
## 4                Beta  2011.0
## 5       Nice Question  2011.0
## 6              Editor  2012.0
## 7         Nice Answer  2012.0
## 8            Yearling  2012.0
## 9       Nice Question  2012.0
## 10      Nice Question  2013.0
## 11           Yearling  2013.0
## 12   Popular Question  2014.0
## 13           Yearling  2014.0
## 14         Taxonomist  2014.0
## 15   Notable Question  2015.0
## 16      Nice Question  2017.0
```

Hint: use `Badges.Date.astype("datetime64").dt.strftime("%Y").astype("float")`.

14.3.4 Grouping and Aggregating

Exercise 14.11 *Count how many badges of each type the user with `Id=23` has received. Also, for each badge type, return the minimal, average, and maximal receiving year. Return only the top 4 badges (with respect to the counts).*

```
res8a = pd.read_sql_query("""
    SELECT
        Name,
        COUNT(*) AS Count,
        MIN(CAST(strftime('%Y', Date) AS FLOAT)) AS MinYear,
        AVG(CAST(strftime('%Y', Date) AS FLOAT)) AS MeanYear,
        MAX(CAST(strftime('%Y', Date) AS FLOAT)) AS MaxYear
    FROM Badges
    WHERE UserId=23
    GROUP BY Name
    ORDER BY Count DESC, MeanYear ASC
    LIMIT 4
""", conn)
res8a
##                 Name  Count  MinYear  MeanYear  MaxYear
## 0      Nice Question      4   2011.0   2013.25   2017.0
## 1           Yearling      3   2012.0   2013.00   2014.0
## 2   Popular Question      3   2014.0   2014.00   2014.0
## 3   Notable Question      2   2015.0   2015.00   2015.0
```

Exercise 14.12 *Similarly to the above, but now count how many unique combinations of pairs (`Name, Date`) are there; then, return only the rows having `Count` greater than 1 and order the results by `Count` decreasingly.*

```
res9a = pd.read_sql_query("""
    SELECT
        Name,
        CAST(strftime('%Y', Date) AS FLOAT) AS Year,
        COUNT(*) AS Count
    FROM Badges
    WHERE UserId=23
    GROUP BY Name, Year
    HAVING Count > 1
    ORDER BY Count DESC
""", conn)
```

(continues on next page)

(*continued from previous page*)

```
res9a
##               Name    Year  Count
## 0  Popular Question  2014.0      3
## 1  Notable Question  2015.0      2
```

Note that `WHERE` *is performed before* `GROUP BY`*, and* `HAVING` *is applied thereafter.*

14.3.5 Joining

Exercise 14.13 *Join (merge)* `Tags`*,* `Posts`*, and* `Users` *for all posts with* `OwnerUserId` *not equal to* `-1` *(i.e., the tags which were created by “alive” users). Return the top 6 records with respect to* `Tags.Count`*.*

```
res10a = pd.read_sql_query("""
    SELECT Tags.TagName, Tags.Count, Posts.OwnerUserId,
        Users.Age, Users.Location, Users.DisplayName
    FROM Tags
    JOIN Posts ON Posts.Id=Tags.WikiPostId
    JOIN Users ON Users.AccountId=Posts.OwnerUserId
    WHERE OwnerUserId != -1
    ORDER BY Tags.Count DESC, Tags.TagName ASC
    LIMIT 6
""", conn)
res10a
##          TagName  Count  OwnerUserId   Age          Location      
↪DisplayName
## 0         canada    802        101.0  34.0     Mumbai, India            
↪hitec
## 1         europe    681        583.0  35.0  Philadelphia, PA       Adam
↪Tuttle
## 2  visa-refusals    554       1737.0  34.0      New York, NY  Benjamin
↪Pollack
## 3      australia    411        101.0  34.0     Mumbai, India            
↪hitec
## 4             eu    204        583.0  35.0  Philadelphia, PA       Adam
↪Tuttle
## 5  new-york-city    204        101.0  34.0     Mumbai, India            
↪hitec
```

Exercise 14.14 *First, create an auxiliary (temporary) table named* `UpVotesTab`*, where we store* *the information about the number of up-votes (`VoteTypeId=2`) that each post has received. Then, join (merge) this table with `Posts` and fetch some details about the 5 questions (`PostTypeId=1`) with the most up-votes:*

```
res11a = pd.read_sql_query("""
    SELECT UpVotesTab.*, Posts.Title FROM
    (
        SELECT PostId, COUNT(*) AS UpVotes
        FROM Votes
        WHERE VoteTypeId=2
        GROUP BY PostId
    ) AS UpVotesTab
    JOIN Posts ON UpVotesTab.PostId=Posts.Id
    WHERE Posts.PostTypeId=1
    ORDER BY UpVotesTab.UpVotes DESC LIMIT 5
""", conn)
res11a
##    PostId  UpVotes                                              Title
## 0    3080      307  OK we're all adults here, so really, how on ea...
## 1   38177      254  How do you know if Americans genuinely/literal...
## 2   24540      221  How to intentionally get denied entry to the U...
## 3   20207      211  Why are airline passengers asked to lift up wi...
## 4   96447      178                       Why prohibit engine braking?
```

14.3.6 Solutions to Exercises

Below are example solutions to the above exercises.

Example 14.15

```
res2b = (
    Tags.
    loc[Tags.TagName.str.contains("europe"), :].
    reset_index(drop=True)
)
res2b
##    Count  ExcerptPostId    Id          TagName  WikiPostId
## 0    681          286.0    30           europe       285.0
## 1     27         2384.0   174    eastern-europe      2383.0
## 2     12         6403.0   512  southeast-europe      6402.0
## 3     20         4513.0   669    central-europe      4512.0
## 4     36        16392.0  1950    western-europe     16391.0
pd.testing.assert_frame_equal(res2a, res2b)
```

Example 14.16

```
res3b = (
    Tags.
    loc[
        Tags.TagName.isin(["poland", "australia", "china"]),
        ["TagName", "Count"]
    ].
    reset_index(drop=True)
)
res3b
##      TagName  Count
## 0      china    443
## 1  australia    411
## 2     poland    139
pd.testing.assert_frame_equal(res3a, res3b)
```

Example 14.17

```
res4b = (
    Posts.
    loc[
        (Posts.PostTypeId == 1) & (Posts.ViewCount >= 10000) &
        (Posts.FavoriteCount >= 35) & (Posts.FavoriteCount <= 100),
        ["Title", "Score", "ViewCount", "FavoriteCount"]
    ].
    reset_index(drop=True)
)
res4b
##                                                 Title  Score  ViewCount  
↪FavoriteCount
## 0  When traveling to a country with a different c...    136    16838.0   
↪         35.0
## 1          How can I do a "broad" search for flights?     95    33554.0   
↪         49.0
## 2  Tactics to avoid getting harassed by corrupt p...    156    13220.0   
↪         42.0
## 3  Flight tickets: buy two weeks before even duri...    109    49440.0   
↪         36.0
## 4  OK we're all adults here, so really, how on ea...    306    73808.0   
↪         79.0
## 5  How to intentionally get denied entry to the U...    219    63873.0   
↪         53.0
## 6  How do you know if Americans genuinely/literal...    254    30312.0   
↪         79.0
```

(continues on next page)

(continued from previous page)

```
## 7  OK, we are all adults here, so what is a bidet...    140    52265.0  
↪      38.0
## 8           How to cope with too slow Wi-Fi at hotel?     82    16380.0  
↪      41.0
pd.testing.assert_frame_equal(res4a, res4b)
```

Example 14.18

```
res5b = (
    Posts.
    loc[
        Posts.ParentId.isna() & (~Posts.Title.isna()),
        ["Title", "Score"]
    ].
    sort_values("Score", ascending=False).
    head(5).
    reset_index(drop=True)
)
res5b
##                                                Title  Score
## 0  OK we're all adults here, so really, how on ea...    306
## 1  How do you know if Americans genuinely/literal...    254
## 2  How to intentionally get denied entry to the U...    219
## 3  Why are airline passengers asked to lift up wi...    210
## 4                       Why prohibit engine braking?    178
pd.testing.assert_frame_equal(res5a, res5b)
```

Example 14.19

```
res6b = (
    Badges.
    loc[Badges.UserId == 23, ["Name"]].
    drop_duplicates().
    reset_index(drop=True)
)
res6b
##                 Name
## 0          Supporter
## 1            Student
## 2            Teacher
## 3            Scholar
## 4               Beta
## 5       Nice Question
```

(continues on next page)

(continued from previous page)

```
## 6              Editor
## 7         Nice Answer
## 8            Yearling
## 9    Popular Question
## 10         Taxonomist
## 11  Notable Question
pd.testing.assert_frame_equal(res6a, res6b)
```

Example 14.20

```
Badges2 = Badges.copy()
Badges2.loc[:, "Year"] = (
    Badges2.Date.astype("datetime64").dt.strftime("%Y").astype("float")
)
res7b = (
    Badges2.
    loc[Badges2.UserId == 23, ["Name", "Year"]].
    drop_duplicates().
    reset_index(drop=True)
)
res7b
##                 Name    Year
## 0          Supporter  2011.0
## 1            Student  2011.0
## 2            Teacher  2011.0
## 3            Scholar  2011.0
## 4               Beta  2011.0
## 5      Nice Question  2011.0
## 6             Editor  2012.0
## 7        Nice Answer  2012.0
## 8           Yearling  2012.0
## 9      Nice Question  2012.0
## 10     Nice Question  2013.0
## 11          Yearling  2013.0
## 12  Popular Question  2014.0
## 13          Yearling  2014.0
## 14        Taxonomist  2014.0
## 15  Notable Question  2015.0
## 16     Nice Question  2017.0
pd.testing.assert_frame_equal(res7a, res7b)
```

Example 14.21

```
Badges2 = Badges.copy()
Badges2.loc[:, "Year"] = (
    Badges2.Date.astype("datetime64").dt.strftime("%Y").astype("float")
)
res8b = (
    Badges2.
    loc[Badges2.UserId == 23, ["Name", "Year"]].
    groupby("Name")["Year"].
    aggregate([len, np.min, np.mean, np.max]).
    sort_values(["len", "mean"], ascending=[False, True]).
    head(4).
    reset_index()
)
res8b.columns = ["Name", "Count", "MinYear", "MeanYear", "MaxYear"]
res8b
##                 Name  Count  MinYear  MeanYear  MaxYear
## 0      Nice Question      4   2011.0   2013.25   2017.0
## 1           Yearling      3   2012.0   2013.00   2014.0
## 2   Popular Question      3   2014.0   2014.00   2014.0
## 3   Notable Question      2   2015.0   2015.00   2015.0
```

Note that had we not converted `Year` to `float`, we would obtain meaningless average year, without any warning.

```
pd.testing.assert_frame_equal(res8a, res8b)
```

Example 14.22

```
Badges2 = Badges.copy()
Badges2.loc[:, "Year"] = (
    Badges2.Date.astype("datetime64").dt.strftime("%Y").astype("float")
)
res9b = (
    Badges2.
    loc[ Badges2.UserId == 23, ["Name", "Year"] ].
    groupby(["Name", "Year"]).
    size().
    rename("Count").
    reset_index()
)
res9b = (
    res9b.
```

(continues on next page)

(continued from previous page)

```
    loc[ res9b.Count > 1, : ].
    sort_values("Count", ascending=False).
    reset_index(drop=True)
)
res9b
##                Name    Year  Count
## 0  Popular Question  2014.0      3
## 1  Notable Question  2015.0      2
```

Note that the HAVING part is performed after WHERE and GROUP BY.

```
pd.testing.assert_frame_equal(res9a, res9b)
```

Example 14.23

```
res10b = pd.merge(Posts, Tags, left_on="Id", right_on="WikiPostId")
res10b = pd.merge(Users, res10b, left_on="AccountId", right_on="OwnerUserId")
res10b = (
    res10b.
    loc[
        (res10b.OwnerUserId != -1) & (~res10b.OwnerUserId.isna()),
        ["TagName", "Count", "OwnerUserId", "Age", "Location", "DisplayName"]
    ].
    sort_values(["Count", "TagName"], ascending=[False, True]).
    head(6).
    reset_index(drop=True)
)
res10b
##          TagName  Count  OwnerUserId   Age          Location      
 ↪DisplayName
## 0         canada    802        101.0  34.0     Mumbai, India            
 ↪hitec
## 1         europe    681        583.0  35.0  Philadelphia, PA       Adam
 ↪Tuttle
## 2   visa-refusals    554       1737.0  34.0      New York, NY  Benjamin
 ↪Pollack
## 3      australia    411        101.0  34.0     Mumbai, India            
 ↪hitec
## 4             eu    204        583.0  35.0  Philadelphia, PA       Adam
 ↪Tuttle
## 5   new-york-city    204        101.0  34.0     Mumbai, India            
 ↪hitec
```

Note that in SQL, "not equals to -1" implies that "IS NOT NULL".

```
pd.testing.assert_frame_equal(res10a, res10b)
```

Example 14.24 *To obtain `res11b`, we first create an auxiliary data frame that corresponds to the subquery.*

```
UpVotesTab = (
    Votes.
    loc[Votes.VoteTypeId==2, :].
    groupby("PostId").
    size().
    rename("UpVotes").
    reset_index()
)
```

And now:

```
res11b = pd.merge(UpVotesTab, Posts, left_on="PostId", right_on="Id")
res11b = (
    res11b.
    loc[res11b.PostTypeId==1, ["PostId", "UpVotes", "Title"]].
    sort_values("UpVotes", ascending=False).
    head(5).
    reset_index(drop=True)
)
res11b
##    PostId  UpVotes                                              Title
## 0    3080      307  OK we're all adults here, so really, how on ea...
## 1   38177      254  How do you know if Americans genuinely/literal...
## 2   24540      221  How to intentionally get denied entry to the U...
## 3   20207      211  Why are airline passengers asked to lift up wi...
## 4   96447      178                       Why prohibit engine braking?
pd.testing.assert_frame_equal(res11a, res11b)
```

14.4 Closing the Database Connection

We have said we should not forget about:

```
conn.close()
```

This gives some sense of closure. Such a relief.

14.5 Common Data Serialisation Formats for the Web

CSV files are an all-round way to exchange *tabular* data between different programming and data analysis environments.

For unstructured or non-tabularly-structured data, XML as well as JSON (and its superset, YAML) are the common formats of choice, especially for communicating with different Web APIs.

Important: It is recommended that we solve some of the following exercises to make sure we can fetch data in these formats. Warning: often, this will quite tedious labour, neither art nor science; see also [[vdLdJ18]] and [[DJ03]]. Good luck.

Exercise 14.25 *Consider the Web API for accessing[134] the on-street parking bay sensor data in Melbourne, VIC, Australia. Using, for example, the* **`json`** *package, convert the data[135] in the JSON format to a* **`pandas`** *data frame.*

Exercise 14.26 *Australian Radiation Protection and Nuclear Safety Agency publishes[136] UV data for different Aussie cities. Using, for example, the* **`xml`** *package, convert this XML dataset[137] to a* **`pandas`** *data frame.*

Exercise 14.27 *(*) Check out the English Wikipedia article featuring a list of 20th-century classical composers[138]. Using* **`pandas.read_html`***, convert the* Climate Data *table included therein to a data frame.*

Exercise 14.28 *(*) Using, for example, the* **`lxml`** *package, write a function that converts each bullet list featured in a given Wikipedia article (e.g., this one[139]), to a list of strings.*

[134] https://data.melbourne.vic.gov.au/Transport/On-street-Parking-Bay-Sensors/vh2v-4nfs
[135] https://data.melbourne.vic.gov.au/resource/vh2v-4nfs.json
[136] https://www.arpansa.gov.au/our-services/monitoring/ultraviolet-radiation-monitoring/ultraviolet-radation-data-information
[137] https://uvdata.arpansa.gov.au/xml/uvvalues.xml
[138] https://en.wikipedia.org/wiki/List_of_20th-century_classical_composers
[139] https://en.wikipedia.org/wiki/Category:Fr%C3%A9d%C3%A9ric_Chopin

Exercise 14.29 *(**) Import an archived version of a Stack Exchange*[140] *site that you find interesting and store it in an SQLite database. You can find the relevant data dumps here*[141].

Exercise 14.30 *(**) Download*[142] *and then import an archived version of one of the wikis hosted by the Wikimedia Foundation*[143] *(e.g., the whole English Wikipedia) so that it can be stored in an SQLite database.*

14.6 Working with Many Files

For the mass-processing of many files, it is worth knowing *of* the most basic functions for dealing with file paths, searching for files, etc. Usually we will be looking up ways to complete specific tasks at hand, e.g., how to read data from a ZIP archive, on the internet.

14.6.1 File Paths

UNIX-like operating systems, including GNU/Linux and macOS, use slashes, `/`, as path separators, e.g., `"/home/marek/file.csv"`. Windows, however, uses backslashes, `\`, which have a special meaning in character strings (escape sequences, see Section 2.1.3), therefore they should be input as, e.g., `"c:\\users\\marek\\file.csv"` or `r"c:\users\marek\file.csv"` (note the `r` prefix).

When constructing file paths programmatically, it is thus best to rely on **`os.path.join`**, which takes care of the system-specific nuances.

```
import os.path
os.path.join("~", "Desktop", "file.csv")  # we're on GNU/Linux
## '~/Desktop/file.csv'
```

Note that the tilde, `~`, denotes the current user's home directory.

Note: For storing auxiliary data, we will often be using the system's temporary directory. See the **`tempfile`** module for functions that generate appropriate file paths therein.

[140] https://stackexchange.com/
[141] https://archive.org/details/stackexchange
[142] https://meta.wikimedia.org/wiki/Data_dumps
[143] https://wikimediafoundation.org/

For instance, a subdirectory inside the temporary directory can be created via a call to **`tempfile.mkdtemp`**.

Important: We will frequently be referring to file paths relative to the working directory of the currently executed Python session (e.g., from which IPython/Jupyter notebook server was started). The latter can be read by calling **`os.getcwd`**.

And thus, all non-absolute file names (ones that do not start with `` `~` ``, `` `/` ``, `` `c:\\` ``, and the like), for example, `"filename.csv"` or **`os.path.join`**`("subdir", "filename.csv")` are always relative to the current working directory.

For instance, if the working directory is `"/home/marek/projects/python"`, then `"filename.csv"` refers to `"/home/marek/projects/python/filename.csv"`.

Also, `` `..` `` denotes the current working directory's parent directory. Thus, `"../filename2.csv"` resolves to `"/home/marek/projects/filename2.csv"`.

Exercise 14.31 *Print the result output by **`os.getcwd`**. Next, download the file air_quality_2018_param.csv[144] and save it in the current Python session's working directory (e.g., in your web browser, right-click on the web page's canvas and select* Save Page As… *). Load it as a **`pandas`** data frame by passing simply `"air_quality_2018_param.csv"` as the input path.*

Exercise 14.32 *(*) Download the aforementioned file programmatically (if you have not done so) using the **`requests`** module.*

14.6.2 File Search

`glob.glob` and **`os.listdir`** can generate a list of files in a given directory (and possibly all its subdirectories).

`os.path.isdir` and **`os.path.isfile`** can be used to determine the type of a given object in the file system.

Exercise 14.33 *Write a function that computes the total size of all the files in a given directory and all its subdirectories.*

[144] https://raw.githubusercontent.com/gagolews/teaching_data/master/marek/air_quality_2018_param.csv

14.7 Exercises

Exercise 14.34 *Find an example of an XML and JSON file. Which one is more human-readable? Do they differ in terms of capabilities?*

Exercise 14.35 *What is wrong with constructing file paths like* `"~" + "\\" + "filename.csv"`*?*

Exercise 14.36 *What are the benefits of using a SQL-supporting relational database management system in data science activities?*

Exercise 14.37 *(**) How to populate a database with gigabytes of data read from many CSV files?*

Part V

Text and Other Data

15

Working with Text Data

In [[Gag22]] it is noted that *effective processing of character strings is required at various stages of data analysis pipelines: from data cleansing and preparation, through information extraction, to report generation. Pattern searching, string collation and sorting, normalisation, transliteration, and formatting are ubiquitous in text mining, natural language processing, and bioinformatics.* Means for the handling of string data *should be included in each statistician's or data scientist's repertoire to complement their numerical computing and data wrangling skills.*

Diverse data cleansing and preparation operations (compare, e.g., [[vdLdJ18]] and [[DJ03]]) *need to be applied before an analyst can begin to enjoy an orderly and meaningful data frame, matrix, or spreadsheet being finally at their disposal. Activities related to information retrieval, computer vision, bioinformatics, natural language processing, or even musicology can also benefit from including them in data processing pipelines.*

In this part we discuss the most basic string operations in base Python, together with their vectorised versions in **numpy** and **pandas**.

15.1 String Operations in Base Python

Recall that the `str` class represents individual character strings:

```
x = "spam"
type(x)
## <class 'str'>
```

There are a few binary operators overloaded for strings, e.g., `+` stands for string concatenation:

```
x + " and eggs"
## 'spam and eggs'
```

`*` duplicates a given string:

```
x * 3
## 'spamspamspam'
```

Further, `str` is a sequential type, therefore we can extract individual code points and create substrings using the index operator:

```
x[-1]  # last letter
## 'm'
```

Recall that strings are immutable. However, parts of strings can always be reused in conjunction with the concatenation operator:

```
x[:2] + "ecial"
## 'special'
```

15.1.1 Unicode as the Universal Encoding

It is worth knowing that all strings in Python (from version 3.0) use Unicode(https://www.unico which is a universal encoding capable of representing ca. 150,000 characters covering letters and numbers in contemporary and historic alphabets/scripts, mathematical, political, phonetic, and other symbols, emojis, etc. It is thus a very powerful representation.

Note: Despite the wide support for Unicode, sometimes our own or other readers' display (e.g., web browsers when viewing a HTML version of the output report) might not be able to *render* all code points properly (e.g., due to missing fonts). However, we should rest assured that they are still there, and are processed correctly if string functions are applied thereon.

Note: (**) More precisely, Python strings are UTF-8[145]-encoded. Most web pages and API data are nowadays served in UTF-8. However, occasionally we can encounter files encoded in ISO-8859-1 (Western Europe), Windows-1250 (Eastern Europe), Windows-1251 (Cyrillic), GB18030 and Big5 (Chinese), EUC-KR (Korean), Shift-JIS and EUC-JP (Japanese), amongst others; they can be converted using the **`str.decode`** method.

[145] https://www.ietf.org/rfc/rfc3629.txt

15.1.2 Normalising Strings

Dirty text data are a pain, especially if similar (semantically) tokens are encoded in many different ways. For the sake of string matching, we might want, e.g., the German `"groß"`, `"GROSS"`, and `"  gross  "` to compare all equal.

`str.strip` removes whitespaces (spaces, tabs, newline characters) at both ends of strings (see also **`str.lstrip`** and **`str.rstrip`** for their nonsymmetric versions).

`str.lower` and **`str.upper`** change letter case. For caseless comparison/matching, **`str.casefold`** might be a slightly better option as it unfolds many more code point sequences:

```
"Groß".lower(), "Groß".upper(), "Groß".casefold()
## ('groß', 'GROSS', 'gross')
```

Important: (**) More advanced string transliteration can be performed by means of the **`ICU`**[146] library, which the **`PyICU`** package provides wrappers for.

For instance, converting all code points to ASCII (English) might be necessary when identifiers are expected to miss some diacritics that would normally be included (as in `"Gągolewski"` vs `"Gagolewski"`):

```
import icu  # PyICU package
icu.Transliterator.createInstance("Lower; Any-Latin; Latin-ASCII").
↪transliterate(
    "Χαίρετε! Groß gżegżółka — © La Niña – köszönöm – Gągolewski"
)
## 'chairete! gross gzegzolka - (C) la nina - koszonom - gagolewski'
```

Converting between different Unicode Normalisation Forms[147] (also available in the **`unicodedata`** package and via **`pandas.Series.str.normalize`**) might be used for the removal of some formatting nuances:

```
icu.Transliterator.createInstance("NFKD; NFC").transliterate("¼ąr²︷")
## '¼ąr2{'
```

---

[146] https://icu.unicode.org/

[147] https://www.unicode.org/faq/normalization.html

15.1.3 Substring Searching and Replacing

Determining if a string features a particular fixed substring can be done in a number of different ways.

For instance:

```
food = "bacon, spam, spam, eggs, and spam"
"spam" in food
## True
```

verifies whether a particular substring exists,

```
food.count("spam")
## 3
```

counts the number of occurrences of a substring,

```
food.find("spam")
## 7
```

locates the first pattern occurrence (see also **`str.rfind`** as well as **`str.index`** and **`str.rindex`**),

```
food.replace("spam", "veggies")
## 'bacon, veggies, veggies, eggs, and veggies'
```

replaces matching substrings with another string.

Exercise 15.1 *Read the manual of the following methods:* ***`str.startswith`****,* ***`str.endswith`*** ***`str.removeprefix`****,* ***`str.removesuffix`****.*

The splitting of long strings at specific fixed delimiter strings can be done via:

```
food.split(", ")
## ['bacon', 'spam', 'spam', 'eggs', 'and spam']
```

see also **`str.partition`**. The **`str.join`** method implements the inverse operation:

```
", ".join(["spam", "bacon", "eggs", "spam"])
## 'spam, bacon, eggs, spam'
```

Important: In Chapter 16, we will discuss pattern matching with regular expressions, which can be useful in, amongst others, extracting more abstract data chunks (numbers, URLs, email addresses, IDs) from within strings.

15.1.4 Locale-Aware Services in ICU (**)

Recall that relational operators such as `<` and `>=` perform lexicographic comparing of strings:

```
"spam" > "egg"
## True
```

We have: `"a"` < `"aa"` < `"aaaaaaaaaaaa"` < `"ab"` < `"aba"` < `"abb"` < `"b"` < `"ba"` < `"baaaaaa"` < `"bb"` < `"Spanish inquisition"`.

Lexicographic ordering (character-by-character, from left to right) is, however, not necessarily appropriate for strings featuring numerals:

```
"a9" < "a123"
## False
```

Also, it only takes into account the numeric codes[148] corresponding to each Unicode character, therefore does not work well with non-English alphabets:

```
"MIELONECZKĄ" < "MIELONECZKI"
## False
```

In Polish, *A with ogonek* (Ą) should sort after *A* and before *B*, let alone *I*. However, their corresponding numeric codes in the Unicode table are: 260 (Ą), 65 (A), 66 (B), and 73 (I), therefore the resulting ordering is incorrect, natural language processing-wisely.

It is best to perform string collation using the services provided by **ICU**. Here is an example of German phone book-like collation where `"ö"` is treated the same as `"oe"`:

```
c = icu.Collator.createInstance(icu.Locale("de_DE@collation=phonebook"))
c.setStrength(0)
c.compare("Löwe", "loewe")
## 0
```

A result of `0` means that the strings are deemed equal.

[148] https://www.unicode.org/charts/

In some languages, contractions occur, e.g., in Slovak and Czech, two code points "ch" are treated as a single entity and are sorted after "h":

```
icu.Collator.createInstance(icu.Locale("sk_SK")).compare("chladný", "hladný")
## 1
```

i.e., we have "chladný" > "hladný" (the 1st argument is greater than the 2nd one). Compare the above to something similar in Polish:

```
icu.Collator.createInstance(icu.Locale("pl_PL")).compare("chłodny", "hardy")
## -1
```

i.e., "chłodny" < "hardy" (the first argument is less than the 2nd one).

Also, with **ICU**, numeric collation is possible:

```
c = icu.Collator.createInstance()
c.setAttribute(icu.UCollAttribute.NUMERIC_COLLATION, icu.UCollAttributeValue.
↪ON)
c.compare("a9", "a123")
## -1
```

Which is the correct result: "a9" is less than "a123" (compare the above to the example which was using using `<`).

15.2 String Operations in pandas

String sequences in Series are by default[149] using the broadest possible object data type:

```
pd.Series(["spam", "bacon", "spam"])
## 0     spam
## 1    bacon
## 2     spam
## dtype: object
```

which basically means that we deal with a sequence of Python objects of arbitrary type (here, are all of class str). This allows for the encoding of missing values by means of the None object.

[149] https://pandas.pydata.org/pandas-docs/stable/user_guide/text.html

Vectorised versions of base string operations are available[150] via the **pandas.Series.str** accessor, which we usually refer to by calling `x.str.method_name()`, for instance:

```
x = pd.Series(["spam", "bacon", None, "buckwheat", "spam"])
x.str.upper()
## 0         SPAM
## 1        BACON
## 2         None
## 3    BUCKWHEAT
## 4         SPAM
## dtype: object
```

We thus have **pandas.Series.str.strip**, **pandas.Series.str.split**, **pandas.Series.str.find**, and so forth.

But there is more. For example, a function to compute the length of each string:

```
x.str.len()
## 0    4.0
## 1    5.0
## 2    NaN
## 3    9.0
## 4    4.0
## dtype: float64
```

Concatenating all items into a single string:

```
x.str.cat(sep="; ")
## 'spam; bacon; buckwheat; spam'
```

Vectorised string concatenation:

```
x + " and spam"
## 0         spam and spam
## 1        bacon and spam
## 2                   NaN
## 3    buckwheat and spam
## 4         spam and spam
## dtype: object
```

Conversion to numeric:

[150] https://numpy.org/doc/stable/reference/routines.char.html

```
pd.Series(["1.3", "-7", None, "3523"]).astype(float)
## 0       1.3
## 1      -7.0
## 2       NaN
## 3    3523.0
## dtype: float64
```

Selecting substrings:

```
x.str.slice(2, -1)  # like x.iloc[i][2:-1] for all i
## 0          a
## 1         co
## 2       None
## 3     ckwhea
## 4          a
## dtype: object
```

Replacing substrings:

```
x.str.slice_replace(0, 2, "tofu")  # like x.iloc[i][2:-1] = "tofu"
## 0          tofuam
## 1         tofucon
## 2            None
## 3     tofuckwheat
## 4          tofuam
## dtype: object
```

Exercise 15.2 *Consider the* `nasaweather_glaciers`[151] *data frame. All glaciers are assigned 11/12-character unique identifiers. The ID number is assigned to the glacier as defined by the WGMS convention that forms the glacier ID number by combining the following five elements. Extract all of them and store them as independent columns in the data frame.*

1. *2-character political unit,*
2. *1-digit continent code,*
3. *4-character drainage code,*
4. *2-digit free position code,*
5. *2- or 3-digit local glacier code.*

[151] https://github.com/gagolews/teaching_data/blob/master/other/nasaweather_glaciers.csv

15.3 Working with String Lists

`Series` can also consist of lists of strings of varying lengths. They can not only be input manually (via the **pandas.Series** constructor), but also through string splitting. For instance:

```
x = pd.Series([
    "spam",
    "spam, bacon, spam",
    None,
    "spam, eggs, bacon, spam, spam"
])
xs = x.str.split(", ", regex=False)
xs
## 0                             [spam]
## 1                [spam, bacon, spam]
## 2                               None
## 3    [spam, eggs, bacon, spam, spam]
## dtype: object
```

and now, e.g., looking at the last element:

```
xs.iloc[-1]
## ['spam', 'eggs', 'bacon', 'spam', 'spam']
```

reveals that it is indeed a list of strings.

There are a few vectorised operations that enable us to work with such variable length lists, such as concatenating all strings:

```
xs.str.join("; ")
## 0                             spam
## 1                spam; bacon; spam
## 2                             None
## 3    spam; eggs; bacon; spam; spam
## dtype: object
```

selecting, say, the first string in each list:

```
xs.str.get(0)
```

(continues on next page)

(continued from previous page)

```
## 0    spam
## 1    spam
## 2    None
## 3    spam
## dtype: object
```

or slicing:

```
xs.str.slice(0, -1)  # like xs.iloc[i][0:-1] for all i
## 0                            []
## 1                 [spam, bacon]
## 2                          None
## 3    [spam, eggs, bacon, spam]
## dtype: object
```

Exercise 15.3 *(*) Using **pandas.merge**, join the following datasets: countries[152], world_factbook_2020[153] and ssi_2016_dimensions[154] based on the country names. Note that some manual data cleansing will be necessary.*

Exercise 15.4 *(**) Given a `Series` object featuring lists of strings:*

1. *determine the list of all unique strings (e.g., for `xs` above we have: `["spam", "bacon", "eggs"]`), let's call it `xu`;*
2. *create a data frame `x` with `xs.shape[0]` rows and `len(xu)` columns such that `x.iloc[i, j]` is equal to 1 if `xu[j]` is amongst `xs.loc[i]` and equal to 0 otherwise;*
3. *given `x` (and only `x`: neither `xs` nor `xu`), perform the inverse operation.*

15.4 String Operations in numpy (**)

There is a huge overlap between the **numpy** and **pandas** capabilities for string handling, with the latter being more powerful. Still, some readers will find the following useful.

As mentioned in our introduction to **numpy** vectors, objects of type `ndarray` can store not only numeric and logical data, but also character strings. For example:

[152] https://github.com/gagolews/teaching_data/blob/master/other/countries.csv
[153] https://github.com/gagolews/teaching_data/blob/master/marek/world_factbook_2020.csv
[154] https://github.com/gagolews/teaching_data/blob/master/marek/ssi_2016_dimensions.csv

```
x = np.array(["spam", "bacon", "egg"])
x
## array(['spam', 'bacon', 'egg'], dtype='<U5')
```

Here, the data type `"<U5"` (compare also `x.dtype`) means that we deal with Unicode strings of length no greater than 5. Thus, unfortunately, replacing elements with too long a content will result in truncated strings:

```
x[2] = "buckwheat"
x
## array(['spam', 'bacon', 'buckw'], dtype='<U5')
```

In order to remedy this, we first need to recast the vector manually:

```
x = x.astype("<U10")
x[2] = "buckwheat"
x
## array(['spam', 'bacon', 'buckwheat'], dtype='<U10')
```

Conversion from/to numeric is also possible:

```
np.array(["1.3", "-7", "3523"]).astype(float)
## array([ 1.300e+00, -7.000e+00,  3.523e+03])
np.array([1, 3.14, -5153]).astype(str)
## array(['1.0', '3.14', '-5153.0'], dtype='<U32')
```

The **`numpy.char`**[155] module includes a number of vectorised versions of string routines, most of which we have discussed above. For example:

```
x = np.array([
    "spam", "spam, bacon, and spam",
    "spam, eggs, bacon, spam, spam, and spam"
])
np.char.split(x, ", ")
## array([list(['spam']), list(['spam', 'bacon', 'and spam']),
##        list(['spam', 'eggs', 'bacon', 'spam', 'spam', 'and spam'])],
##       dtype=object)
np.char.count(x, "spam")
## array([1, 2, 4])
```

155 https://numpy.org/doc/stable/reference/routines.char.html

Operations that we would normally perform via the use of binary operators (i.e., `+`, `*`, `<`, etc.) are available through standalone functions:

```
np.char.add(["spam", "bacon"], " and spam")
## array(['spam and spam', 'bacon and spam'], dtype='<U14')
np.char.equal(["spam", "bacon", "spam"], "spam")
## array([ True, False,  True])
```

Also the function that returns the length of each string is noteworthy:

```
np.char.str_len(x)
## array([ 4, 21, 39])
```

15.5 Formatted Outputs (*)

When preparing reports from data analysis (e.g., using Jupyter Notebooks or writing directly to Markdown files which we later compile to PDF or HTML using **pandoc**[156]) it is important to be able to output nicely formatted content programmatically.

15.5.1 Formatting Strings

Recall that string formatting for inclusion of data stored in existing objects can easily be done using f-strings (formatted string literals) of the type `f"...{expression}..."`. For instance:

```
pi = 3.14159265358979323846
f"π = {pi:.2f}"
## 'π = 3.14'
```

creates a string including the variable `pi` formatted as a `float` rounded to two places after the decimal separator.

Note: (**) Similar functionality can be achieved using the **`str.format`** method as well as the `%` operator overloaded for strings, which uses **`sprintf`**-like value placeholders known to some readers from other programming languages (such as C):

[156] https://pandoc.org/

```
"π = {:.2f}".format(pi), "π = %.2f" % pi
## ('π = 3.14', 'π = 3.14')
```

15.5.2 str and repr

The **str** and **repr** functions can create string representations of a number of objects, with the former being more human-readable and latter slightly more technical.

```
x = np.array([1, 2, 3])
str(x), repr(x)
## ('[1 2 3]', 'array([1, 2, 3])')
```

Note that **repr** often returns an output that can be interpreted as executable Python code literally (with no or few adjustments, however, **pandas** objects are one of the many exceptions).

15.5.3 Justifying Strings

str.center, **str.ljust**, **str.rjust** can be used to centre, left-, or-right justify a string so that it's of at least given width, which might make the display thereof more aesthetic. Very long strings, possibly containing whole text paragraphs can be dealt with using the **wrap** and **shorten** functions from the **textwrap** package.

15.5.4 Direct Markdown Output in Jupyter

Further, note that with IPython/Jupyter, we can output strings that will be directly interpreted as Markdown-formatted:

```
import IPython.display
x = 2+2
out = f"*Result*: $2^2=2+2={x}$."
IPython.display.Markdown(out)
```

Result: $2^2 = 2 + 2 = 4$.

Recall that Markdown[157] is a very flexible markup language, allowing us to insert itemised and numbered lists, mathematical formulae, tables, images, etc.

[157] https://daringfireball.net/projects/markdown/syntax

15.5.5 Manual Markdown File Output

We can also generate Markdown code programmatically in form of standalone `.md` files:

```
f = open("/tmp/test-report.md", "w")  # open for writing (overwrite if
↪exists)
f.write("**Yummy Foods** include, but are not limited to:\n\n")
x = ["spam", "bacon", "eggs", "spam"]
for e in x:
    f.write(f"* {e}\n")
f.write("\nAnd now for something completely different:\n\n")
f.write("Rank | Food\n")
f.write("-----|-----\n")
for i in range(len(x)):
    f.write(f"{i+1:4} | {x[i][::-1]:10}\n")
f.close()
## 50
## 7
## 8
## 7
## 7
## 46
## 12
## 12
## 18
## 18
## 18
## 18
```

Here is the resulting raw Markdown source file:

```
with open("/tmp/test-report.md", "r") as f:
    out = f.read()
print(out)
## **Yummy Foods** include, but are not limited to:
##
## * spam
## * bacon
## * eggs
## * spam
##
```

(continues on next page)

(continued from previous page)

```
## And now for something completely different:
##
## Rank | Food
## -----|-----
##    1 | maps
##    2 | nocab
##    3 | sgge
##    4 | maps
```

We can run it through the **pandoc**[158] tool to convert it to a number of formats, including HTML, PDF, EPUB, and ODT. We may also render it directly into our report:

```
IPython.display.Markdown(out)
```

Yummy Foods include, but are not limited to:

- spam
- bacon
- eggs
- spam

And now for something completely different:

Rank	Food
1	maps
2	nocab
3	sgge
4	maps

Note: Figures created in **matplotlib** can be exported to PNG, SVG, or PDF files using the **matplotlib.pyplot.savefig** function.

Note: Data frames can be nicely prepared for display in a report using **pandas.DataFrame.to_markdown**.

[158] https://pandoc.org/

15.6 Exercises

Exercise 15.5 *List some ways to normalise character strings.*

Exercise 15.6 *(**) What are the challenges of processing non-English text?*

16

Regular Expressions (*)

Note that this section contains large excerpts from yours truly's other work [[Gag22]].

Regular expressions (regexes) provide us with a concise grammar for defining systematic patterns which can be sought in character strings. Examples of such patterns include: specific fixed substrings, emojis of any kind, stand-alone sequences of lower-case Latin letters ("words"), substrings that can be interpreted as real numbers (with or without fractional parts, also in scientific notation), telephone numbers, email addresses, or URLs.

Theoretically, the concept of regular pattern matching dates back to the so-called regular languages and finite state automata [[Kle51]], see also [[RS59]] and [[HU79]]. Regexes in the form as we know today have already been present in one of the pre-Unix implementations of the command-line text editor **`qed`** *[[RT70]] (the predecessor of the well-known* **`sed`***).*

16.1 Regex Matching with `re`

In Python, the **`re`** module implements a regular expression matching engine that accepts patterns following a similar syntax to the ones available in the Perl language.

Before we proceed with a detailed discussion on how to read and write regexes, let us first review the methods for identifying the matching substrings. Below we use the `r"\bni+\b"` regex as an example, which catches `"n"` followed by at least one `"i"` that begins and ends at a *word boundary*, i.e., which may be considered standalone words.

In particular, **`re.findall`** extracts all non-overlapping matches to a given regex:

```
import re
x = "We're the knights who say ni! niiiii! ni! niiiiiiiii!"
re.findall(r"\bni+\b", x)
## ['ni', 'niiiii', 'ni', 'niiiiiiiii']
```

Note that the order of arguments is *look for what, where*, not the other way around.

Important: Note that we used the `r"..."` prefix when entering a string so that `\b` is not treated as a escape sequence denoting the backspace character. Otherwise, the above would have to be input as `"\\bni+\\b"`.

Hadn't we insisted on matching at the word boundaries (i.e., if we used `"ni+"` instead), we would also match the `"ni"` in `"knights"`.

The **`re.search`** function returns an object of class `re.Match` that enables us to get some more information about the first match:

```
r = re.search(r"\bni+\b", x)
r.start(), r.end(), r.group()
## (26, 28, 'ni')
```

The above includes the start and end position (index) and the match itself. If the regex contains *capture groups* (see below for more details), we can also pinpoint the matches thereto.

Moreover, **`re.finditer`** returns an iterable object that includes the same details, but now about all the matches:

```
rs = re.finditer(r"\bni+\b", x)
for r in rs:
    print((r.start(), r.end(), r.group()))
## (26, 28, 'ni')
## (30, 36, 'niiiii')
## (38, 40, 'ni')
## (42, 52, 'niiiiiiiii')
```

`re.split` divides a string into chunks separated by matches to a given regex:

```
re.split(r"!\s+", x)
## ["We're the knights who say ni", 'niiiii', 'ni', 'niiiiiiiii!']
```

The `r"!\s*"` regex matches the exclamation mark followed by one or more whitespace characters.

`re.sub` replaces each match with a given string:

```
re.sub(r"\bni+\b", "nu", x)
## "We're the knights who say nu! nu! nu! nu!"
```

Note: (**) More flexible replacement strings can be generated by passing a custom function as the second argument:

```
re.sub(r"\bni+\b", lambda m: "n" + "u"*(m.end()-m.start()-1), x)
## "We're the knights who say nu! nuuuuu! nu! nuuuuuuuuu!"
```

16.2 Regex Matching with pandas

The `pandas.Series.str` accessor also defines a number of vectorised functions that utilise the `re` package's matcher.

Example `Series` object:

```
x = pd.Series(["ni!", "niiii, ni, nii!", None, "spam, bacon", "nii, ni!"])
x
## 0                ni!
## 1    niiii, ni, nii!
## 2               None
## 3        spam, bacon
## 4           nii, ni!
## dtype: object
```

Here are the most notable functions; their names are self-explanatory, so let's just get the picture instead of the words' abundance.

```
x.str.contains(r"\bni+\b")
## 0     True
## 1     True
## 2     None
## 3    False
## 4     True
## dtype: object
x.str.count(r"\bni+\b")
```

(continues on next page)

(continued from previous page)

```
## 0    1.0
## 1    3.0
## 2    NaN
## 3    0.0
## 4    2.0
## dtype: float64
x.str.replace(r"\bni+\b", "nu", regex=True)
## 0            nu!
## 1    nu, nu, nu!
## 2           None
## 3    spam, bacon
## 4        nu, nu!
## dtype: object
x.str.findall(r"\bni+\b")
## 0                 [ni]
## 1    [niiii, ni, nii]
## 2                 None
## 3                   []
## 4           [nii, ni]
## dtype: object
x.str.split(r",\s+")  # a comma, one or more whitespaces
## 0                 [ni!]
## 1    [niiii, ni, nii!]
## 2                  None
## 3        [spam, bacon]
## 4           [nii, ni!]
## dtype: object
```

In the two last cases, we get lists of strings in result.

Also, later we will mention **pandas.Series.str.extract** and **pandas.Series.str.extractall** which work with regexes that include capture groups.

Note: (*) If we intend to seek matches to the same pattern in different strings without the use of **pandas**, it might be a good to pre-compile a regex first and then use the **re.Pattern.findall** method instead or **re.findall**:

```
p = re.compile(r"\bni+\b")  # returns an object of class `re.Pattern`
```

(continues on next page)

(continued from previous page)

```
p.findall("We're the knights who say ni! ni! niiiii! nininiiiiiiiii!")
## ['ni', 'ni', 'niiiii']
```

16.3 Matching Individual Characters

Most programming languages and text editors (including **Kate**[159], **Eclipse**[160], and **VSCode**[161]) support finding or replacing patters with regexes. Therefore, they should be amongst the instruments at every data scientist's disposal. One general introduction to regexes is [[Fri06]]. The **re** module flavour is summarised in the official manual[162], see also [[Kuc22]]. In the following sections we review the most important elements of the regex syntax as we did in [[Gag22]].

We begin by discussing different ways to define character sets. In this part, determining the length of all matching substrings will be quite straightforward.

Important: The following characters have special meaning to the regex engine:

```
. \ | ( ) [ ] > { } ^ $ * + ?
```

Any regular expression that contains none of the above behaves like a fixed pattern:

```
re.findall("spam", "spam, eggs, spam, bacon, sausage, and spam")
## ['spam', 'spam', 'spam']
```

159 https://kate-editor.org/
160 https://www.eclipse.org/ide/
161 https://code.visualstudio.com/
162 https://docs.python.org/3/library/re.html

There are hence 3 occurrences of a pattern that is comprised of 4 code points, “s” followed by “p”, then by “a”, and ending with “m”.

If we wish to include a special character as part of a regular expression – so that it is treated literally – we’ll need to escape it with a backslash, “\”.

```
re.findall(r"\.", "spam...")
## ['.', '.', '.']
```

16.3.1 Matching Any Character

The (unescaped) dot, “.”, matches any code point except the newline.

```
x = "Spam, ham,\njam, SPAM, eggs, and spam"
re.findall("..am", x, re.IGNORECASE)
## ['Spam', ' ham', 'SPAM', 'spam']
```

The above matches non-overlapping length-4 substrings that end with “am”, case insensitively.

The dot’s insensitivity to the newline character is motivated by the need to maintain the compatibility with tools such as **grep** (when searching within text files in a line-by-line manner). This behaviour can be altered by setting the DOTALL flag.

```
re.findall("..am", x, re.DOTALL|re.IGNORECASE)
## ['Spam', ' ham', '\njam', 'SPAM', 'spam']
```

16.3.2 Defining Character Sets

Sets of characters can be introduced by enumerating their members within a pair of square brackets. For instance, “[abc]” denotes the set *{a, b, c}* – such a regular expression matches one (and only one) symbol from this set. Moreover, in:

```
re.findall("[hj]am", x)
## ['ham', 'jam']
```

the “[hj]am” regex matches: “h” or “j”, followed by “a”, followed by “m”. In other words, "ham" and "jam" are the only two strings that are matched by this pattern (unless matching is done case-insensitively).

Important: The following characters, if used within square brackets, may be treated non-literally:

```
\ [ ] ^ - & ~ |
```

Therefore, to include them as-is in a character set, the backslash-escape must be used. For example, “`[\[\]\\]`” matches a backslash or a square bracket.

16.3.3 Complementing Sets

Including “`^`” (the caret) after the opening square bracket denotes the set complement. Hence, “`[^abc]`” matches any code point except “`a`”, “`b`”, and “`c`”. Here is an example where we seek any substring that consists of 3 non-spaces.

```
x = "Nobody expects the Spanish Inquisition!"
re.findall("[^ ][^ ][^ ]", x)
## ['Nob', 'ody', 'exp', 'ect', 'the', 'Spa', 'nis', 'Inq', 'uis', 'iti',
↪'on!']
```

16.3.4 Defining Code Point Ranges

Each Unicode code point can be referenced by its unique numeric identifier for more details. For instance, “`a`” is assigned code U+0061 and “`z`” is mapped to U+007A. In the pre-Unicode era (mostly with regards to the ASCII codes, ≤ U+007F, representing English letters, decimal digits, some punctuation characters, and a few control characters), we were used to relying on specific code ranges; e.g., “`[a-z]`” denotes the set comprised of all characters with codes between U+0061 and U+007A, i.e., lowercase letters of the English (Latin) alphabet.

```
re.findall("[0-9A-Za-z]", "Gągolewski")
## ['G', 'g', 'o', 'l', 'e', 'w', 's', 'k', 'i']
```

The above pattern denotes a union of 3 code ranges: digits and ASCII upper- and lowercase letters.

Nowadays, in the processing of text in natural languages, this notation should rather be avoided. Note the missing “ą” (Polish “a” with ogonek) in the result.

16.3.5 Using Predefined Character Sets

Some other noteworthy Unicode-aware code point classes include the “word characters”:

```
x = "ąąbßÆAĄB□12□□□,.;'! \t-+=\n[]©□□”„"
re.findall(r"\w", x)
## ['a', 'ą', 'b', 'ß', 'Æ', 'A', 'Ą', 'B', '□', '1', '2', '□', '□', '□']
```

decimal digits:

```
re.findall(r"\d", x)
## ['1', '2', '□', '□', '□']
```

and whitespaces:

```
re.findall(r"\s", x)
## [' ', '\t', '\n']
```

Moreover, e.g., “`\W`” is equivalent to “`[^\w]`” , i.e., denotes its complement.

16.4 Alternating and Grouping Subexpressions

16.4.1 Alternation Operator

The alternation operator, “`|`” (the pipe or bar), matches either its left or its right branch, for instance:

```
x = "spam, egg, ham, jam, algae, and an amalgam of spam, all al dente"
re.findall("spam|ham", x)
## ['spam', 'ham', 'spam']
```

16.4.2 Grouping Subexpressions

"`|`" has a very low precedence. Therefore, if we wish to introduce an alternative of subexpressions, we need to group them using the "`(?:...)`" syntax. For instance, "`(?:sp|h)am`" matches either "`spam`" or "`ham`".

Note that the bare use of the round brackets, `"(...)"` (i.e., without the `"?:"`) part, has the side-effect of creating new capturing groups, see below for more details.

Also, matching is always done left-to-right, on a first-come, first-served basis. Hence, if the left branch is a subset of the right one, the latter will never be matched. In particular, "`(?:al|alga|algae)`" can only match "`al`". To fix this, we can write "`(?:algae|alga|al)`".

16.4.3 Non-grouping Parentheses

Some parenthesised subexpressions – those in which the opening bracket is followed by the question mark – have a distinct meaning. In particular, "`(?#...)`" denotes a free-format comment that is ignored by the regex parser:

```
re.findall(
  "(?# match 'sp' or 'h')(?:sp|h)(?# and 'am')am|(?# or match 'egg')egg",
  x
)
## ['spam', 'egg', 'ham', 'spam']
```

This is just horrible. Luckily, constructing more sophisticated regexes by concatenating subfragments thereof is more readable:

```
re.findall(
       "(?:sp|h)" +    # match either 'sp' or 'h'
       "am" +          # followed by 'am'
    "|" +         # ... or ...
       "egg",          # just match 'egg'
    x
)
## ['spam', 'egg', 'ham', 'spam']
```

What is more, e.g., "`(?i)`" enables the case-insensitive mode.

```
re.findall("(?i)spam", "Spam spam SPAMITY spAm")
## ['Spam', 'spam', 'SPAM', 'spAm']
```

16.5 Quantifiers

More often than not, a variable number of instances of the same subexpression needs to be captured or its presence should be made optional. This can be achieved by means of the following quantifiers:

- "?" matches 0 or 1 times;
- "*" matches 0 or more times;
- "+" matches 1 or more times;
- "`{n,m}`" matches between `n` and `m` times;
- "`{n,}`" matches at least `n` times;
- "`{n}`" matches exactly `n` times.

These operators are applied onto the directly preceding atoms. For example, "`ni+`" captures `"ni"`, `"nii"`, `"niii"`, etc., but neither `"n"` alone nor `"ninini"` altogether.

By default, the quantifiers are greedy – they match the repeated subexpression as many times as possible. The "?" suffix (hence, quantifiers such as "`??`", "`*?`", "`+?`", and so forth) tries with as few occurrences as possible (to obtain a match still).

Greedy:

```
x = "sp(AM)(maps)(SP)am"
re.findall(r"\(.+\)", x)
## ['(AM)(maps)(SP)']
```

Lazy:

```
re.findall(r"\(.+?\)", x)
## ['(AM)', '(maps)', '(SP)']
```

Greedy (but clever):

```
re.findall(r"\([^)]+\)", x)
## ['(AM)', '(maps)', '(SP)']
```

The first regex is greedy: it matches an opening bracket, then as many characters as possible (including ")") that are followed by a closing bracket. The two other patterns terminate as soon as the first closing bracket is found.

More examples:

```
x = "spamamamnomnomnomammmmmmmm"
re.findall("sp(?:am|nom)+", x)
## ['spamamamnomnomnomam']
re.findall("sp(?:am|nom)+?", x)
## ['spam']
```

And:

```
re.findall("sp(?:am|nom)+?m*", x)
## ['spam']
re.findall("sp(?:am|nom)+?m+", x)
## ['spamamamnomnomnomammmmmmmm']
```

Let's stress that the quantifier is applied to the subexpression that stands directly before it. Grouping parentheses can be used in case they are needed.

```
x = "12, 34.5, 678.901234, 37...629, ..."
re.findall(r"\d+\.\d+", x)
## ['34.5', '678.901234']
```

matches digits, a dot, and another series of digits.

```
re.findall(r"\d+(?:\.\d+)?", x)
## ['12', '34.5', '678.901234', '37', '629']
```

finds digits which are possibly (but not necessarily) followed by a dot and a digit sequence.

Exercise 16.1 *Write a regex that extracts all #hashtags from a string #omg #SoEasy.*

16.6 Capture Groups and References Thereto

Round-bracketed subexpressions (without the `"?:"` prefix) form the so-called *capture groups* that can be extracted separately or be referred to in other parts of the same regex.

16.6.1 Extracting Capture Group Matches

The above is evident when we use **re.findall**:

```
x = "name='Sir Launcelot', quest='Seek the Grail', favecolour='blue'"
re.findall(r"(\w+)='(.+?)'", x)
## [('name', 'Sir Launcelot'), ('quest', 'Seek the Grail'), ('favecolour',
↪'blue')]
```

Simply returned the matches to the capture groups, not the whole matching substring.

re.find and **re.finditer** can pinpoint each component:

```
r = re.search(r"(\w+)='(.+?)'", x)
print("all (0):", (r.start(), r.end(), r.group()))
print("     1 :", (r.start(1), r.end(1), r.group(1)))
print("     2 :", (r.start(2), r.end(2), r.group(2)))
## all (0): (0, 20, "name='Sir Launcelot'")
##      1 : (0, 4, 'name')
##      2 : (6, 19, 'Sir Launcelot')
```

Here is a vectorised version of the above from **pandas**, returning the first match:

```
y = pd.Series([
    "name='Sir Launcelot'",
    "quest='Seek the Grail'",
    "favecolour='blue', favecolour='yel.. Aaargh!'"
])
y.str.extract(r"(\w+)='(.+?)'")
##             0               1
## 0        name   Sir Launcelot
## 1       quest  Seek the Grail
## 2  favecolour            blue
```

We see that the findings are presented in a data frame form. The first column gives the matches to the first capture group, and so forth.

All matches are available too:

```
y.str.extractall(r"(\w+)='(.+?)'")
##                     0               1
##   match
## 0 0              name   Sir Launcelot
```

(continues on next page)

(continued from previous page)

```
## 1 0          quest   Seek the Grail
## 2 0      favecolour            blue
##   1      favecolour   yel.. Aaargh!
```

Recall that if we just need the grouping part of "`(...)`", i.e., without the capturing feature, "`(?:...)`" can be applied.

Also, named capture groups defined like "`(?P<name>...)`" are supported.

```
y.str.extract("(?:\\w+)='(?P<value>.+?)'")
##               value
## 0   Sir Launcelot
## 1  Seek the Grail
## 2            blue
```

16.6.2 Replacing with Capture Group Matches

Matches to particular capture groups can be recalled in replacement strings when using **re.sub** and **pandas.Series.str.replace**. Here, the match in its entirety is denoted with "`\g<0>`", then "`\g<1>`" stores whatever was caught by the first capture group, "`\g<2>`" is the match to the second capture group, etc.

```
re.sub(r"(\w+)='(.+?)'", r"\g<2> is a \g<1>", x)
## 'Sir Launcelot is a name, Seek the Grail is a quest, blue is a favecolour'
```

Named capture groups can be referred to too:

```
re.sub(r"(?P<key>\w+)='(?P<value>.+?)'",
  r"\g<value> is a \g<key>", x)
## 'Sir Launcelot is a name, Seek the Grail is a quest, blue is a favecolour'
```

16.6.3 Back-Referencing

Matches to capture groups can also be part of the regexes themselves. For example, "`\1`" denotes whatever has been consumed by the first capture group.

Even though, in general, parsing HTML code with regexes is not recommended, let's consider the following examples:

```
x = "<strong><em>spam</em></strong><code>eggs</code>"
re.findall(r"<[a-z]+>.*?</[a-z]+>", x)
## ['<strong><em>spam</em>', '<code>eggs</code>']
re.findall(r"(<([a-z]+)>.*?</\2>)", x)
## [('<strong><em>spam</em></strong>', 'strong'), ('<code>eggs</code>', 'code
↪')]
```

The second regex guarantees that the match will include all characters between the opening `<tag>` and the corresponding (not: any) closing `</tag>`. Named capture groups can be referenced using the `(?P=name)` syntax (the angle brackets are part of the token):

```
re.findall(r"(<(?P<tagname>[a-z]+)>.*?</(?P=tagname)>)", x)
## [('<strong><em>spam</em></strong>', 'strong'), ('<code>eggs</code>', 'code
↪')]
```

16.7 Anchoring

Lastly, let's mention the ways to match a pattern at a given abstract position within a string.

16.7.1 Matching at the Beginning or End of a String

"`^`" and "`$`" match, respectively, start and end of the string (or each line within a string, if the `re.MULTILINE` flag is set).

```
x = pd.Series(["spam egg", "bacon spam", "spam", "egg spam bacon", "sausage
↪"])
rs = ["spam", "^spam", "spam$", "spam$|^spam", "^spam$"]
pd.concat([x.str.contains(r) for r in rs], axis=1, keys=rs)
##     spam  ^spam  spam$  spam$|^spam  ^spam$
## 0   True   True  False         True   False
## 1   True  False   True         True   False
## 2   True   True   True         True    True
## 3   True  False  False        False   False
## 4  False  False  False        False   False
```

The 5 regular expressions match "`spam`", respectively, anywhere within the string, at the beginning, at the end, at the beginning or end, and in strings that are equal to the pattern itself.

Exercise 16.2 *Write a regex that does the same job as* ***`str.strip`***.

16.7.2 Matching at Word Boundaries

Furthermore, "`\b`" matches at a "word boundary", e.g., near spaces, punctuation marks, or at the start/end of a string (i.e., wherever there is a transition between a word, "`\w`", and a non-word character, "`\W`", or vice versa).

In the following example, we match all stand-alone numbers (this regular expression is provided for didactic purposes only):

```
re.findall(r"[-+]?\b\d+(?:\.\d+)?\b", "+12, 34.5, -5.3243")
## ['+12', '34.5', '-5.3243']
```

16.7.3 Looking Behind and Ahead

There are also ways to guarantee that a pattern occurrence begins or ends with a match to some subexpression: "`(?<=...)...`" is the so-called look-behind, whereas "`...(?=...)`" denotes the look-ahead. Moreover, "`(?<!...)...`" and "`...(?!...)`" are their negated ("negative look-behind/ahead") versions.

```
x = "I like spam, spam, eggs, and spam."
re.findall(r"\b\w+\b(?=[,.])", x)
## ['spam', 'spam', 'eggs', 'spam']
re.findall(r"\b\w+\b(?![,.])", x)
## ['I', 'like', 'and']
```

The first regex captures words that end with "`,`" or "`.`". The second one matches words that end neither with "`,`" nor "`.`".

Exercise 16.3 *Write a regex that extracts all standalone numbers accepted by Python, including* `12.123`*,* `-53`*,* `+1e-9`*,* `-1.2423e10`*,* `4.` *and* `.2`*.*

Exercise 16.4 *Write a regex that matches all email addresses.*

Exercise 16.5 *Write a regex that matches all URLs starting with* `http://` *or* `https://`*.*

Exercise 16.6 *Cleanse the* `warsaw_weather`[163] *dataset so that it contains analysable numeric data.*

[163] https://github.com/gagolews/teaching_data/blob/master/marek/warsaw_weather.csv

16.8 Exercises

Exercise 16.7 *What are the problems with the* `"[A-Za-z]"` *and* `"[A-z]"` *character sets?*

Exercise 16.8 *Name the two ways to turn on case-insensitive regex matching.*

Exercise 16.9 *What is a word boundary?*

Exercise 16.10 *What is the difference between the* `"^"` *and* `"$"` *anchors?*

Exercise 16.11 *When we would prefer using* `"[0-9]"` *instead of* `"\d"`*?*

Exercise 16.12 *What is the difference between the* `"?"`, `"??"`, `"*"`, `"*?"`, `"+"`, *and* `"+?"` *quantifiers?*

Exercise 16.13 *Does* `"."` *match all the characters?*

Exercise 16.14 *What are named capture groups and how to refer to the matches thereto in* **`re.sub`***?*

17

Outliers, Missing, Censored, and Incorrect Data

So far we have been assuming that observations are of "decent quality", i.e., trustworthy. It would be nice if in reality that was always the case, but it is not.

In this section we briefly address the most basic methods for dealing with "suspicious" observations: outliers, missing, censored, and incorrect data.

17.1 Missing Data

Consider the `nhanes_p_demo_bmx_2020` dataset being another excerpt from the National Health and Nutrition Examination Survey by the US Centres for Disease Control and Prevention:

```
nhanes = pd.read_csv("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/nhanes_p_demo_bmx_2020.csv",
    comment="#")
nhanes.head(3)
##       SEQN  BMDSTATS  BMXWT  BMIWT  ...       WTMECPRP  SDMVPSU  SDMVSTRA ␣
 ↪INDFMPIR
## 0  109263         4    NaN    NaN  ...    8951.815567        3       156  ␣
 ↪   4.66
## 1  109264         1   42.2    NaN  ...   12271.157043        1       155  ␣
 ↪   0.83
## 2  109265         1   12.0    NaN  ...   16658.764203        1       157  ␣
 ↪   3.06
##
## [3 rows x 50 columns]
```

Some of the columns feature `NaN` (not-a-number) values, which are used here to encode *missing data*.

The reasons behind why some items are missing might be numerous, for instance:

- a participant doesn't know the answer to a given question;
- a patient refused to answer a given question;
- a participant does not take part in the study anymore (attrition, death, etc.);
- an item is not applicable (e.g., number of minutes spent cycling weekly when someone answered they don't like bikes at all);
- a piece of information was not collected, e.g., due to the lack of funding or equipment failure.

17.1.1 Representing and Detecting Missing Values

Sometimes missing values will be specially encoded, especially in CSV files, e.g., with `-1`, `0`, `9999`, `numpy.inf`, `-numpy.inf`, or `None`, strings such as `"NA"`, `"N/A"`, `"Not Applicable"`, `"---"` – we should always inspect our datasets carefully.

Generally, we will be converting them to `NaN` (as in: `numpy.nan`) in numeric (floating-point) columns or to Python's `None` otherwise, to assure consistent representation.

Vectorised functions such as **`numpy.isnan`** (or, more generally, **`numpy.isfinite`**) and **`pandas.isnull`** as well as **`isna`** methods in the `DataFrame` and `Series` classes can be used to verify whether an item is missing or not.

For instance, here are the counts and proportions of missing values in each column (we will display only the top 5 columns to save space):

```
nhanes_na_stats = nhanes.isna().apply([np.sum, np.mean]).T
nhanes_na_stats.nlargest(5, "sum")
##                sum      mean
## BMIHEAD    14300.0  1.000000
## BMIRECUM   14257.0  0.996993
## BMIHT      14129.0  0.988042
## BMXHEAD    13990.0  0.978322
## BMIHIP     13924.0  0.973706
```

Looking at the column descriptions[164], `BMIHEAD` stands for "Head Circumference Comment", whereas `BMXHEAD` is "Head Circumference (cm)", but it is only applicable for infants.

[164] https://wwwn.cdc.gov/Nchs/Nhanes/2017-2018/P_BMX.htm

Exercise 17.1 *Read the column descriptions (refer to the comments in the CSV file for the relevant URLs) to identify the possible reasons for some of the NHANES data being missing.*

Exercise 17.2 *Study the **pandas** manual to learn about the difference between the **DataFrameGroupBy.size** and **DataFrameGroupBy.count** methods.*

17.1.2 Computing with Missing Values

Our use of `NaN` to denote a missing piece of information is actually an ugly (but still functioning) hack. The original use case for not-a-number is to represent the results of incorrect operations, e.g., logarithms of negative numbers or subtracting two infinite entities. Therefore, we need extra care when handling them. On a side note, e.g., the R environment has a built-in, seamless support for missing values.

Generally, arithmetic operations on missing values yield a result that is undefined as well:

```
np.nan + 2  # "don't know" + 2 == "don't know"
## nan
np.mean([1, np.nan, 2, 3])
## nan
```

There are versions of certain aggregation functions that ignore missing values whatsoever: **numpy.nanmean**, **numpy.nanmin**, **numpy.nanmax**, **numpy.nanpercentile**, **numpy.std**, etc.

```
np.nanmean([1, np.nan, 2, 3])
## 2.0
```

However, running aggregation functions directly on `Series` objects ignores missing entities by default. Compare an application of **numpy.mean** on a `Series` instance vs on a vector:

```
x = nhanes.loc[:6, "BMXHT"]  # some example Series, whatever
np.mean(x), np.mean(x.to_numpy())
## (148.5, nan)
```

This is quite an unfortunate behaviour, because this way we might miss (sic!) the presence of missing values. This is why it is very important always to pre-inspect a dataset carefully.

Also, due to `NaN`'s being of floating-point type, it cannot be present in, amongst others, logical vectors. By convention, comparisons against missing values yield `False` (instead of the more semantically valid missing value):

```
x  # preview
## 0      NaN
## 1    154.7
## 2     89.3
## 3    160.2
## 4      NaN
## 5    156.0
## 6    182.3
## Name: BMXHT, dtype: float64
y = (x > 40)
y
## 0    False
## 1     True
## 2     True
## 3     True
## 4    False
## 5     True
## 6     True
## Name: BMXHT, dtype: bool
```

Note: (*) If we want to retain the missingness information (we don't know if a missing value is greater than 40), we need to do it manually:

```
y = y.astype("object")  # required for numpy vectors, not for pandas Series
y[np.isnan(x)] = None
y
## 0    None
## 1    True
## 2    True
## 3    True
## 4    None
## 5    True
## 6    True
## Name: BMXHT, dtype: object
```

Exercise 17.3 *Read the* ***pandas*** *manual*[165] *for more technical details on missing value handling.*

[165] https://pandas.pydata.org/pandas-docs/stable/user_guide/missing_data.html

17.1.3 Missing at Random or Not?

At a general level (from the mathematical modelling perspective), we may distinguish between different missingness patterns (as per Rubin's [[Rub76]]):

- *missing completely at random*: reasons are unrelated to data and probabilities of cases being missing are all the same;
- *missing at random*: there are different probabilities of being missing within different groups (e.g., males might systematically refuse to answer specific questions);
- *missing not at random*: due to reasons unknown to us (e.g., data was collected at different times, there might be systematic differences but within the groups that we cannot easily identify, e.g., amongst participants with data science background where we did not ask about education or occupation).

It is important to try to determine the reason for missingness, because this will usually imply the kinds of techniques that are more or less suitable in specific cases.

17.1.4 Discarding Missing Values

We may try removing (discarding) the rows or columns that feature at least one, some, or too many missing values.

However, for this we need to be "rich enough" – such a scheme will obviously not work for small datasets, where each observation is precious (but on the other hand, if we want to infer from too small datasets, we should ask ourselves whether this is a good idea at all… it might be better to simply refrain from any data analysis than to come up with conclusions that are likely to be unjustified).

Also, we should not exercise data removal in situations where missingness is conditional (e.g., data only available for infants) or otherwise group-dependent (not-completely at random; e.g., it might result in an imbalanced dataset).

Exercise 17.4 *With the* `nhanes_p_demo_bmx_2020` *dataset:*

1. *remove all columns that are comprised of missing values only,*
2. *remove all columns that are made of more than 20% missing values,*
3. *remove all rows that only consist of missing values,*
4. *remove all rows that feature at least one missing value,*
5. *remove all columns that feature at least one missing value.*

Hint: **`pandas.DataFrame.dropna`** *might be useful in the simplest cases, and* **`numpy.isnan`** *or* **`pandas.DataFrame.isna`** *with* **`loc`**`[...]` *or* **`iloc`**`[...]` *otherwise.*

17.1.5 Mean Imputation

When we cannot afford or it is inappropriate/inconvenient to proceed with the removal of missing observations or columns, we might try applying some missing value *imputation* techniques. Although, let us be clear – this is merely a replacement thereof by some hopefully useful guesstimates.

Important: Whatever we decide to do with the missing values, we need to be explicit about the way we have handled them in all the reports from data analysis, as sometimes they might strongly affect the results.

Let's consider an example vector with missing values, comprised of heights of the adult participants of the NHANES study.

```
x = nhanes.loc[nhanes.loc[:, "RIDAGEYR"] >= 18, "BMXHT"]
```

The simplest approach is to replace each missing value with the corresponding column's mean (for each column separately). This does not change the overall average (but decreases the variance).

```
xi = x.copy()
xi[np.isnan(xi)] = np.nanmean(xi)
```

Similarly, we could have considered replacing missing values with the median, or – in case of categorical data – the mode.

Our height data are definitely not missing completely at random – in particular, we expect heights to differ, on average, between sexes. Therefore, another basic imputation option is to replace the missing values with the corresponding within-group averages:

```
xg = x.copy()
g = nhanes.loc[nhanes.loc[:, "RIDAGEYR"] >= 18, "RIAGENDR"]
xg[np.isnan(xg) & (g == 1)] = np.nanmean(xg[g == 1])  # male
xg[np.isnan(xg) & (g == 2)] = np.nanmean(xg[g == 2])  # female
```

Unfortunately, whichever imputation method we choose, it will artificially distort the data distribution (and hence introduce some kind of bias), see Figure 17.1.

```
plt.subplot(131)
sns.histplot(x, binwidth=1, binrange=[130, 200], kde=True)
```

(continues on next page)

(continued from previous page)

```
plt.ylim(0, 500)
plt.title("Original")
plt.subplot(132)
sns.histplot(xi, binwidth=1, binrange=[130, 200], kde=True)
plt.ylim(0, 500)
plt.title("Replace by mean")
plt.subplot(133)
sns.histplot(xg, binwidth=1, binrange=[130, 200], kde=True)
plt.ylim(0, 500)
plt.title("Replace by group mean")
plt.show()
```

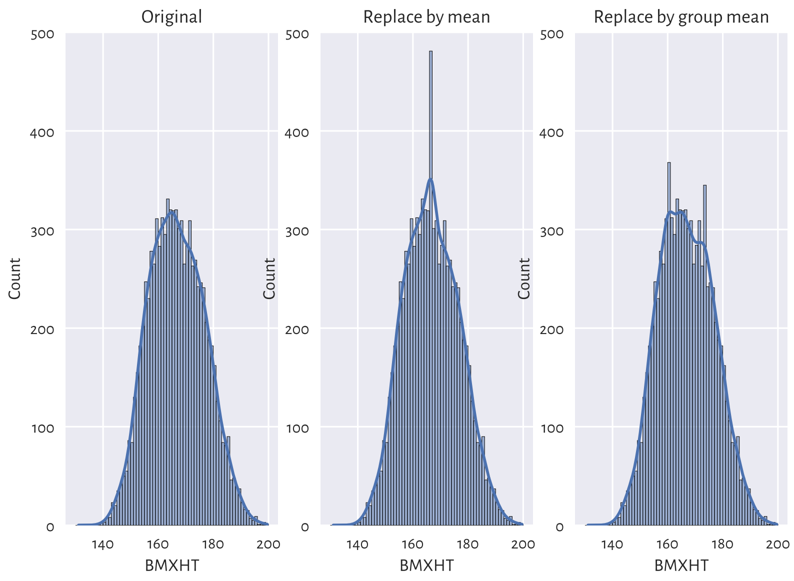


Figure17.1: Mean imputation distorts the data distribution

Note that these effects might not be visible if we increase the bin widths, but they are still there. After all, we have added to the sample many identical values.

Exercise 17.5 *With the* `nhanes_p_demo_bmx_2020` *dataset:*

1. *for each numerical column, replace all missing values with the column averages,*

2. *for each categorical column, replace all missing values with the column modes,*
3. *for each numerical column, replace all missing values with the averages corresponding to a patient's sex (as given by the* `RIAGENDR` *column).*

Note: (*) **`sklearn.impute.KNNImputer`** implements missing value imputation based on averaging data from an observation's non-missing neighbours. This is an extension of the simple idea of finding the most "similar" observation (with respect to chosen criteria) to a given one and "borrowing" missing measurements from it. More generally, different regression or classification models can be built on non-missing data and then the missing observations can be replaced by the values predicted by those models.

Note: (**) Rubin (e.g., [[LR02]]) suggests the use of a procedure called *multiple imputation* (see also [[vB18]]), where copies of the original datasets are created, missing values are imputed by sampling from some estimated distributions, inference is made, and then the results are aggregated. An example algorithm's implementation is available in **`sklearn.impute.IterativeImputer`**.

17.2 Incorrect Data

Missing data might already be present in a given sample but also we might be willing to mark some existing values as missing, e.g., when they are simply incorrect.

For example:

- for text data, misspelled words;
- for spatial data, GPS coordinates of places out of this world, non-existing zip codes, or invalid addresses;
- for date-time data, misformatted date-time strings, incorrect dates such as "29 February 2011", an event's start date being after the end date;
- for physical measurements, observations that do not meet specific constraints, e.g., negative ages, or heights of people over 300 centimetres;
- IDs of entities that simply do not exist (e.g., unregistered or deleted clients' accounts);

and so forth.

In order to be able to identify and handle incorrect data, we need specific knowledge valid for a particular domain. Optimally, basic data validation techniques should already be employed on the data collection stage, for instance when a user submits an online form.

There can be many tools that can assist us with identifying errorneous observations, e.g., spell checkers such as `hunspell`[166].

For smaller datasets, observations can also be manually inspected. However, sometimes we will have to develop our own algorithms for detecting "bugs" in data.

Exercise 17.6 *Given some data frame with numeric columns only, perform what follows.*

1. *Check if all numeric values in each column are between 0 and 1000.*
2. *Check if all values in each column are unique.*
3. *Verify that all the rowwise sums add up to 1.0 (up to a small numeric error).*
4. *Check if the data frame consists of 0s and 1s only. It that is the case, verify that for each row, if there is a 1 in the first column, then all the remaining columns are filled with 1s too.*

Many data validation methods can be reduced to operations on strings. They may be as simple as writing a single regular expression or checking if a label is in a dictionary of possible values and as difficult as writing your own parser for a custom context-sensitive grammar.

17.2.1 Exercises on Validating Data

Once we have imported the data fetched from different sources, it will usually be the case that relevant information will have to be extracted from raw text, e.g., strings like `"1"` should be converted to floating-point numbers. Below we suggest a number of tasks that aid in developing data validation skills involving some operations on textual information.

Exercise 17.7 *Given an example data frame with text columns (manually invented, be creative), perform what follows.*

1. *Remove trailing and leading whitespaces from each string.*
2. *Check if all strings can be interpreted as numbers, e.g., `"23.43"`.*
3. *Verify if a date string in `YYYY-MM-DD` format is correct.*

[166] https://hunspell.github.io/

4. *Determine if a date-time string in `YYYY-MM-DD hh:mm:ss` format is correct.*
5. *Check if all strings are of the form `(+NN) NNN-NNN-NNN` or `(+NN) NNNN-NNN-NNN`, where `N` denotes any digit (valid telephone numbers).*
6. *Inspect whether all strings are valid country names.*
7. *Given a person's date of birth, sex, and their Polish ID number PESEL[167], check if that PESEL is correct.*
8. *Determine if a string represents a correct International Bank Account Number (IBAN[168]) (note that IBANs feature two check digits).*
9. *Transliterate text to ASCII, e.g., `"□ żółty ©"` to `"-> zolty (C)"`.*
10. *Using an external spell checker, determine if every string is a valid English word.*
11. *Using an external spell checker, ascertain that every string is a valid English noun in singular form.*
12. *Resolve all abbreviations by means of a custom dictionary, e.g., `"Kat."` → `"Katherine"`, `"Gr."` → `"Grzegorz"`.*

17.3 Outliers

Another group of inspectionworthy observations consists of *outliers*, which we may define as the samples that reside at areas of substantially lower density than their neighbours.

Outliers might be present due to an error, or their being otherwise anomalous, but they may also simply be interesting, original, or novel. After all, statistics does not give any meaning to data items; humans do.

What we do with outliers is a separate decision. We can get rid of them, correct them, replace them with a missing value (and then possibly impute), etc.

17.3.1 Unidimensional Data

For unidimensional data (or individual columns in matrices and data frames), usually the first few smallest and largest observations should be inspected manually. It might be, for instance, the case that someone accidentally entered a patient's height in metres

[167] https://en.wikipedia.org/wiki/PESEL
[168] https://en.wikipedia.org/wiki/International_Bank_Account_Number

instead of centimetres – such cases are easily detectable. A data scientist is like a detective.

Recall that in the section on box-and-whisker plots (Section 5.4), one heuristic definition of an outlier that is particularly suited for data that are expected/suspected to come from a normal distribution, was to consider everything that does not fall into the interval $[Q_1 - 1.5\mathrm{IQR}, Q_3 + 1.5\mathrm{IQR}]$.

For skewed distributions such as the ones representing incomes, there might be nothing wrong, at least statistically speaking, with very large isolated observations.

For well-separated multimodal distributions on the real line, outliers may sometimes also fall in-between the areas of high density.

Example 17.8 *That neither box plots themselves, nor the* $1.5\mathrm{IQR}$ *rule might not be ideal tools for multimodal data is exemplified in Figure 17.2, where we have a mixture of N(10, 1) and N(25, 1) samples and 4 potential outliers at 0, 15, 45, and 50.*

```
x = np.loadtxt("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/blobs2.txt")
plt.subplot(121)
sns.boxplot(data=x, orient="h")
plt.subplot(122)
sns.histplot(x, binwidth=1)
plt.show()
```

Note: (*) For the normal distribution N(10, 1), the probability of observing a value ≥ 15 is theoretically non-zero, hence whether we consider 15 an outlier is a matter of taste (or, to be precise, the threshold we impose). Nevertheless, this probability is smaller than 0.000029%, and thus it is simply rational to treat this observation as suspicious.

Exercise 17.9 *For each column in* `nhanes_p_demo_bmx_2020`*, inspect a few smallest and largest observations and see if they make sense. Draw a histogram to verify if the data are perhaps multimodal.*

Exercise 17.10 *Perform the above separately for data in each group as defined by the* `RIAGENDR` *column.*

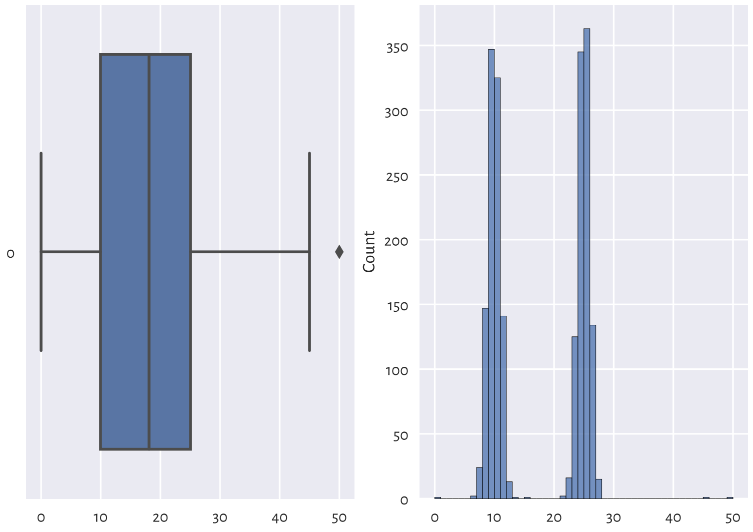


Figure17.2: With boxplots, we may fail to detect some outliers

17.3.2 Robust Aggregates

In Section 5.1.2 we have noted that the arithmetic mean (and hence standard deviation and skewness) are very fragile when exposed to significantly smaller or larger observations than the rest.

For example:

```
x = np.array([1, 2, 3, 4, 5])
np.mean(x)
## 3.0
```

And now:

```
x[-1] = 5000  # oopsie, someone made a typo
np.mean(x)
## 1002.0
```

The sample median is an example of a *robust* aggregate — it ignores almost all but 1-2 middle observations (we'd say it has a high *breakdown point*). Some measures of central tendency that are in-between the mean-median extreme include:

- *trimmed means* – the arithmetic mean of all the observations except a number of, say p, the smallest and the greatest ones,
- *winsorised means* – the arithmetic mean with p smallest and p greatest observations replaced with the (p+1)-smallest the (p+1)-largest one.

As far as spread measures are concerned, the interquartile range (IQR) is a robust statistic. If need be, standard deviation might be replaced with:

- mean absolute deviation from the mean: $\frac{1}{n}\sum_{i=1}^{n} |x_i - \bar{x}|$,
- mean absolute deviation from the median: $\frac{1}{n}\sum_{i=1}^{n} |x_i - m|$,
- median absolute deviation from the median: the median of $(|x_1 - m|, |x_2 - m|, \dots, |x_n - m|)$.

17.3.3 Multidimensional Data (*)

By far we should have got used to the fact that unidimensional data projections might lead to our losing too much information: some values might seem perfectly fine when they are considered in isolation, but already plotting them in 2D reveals the truth.

Consider the following example dataset and the depiction of the distributions of its two natural projections in Figure 17.3.

```
x = np.loadtxt("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/blobs1.txt", delimiter=",")
plt.subplot(221)
sns.boxplot(data=x[:, 0], orient="h")
plt.subplot(222)
sns.histplot(x[:, 0], bins=20)
plt.title("x[:, 0]")
plt.subplot(223)
sns.boxplot(data=x[:, 1], orient="h")
plt.subplot(224)
sns.histplot(x[:, 1], bins=20)
plt.title("x[:, 1]")
plt.show()
```

There is nothing suspicious here. Or is there?

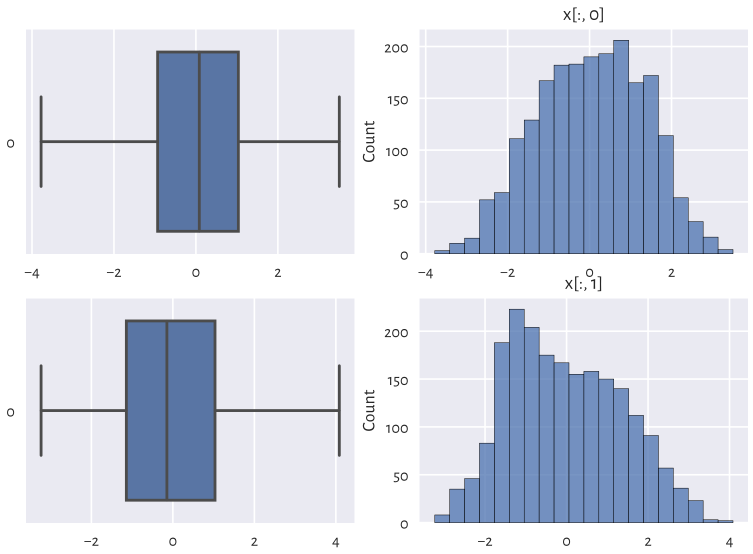


Figure17.3: One-dimensional projections of the `blobs1` dataset

The scatterplot in Figure 17.4 already reveals that the data consist of two quite well-separable blobs:

```
plt.scatter(x[:, 0], x[:, 1])
plt.axis("equal")
plt.show()
```

Also, the are a few observations that we might consider outliers. Yours truly included 8 junk points at the very end of the dataset:

```
x[-8:, :]
## array([[-3. ,  3. ],
##        [ 3. ,  3. ],
##        [ 3. , -3. ],
##        [-3. , -3. ],
##        [-3.5,  3.5],
##        [-2.5,  2.5],
```

(continues on next page)

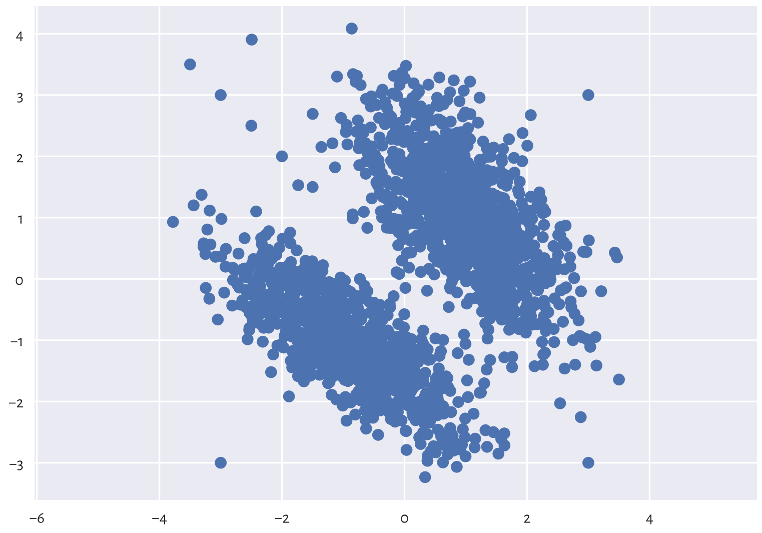


Figure17.4: Scatterplot of the `blobs1` dataset

(continued from previous page)

```
##          [-2.  ,   2. ],
##          [-1.5,   1.5]])
```

Thus, handling multidimensional data requires slightly more sophisticated methods. A quite straightforward approach is to check if there are any points within an observation's radius of some assumed size $\varepsilon > 0$. If that is not the case, we may consider it an outlier.

Example 17.11 *(**) Consider the following code chunk:*

```
import scipy.spatial
t = scipy.spatial.KDTree(x)
n = t.query_ball_tree(t, 0.2)  # epsilon=0.2 (radius)
c = np.array([len(e) for e in n])
c[[0, 1, -2, -1]]  # preview
## array([42, 30,  1,  1])
```

`c[i]` gives the number of points within `x[i, :]`'s ε-radius (with respect to the Euclidean distance),

including the point itself. Therefore, `c[i]==1` *denotes a potential outlier, see Figure 17.5 for an illustration.*

```python
plt.scatter(x[c > 1, 0], x[c > 1, 1], label="normal point")
plt.scatter(x[c == 1, 0], x[c == 1, 1], marker="v", label="outlier")
plt.axis("equal")
plt.legend()
plt.show()
```

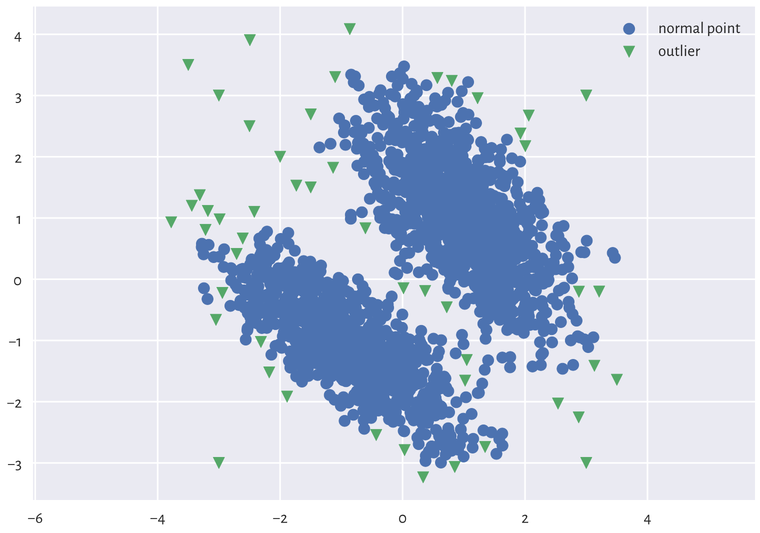


Figure17.5: Outlier detection based on an ε-radius for the `blobs1` dataset

Exercise 17.12 *(*) Use* `sklearn.neighbors.LocalOutlierFactor` *to detect some outliers in the above dataset.*

Exercise 17.13 *(**) Use* `sklearn.svm.OneClassSVM` *for the same purpose.*

Exercise 17.14 *(**) Play with* `sklearn.ensemble.IsolationForest`*.*

17.4 Censored and Interval Data (*)

Censored data frequently appear in the context of reliability, risk analysis, and biostatistics, where the observed objects might "fail" (e.g., break down, die, withdraw), compare, e.g., [[MKK16]]. Our introductory course can't obviously cover everything, but a beginner analyst should at least be aware of such data being a thing, in particular:

- right-censored data: we only know that the actual value is above the recorded one (e.g., we stopped the experiment on the reliability of light bulbs after 1000 hours, so those which still work will not have a time-of-failure precisely known);
- left-censored data: the actual observation is below the recorded one, e.g., we observe a component's failure, but we don't know for how long it has been in operation before the study has started.

Hence, in such cases, the recorded datum of, say, 1000, can actually mean $[1000, \infty)$, $[0, 1000]$, or $(-\infty, 1000]$.

There might also be instances where we know that a value is in some interval $[a, b]$. There are numerical libraries that deal with *interval computations*, and some data analysis methods exist in such a case.

17.5 Exercises

Exercise 17.15 *How can missing values be represented in* **`numpy`** *and* **`pandas`***?*

Exercise 17.16 *Explain some basic strategies for dealing with missing values in numeric vectors.*

Exercise 17.17 *Why we should be very explicit about the way we have handled missing and other suspicious data? Is it a good idea to mark as missing or remove completely the observations that we* don't like *or otherwise deem* inappropriate, controversial, dangerous, incompatible with our political views, *etc.?*

Exercise 17.18 *Is replacing missing values with the sample arithmetic mean for income data (as in, e.g.,* `uk_income_simulated_2020.txt`*) a sensible strategy?*

Exercise 17.19 *What is the differences between data missing completely at random, missing at random, and missing not at random (according to Rubin's [[Rub76]] classification)?*

Exercise 17.20 *List some basic strategies for dealing with data that might contain outliers.*

18

Time Series

So far we have been using **numpy** and **pandas** mostly for storing:

- *independent* measurements, e.g., where each element is a height record of a different subject; we often consider these a sample of a representative subset of one or more populations, each recorded at a particular point in time;
- frequency distributions, i.e., count data and the corresponding categories or labels,
- data to report (as in: tables, figures, heatmaps, etc.).

In this part we'll explore the most basic concepts related to the wrangling of *time series* data – signals indexed by discrete time. Usually a time series is a sequence of measurements sampled at equally spaced moments, e.g., a patient's heart rate probed every second, daily opening stock market prices, or average monthly temperatures recorded in some location.

18.1 Temporal Ordering and Line Charts

Consider the midrange daily temperatures in degrees Celsius at Spokane International Airport (Spokane, WA, US) between 1889-08-01 (first observation) and 2021-12-31 (last observation). Note that midrange, being the mean of the lowest and highest observed temperature on a given day, is not a particularly good estimate of the average daily reading, however, we must work with the data we have.

```
temps = np.loadtxt("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/spokane_temperature.txt")
```

Let us preview the December 2021 data:

```
temps[-31:]  # last 31 days
## array([ 11.9,    5.8,    0.6,    0.8,   -1.9,   -4.4,   -1.9,    1.4,   -1.9,
```

(continues on next page)

(continued from previous page)

```
##         -1.4,   3.9,   1.9,   1.9,  -0.8,  -2.5,  -3.6, -10. ,  -1.1,
##         -1.7,  -5.3,  -5.3,  -0.3,   1.9,  -0.6,  -1.4,  -5. ,  -9.4,
##        -12.8, -12.2, -11.4, -11.4])
```

Here are some data aggregates – the popular quantiles:

```
np.quantile(temps, [0, 0.25, 0.5, 0.75, 1])
## array([-26.9,   2.2,   8.6,  16.4,  33.9])
```

as well as the arithmetic mean and standard deviation:

```
np.mean(temps), np.std(temps)
## (8.990273958441023, 9.16204388619955)
```

and a graphical summary of the data distribution in Figure 18.1:

```
sns.violinplot(data=temps, orient="h")
plt.show()
```

Important: Contrary to the *independent* measurements case, we do not have to treat vectors representing time series simply as mixed bags of unrelated items (note that when computing data aggregates or plotting histograms, the order of elements does not matter).

Instead, in time series, for any given item x_i, its neighbouring elements x_{i-1} and x_{i+1} denote the recordings occurring directly before and after it. We can use this *temporal ordering* to model how consecutive measurements *depend* on each other, describe how they change over time, forecast future values, detect long-time trends, and so forth.

Here are the data for 2021, plotted as a function of time:

```
plt.plot(temps[-365:])
plt.xticks([0, 181, 364], ["2021-01-01", "2021-07-01", "2021-12-31"])
plt.show()
```

What we see in Figure 18.2 is often referred to as a *line chart* – data points are connected by straight line segments. There are some visible seasonal variations, such as that, well, obviously, winter is colder than summer.

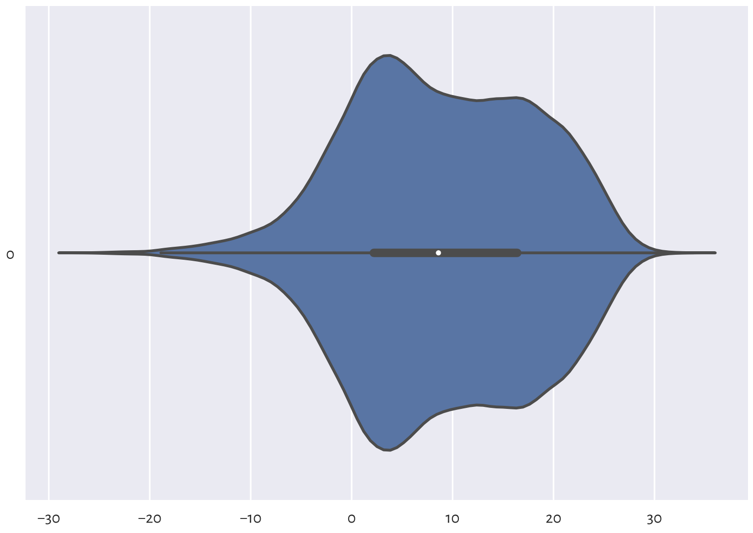


Figure18.1: Distribution of the midrange daily temperatures in Spokane in the period 1889-2021; observations are treated as a bag of unrelated items (temperature on a “randomly chosen day” in a version of planet Earth where there is no climate change)

18.2 Working with Datetimes and Timedeltas

18.2.1 Representation: The UNIX Epoch

`numpy.datetime64`[169] is a type to represent datetimes. Usually, we'll be creating dates from strings, for instance:

```
d = np.array([
    "1889-08-01", "1970-01-01", "2021-12-31", "today"
], dtype="datetime64")
d
```

(continues on next page)

[169] https://numpy.org/doc/stable/reference/arrays.datetime.html

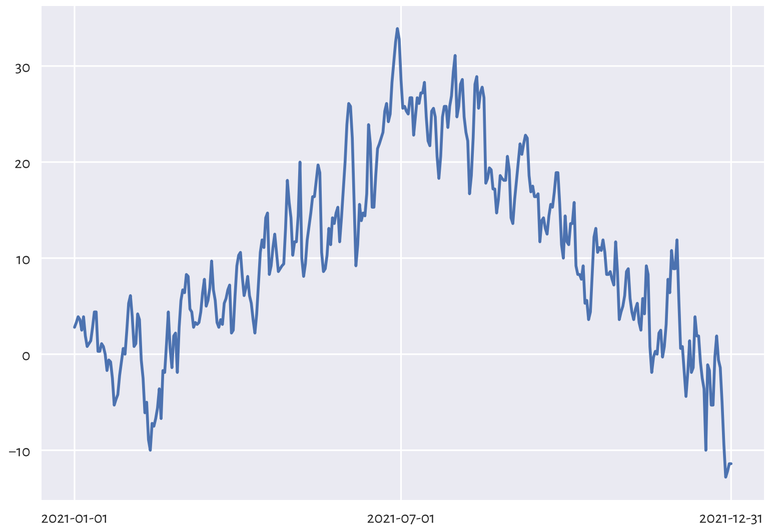


Figure18.2: Line chart of midrange daily temperatures in Spokane for 2021; plotting data as a function of the time variable reveals some seasonal pattern

(continued from previous page)

```
## array(['1889-08-01', '1970-01-01', '2021-12-31', '2022-05-20'],
##       dtype='datetime64[D]')
```

Similarly with datetimes:

```
dt = np.array(["1970-01-01T02:01:05", "now"], dtype="datetime64")
dt
## array(['1970-01-01T02:01:05', '2022-05-20T02:20:41'],
##       dtype='datetime64[s]')
```

Important: Internally, the above are represented as the number of days or seconds since the so-called *Unix Epoch*, 1970-01-01T00:00:00 in the UTC time zone.

Let's verify that:

```
d.astype(float)
## array([-29372.,      0.,  18992.,  19132.])
dt.astype(float)
## array([7.26500000e+03, 1.65301324e+09])
```

Which, when we think about it for a while, is exactly what we expected.

Exercise 18.1 *Write a regular expression that extract all dates in the YYYY-MM-DD format from a (possibly long) string and converts them to datetime64.*

18.2.2 Time Differences

Computing datetime differences is possible thanks to the `numpy.timedelta64` objects:

```
d - np.timedelta64(1, "D")  # minus 1 Day
## array(['1889-07-31', '1969-12-31', '2021-12-30', '2022-05-19'],
##       dtype='datetime64[D]')
dt + np.timedelta64(12, "h")  # plus 12 hours
## array(['1970-01-01T14:01:05', '2022-05-20T14:20:41'],
##       dtype='datetime64[s]')
```

Also, **`numpy.arange`** (and **`pandas.date_range`**) can be used to generate sequences of equidistant datetimes:

```
dates = np.arange("1889-08-01", "2022-01-01", dtype="datetime64[D]")
dates[:3]   # preview
dates[-3:]  # preview
## array(['1889-08-01', '1889-08-02', '1889-08-03'], dtype='datetime64[D]')
## array(['2021-12-29', '2021-12-30', '2021-12-31'], dtype='datetime64[D]')
```

18.2.3 Datetimes in Data Frames

Dates and datetimes can of course be featured in **pandas** data frames:

```
spokane = pd.DataFrame(dict(
    date=np.arange("1889-08-01", "2022-01-01", dtype="datetime64[D]"),
    temp=temps
))
spokane.head()
##          date  temp
## 0 1889-08-01  21.1
```

(continues on next page)

(continued from previous page)

```
## 1 1889-08-02  20.8
## 2 1889-08-03  22.2
## 3 1889-08-04  21.7
## 4 1889-08-05  18.3
```

Interestingly, if we ask the `date` column to become the data frame's `.index` (i.e., row labels), we will be able select date ranges quite easily with **`loc`**`[...]` and string slices (refer to the manual of `pandas.DateTimeIndex` for more details).

```
spokane.set_index("date").loc["2021-12-25":, :].reset_index()
##         date  temp
## 0 2021-12-25  -1.4
## 1 2021-12-26  -5.0
## 2 2021-12-27  -9.4
## 3 2021-12-28 -12.8
## 4 2021-12-29 -12.2
## 5 2021-12-30 -11.4
## 6 2021-12-31 -11.4
```

Example 18.2 *Based on the above, we can plot the data for the last 5 years quite easily, see Figure 18.3.*

```
x = spokane.set_index("date").loc["2017-01-01":, "temp"]
plt.plot(x)
plt.show()
```

Note that now the x-axis labels have been generated automatically.

Note that **`pandas.to_datetime`** can be used to convert strings to datetime objects:

```
dates = ["1991-04-05", "2021-09-12", "2042-12-31"]
dates = pd.Series(pd.to_datetime(dates))
dates
## 0   1991-04-05
## 1   2021-09-12
## 2   2042-12-31
## dtype: datetime64[ns]
```

Exercise 18.3 *From the `birth_dates`[170] dataset, select all people less that 18 years old (as of*

[170] https://github.com/gagolews/teaching_data/blob/master/marek/birth_dates.csv

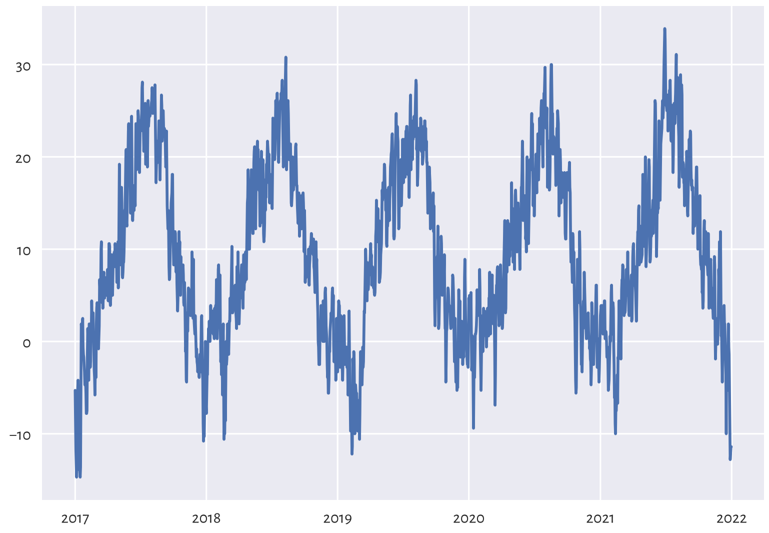


Figure18.3: Line chart of midrange daily temperatures in Spokane for 2017–2021

*the current day). The **pandas.to_datetime** function can also convert arbitrarily formatted date strings, e.g., "MM/DD/YYYY" or DD.MM.YYYY to Series of datetime64s.*

A number of datetime functions and related properties can be referred to via the **pandas.Series.dt** accessor (similarly to **pandas.Series.str** discussed in Chapter 15). In particular, they are are a convenient means for extracting different date or time fields, such as:

```
dates_ymd = pd.DataFrame(dict(
    year  = dates.dt.year,
    month = dates.dt.month,
    day   = dates.dt.day
))
dates_ymd
##     year  month  day
## 0   1991      4    5
## 1   2021      9   12
## 2   2042     12   31
```

Interestingly, **pandas.to_datetime** can also convert data frames with columns named year, month, day, etc. back to datetime objects directly:

```
pd.to_datetime(dates_ymd)
## 0   1991-04-05
## 1   2021-09-12
## 2   2042-12-31
## dtype: datetime64[ns]
```

Example 18.4 *For instance, we can extract the month and year parts of dates to compute the average monthly temperatures it the last 50-ish years:*

```
x = spokane.set_index("date").loc["1970":, ].reset_index()
mean_monthly_temps = x.groupby([
    x.date.dt.year.rename("year"),
    x.date.dt.month.rename("month")
]).temp.mean().unstack()
mean_monthly_temps.head().round(1)
## month    1    2    3    4     5     6     7     8     9   10   11   12
## year
## 1970  -3.4  2.3  2.8  5.3  12.7  19.0  22.5  21.2  12.3  7.2  2.2 -2.4
## 1971  -0.1  0.8  1.7  7.4  13.5  14.6  21.0  23.4  12.9  6.8  1.9 -3.5
## 1972  -5.2 -0.7  5.2  5.6  13.8  16.6  20.0  21.7  13.0  8.4  3.5 -3.7
## 1973  -2.8  1.6  5.0  7.8  13.6  16.7  21.8  20.6  15.4  8.4  0.9  0.7
## 1974  -4.4  1.8  3.6  8.0  10.1  18.9  19.9  20.1  15.8  8.9  2.4 -0.8
```

Figure 18.4 depicts these data on a heatmap:

```
sns.heatmap(mean_monthly_temps)
plt.show()
```

Again we discover the ultimate truth that winters are cold, whereas in the summertime the living is easy, what a wonderful world.

18.2.4 Modelling Event Times with an Exponential Distribution (**)

The exponential distribution family is frequently used for modelling times between different events (i.e., deltas) under the assumption that a system generates on average a constant number of events and that they occur independently of each other.

This may be the case for the times between requests to a cloud service during peak hours, wait times for the next pedestrian to appear at a crossing near the Southern Cross Station in Melbourne, or the amount of time it takes a bank teller to interact with a cus-

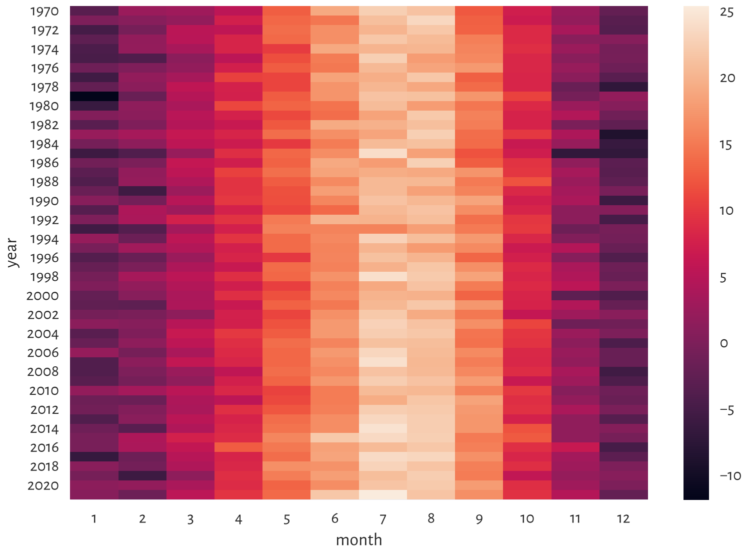


Figure18.4: Average monthly temperatures

tomer (note that there is a whole branch of applied mathematics – or should we refer to it more fashionably: data science – called queuing theory that deals with this type of modelling).

An exponential family is identified by the scale parameter $s > 0$, being at the same time its expected value. The probability density function of Exp(s) is given by

$$f(x) = \frac{1}{s}e^{-x/s}$$

for $x \geq 0$ and $f(x) = 0$ otherwise. We should be careful: some textbooks choose the parametrisation by $\lambda = 1/s$ instead of s; also **scipy** uses this convention.

Here is a pseudorandom sample where there are 5 events per minute on average:

```
np.random.seed(123)
λ = 60/5  # 5 events per 60 seconds on average
d = scipy.stats.expon.rvs(size=1200, scale=λ)
```

(continues on next page)

(continued from previous page)

```
np.round(d[:8], 3)  # preview
## array([14.307,  4.045,  3.087,  9.617, 15.253,  6.601, 47.412, 13.856])
```

This gave us the wait times between the events (deltas), in seconds.

A natural sample estimator of the scale parameter is of course:

```
np.mean(d)
## 11.839894504211724
```

Which is close to what we *expect*, i.e., 12 seconds between the events.

We can convert the above to datetime (starting at a fixed calendar date), e.g., as follows:

```
t0 = np.array("2022-01-01T00:00:00", dtype="datetime64[ms]")
d_ms = np.round(d*1000).astype(int)  # in milliseconds
t = t0 + np.array(np.cumsum(d_ms), dtype="timedelta64[ms]")
t[:8], t[-2:]  # preview
## (array(['2022-01-01T00:00:14.307', '2022-01-01T00:00:18.352',
##         '2022-01-01T00:00:21.439', '2022-01-01T00:00:31.056',
##         '2022-01-01T00:00:46.309', '2022-01-01T00:00:52.910',
##         '2022-01-01T00:01:40.322', '2022-01-01T00:01:54.178'],
##        dtype='datetime64[ms]'), array(['2022-01-01T03:56:45.312', '2022-01-
↪01T03:56:47.890'],
##        dtype='datetime64[ms]'))
```

W converted the deltas to milliseconds so that we didn't lose precision; `datetime64` is based on integers, not floating-point numbers.

As an exercise, let's apply binning and count how many events occur in each hour:

```
b = np.arange(
    "2022-01-01T00:00:00", "2022-01-01T05:00:00",
    1000*60*60,  # number of milliseconds in 1 hour
    dtype="datetime64[ms]"
)
np.histogram(t, bins=b)
## (array([305, 300, 274, 321]), array(['2022-01-01T00:00:00.000', '2022-01-
↪01T01:00:00.000',
##         '2022-01-01T02:00:00.000', '2022-01-01T03:00:00.000',
##         '2022-01-01T04:00:00.000'], dtype='datetime64[ms]'))
```

We expect 5 events per second, hence, 300 of them per hour. On a side note, from a course in statistics we know that for exponential inter-event times, the number of events per unit of time follows a Poisson distribution.

Exercise 18.5 *(**) Consider the wait_times[171] dataset that gives the times between consecutive events, in seconds. Estimate the event rate per hour. Draw a histogram representing the number of events per hour.*

18.3 Basic Operations

18.3.1 Iterative Differences and Cumulative Sums Revisited

Recall the **numpy.diff** function and its almost-inverse, **numpy.cumsum**. The former can turn a time series into a vector of *relative changes* (also called *deltas*, $\Delta_i = x_{i+1} - x_i$).

```
x = temps[-7:]  # last 7 days
x
## array([ -1.4,  -5. ,  -9.4, -12.8, -12.2, -11.4, -11.4])
```

The iterative differences (deltas) are:

```
d = np.diff(x)
d
## array([-3.6, -4.4, -3.4,  0.6,  0.8,  0. ])
```

For instance, between the second and the first day of the last week, the midrange temperature dropped by -3.6°C.

The other way around, here the cumulative sums of the deltas:

```
np.cumsum(d)
## array([ -3.6,  -8. , -11.4, -10.8, -10. , -10. ])
```

This turned deltas back to a shifted version of the original series. However, we will need the first (root) observation therefrom to restore the dataset in full.

```
x[0] + np.append(0, np.cumsum(d))
## array([ -1.4,  -5. ,  -9.4, -12.8, -12.2, -11.4, -11.4])
```

[171] https://raw.githubusercontent.com/gagolews/teaching_data/master/marek/wait_times.txt

Exercise 18.6 *Consider the euraud-20200101-20200630-no-na[172] dataset which lists daily EUR/AUD exchange rates in the first half of 2020 (remember COVID-19?), with missing observations removed. Using* **numpy.diff***, compute the minimal, median, average, and maximal daily price* changes. *Also, draw a box and whisker plot for these deltas.*

Exercise 18.7 *(*) Consider the* btcusd_ohlcv_2021_dates[173] *dataset which gives the daily BTC/USD exchange rates in 2021:*

```
btc = pd.read_csv("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/btcusd_ohlcv_2021_dates.csv",
    comment="#").loc[:, ["Date", "Close"]]
btc["Date"] = btc["Date"].astype("datetime64[D]")
btc.head(12)
##            Date      Close
## 0  2021-01-01  29374.152
## 1  2021-01-02  32127.268
## 2  2021-01-03  32782.023
## 3  2021-01-04  31971.914
## 4  2021-01-05  33992.430
## 5  2021-01-06  36824.363
## 6  2021-01-07  39371.043
## 7  2021-01-08  40797.609
## 8  2021-01-09  40254.547
## 9  2021-01-10  38356.441
## 10 2021-01-11  35566.656
## 11 2021-01-12  33922.961
```

Convert it to a "lagged" representation, being a convenient form for some machine learning algorithms:

1. *Add the* Change *column giving by how much the price changed since the previous day.*
2. *Add the* Dir *column indicating if the change was positive or negative.*
3. *Add the* Lag1*, …,* Lag5 *columns which give the* Changes *in the 5 preceding days.*

The first few rows of the resulting data frame should look like this (assuming we do not want any missing values):

```
##         Date Close    Change Dir      Lag1      Lag2     Lag3     Lag4     Lag5
```

(continues on next page)

[172] https://raw.githubusercontent.com/gagolews/teaching_data/master/marek/euraud-20200101-20200630-no-na.txt

[173] https://github.com/gagolews/teaching_data/blob/master/marek/btcusd_ohlcv_2021_dates.csv

(continued from previous page)

```
## 2021-01-07 39371  2546.68 inc  2831.93  2020.52 -810.11  654.76 2753.12
## 2021-01-08 40798  1426.57 inc  2546.68  2831.93 2020.52 -810.11  654.76
## 2021-01-09 40255  -543.06 dec  1426.57  2546.68 2831.93 2020.52 -810.11
## 2021-01-10 38356 -1898.11 dec  -543.06  1426.57 2546.68 2831.93 2020.52
## 2021-01-11 35567 -2789.78 dec -1898.11  -543.06 1426.57 2546.68 2831.93
## 2021-01-12 33923 -1643.69 dec -2789.78 -1898.11 -543.06 1426.57 2546.68
```

Note that in the 6th row (representing 2021-01-12), `Lag1` corresponds to `Change` on 2021-01-11, `Lag2` gives the `Change` on 2021-01-10, and so forth.

To spice things up, make sure your code can generate any number, say k *of lagged variables.*

18.3.2 Smoothing with Moving Averages

With time series it makes sense to consider batches of consecutive points as there is a time dependence between them. In particular, we can consider computing different aggregates inside *rolling windows* of a particular size.

For example, given sequence $(x_1, x_2, \dots, x_n)$ and some $k \le n$, the k-moving average is a vector $(y_1, y_2, \dots, y_{n-k+1})$ such that

$$y_i = \frac{1}{k} \sum_{j=1}^{k} x_{i+j-1},$$

i.e., the arithmetic mean of k consecutive observations starting at x_i.

For example, here are the temperatures in the last 7 days of December 2011:

```
x = spokane.set_index("date").iloc[-7:, :]
x
##             temp
## date
## 2021-12-25  -1.4
## 2021-12-26  -5.0
## 2021-12-27  -9.4
## 2021-12-28 -12.8
## 2021-12-29 -12.2
## 2021-12-30 -11.4
## 2021-12-31 -11.4
```

The 3-moving average:

```
x.rolling(3, center=True).mean().round(2)
##              temp
## date
## 2021-12-25    NaN
## 2021-12-26  -5.27
## 2021-12-27  -9.07
## 2021-12-28 -11.47
## 2021-12-29 -12.13
## 2021-12-30 -11.67
## 2021-12-31    NaN
```

We get, in this order: the mean of the first 3 observations; the mean of the 2nd, 3rd, and 4th items; then the mean of the 3rd, 4th, and 5th; and so forth. Note that we have centred the observations in such a way that we have the same number of missing values at the start and end of the series. This way, we treat the first 3-day moving average (the average of the temperatures on the first 3 days) as representative for the 2nd day.

And now for something completely different; the 5-moving average:

```
x.rolling(5, center=True).mean().round(2)
##              temp
## date
## 2021-12-25    NaN
## 2021-12-26    NaN
## 2021-12-27  -8.16
## 2021-12-28 -10.16
## 2021-12-29 -11.44
## 2021-12-30    NaN
## 2021-12-31    NaN
```

Applying the moving average has the nice effect of *smoothing* out all kinds of broadly-conceived noise. To illustrate this, compare the temperature data for the last 5 years in Figure 18.3 and in Figure 18.5.

```
x = spokane.set_index("date").loc["2017-01-01":, "temp"]
x30 = x.rolling(30, center=True).mean()
x100 = x.rolling(100, center=True).mean()
plt.plot(x30, label="30-day moving average")
plt.plot(x100, "r--", label="100-day moving average")
plt.legend()
plt.show()
```

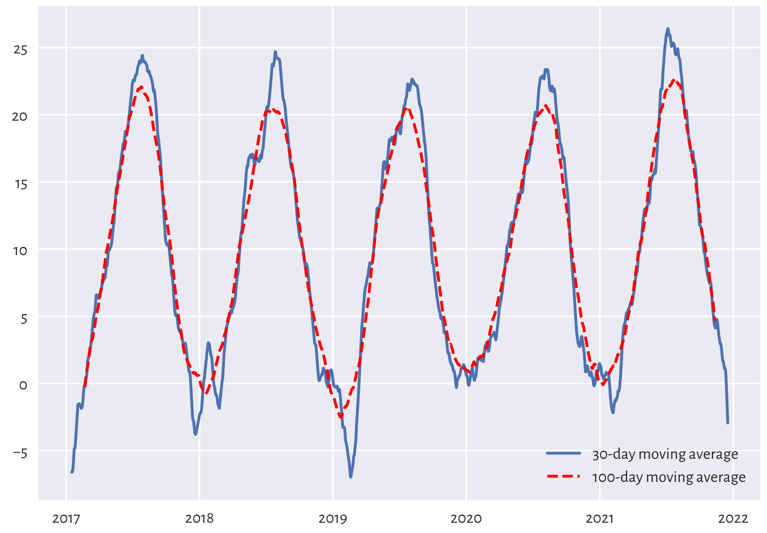


Figure18.5: Line chart of 30- and 100-moving averages of the midrange daily temperatures in Spokane for 2017-2021

Exercise 18.8 *(*) Other aggregation functions can be applied in rolling windows too. Draw, in the same figure, the plots of the 1-year moving minimums, medians, and maximums.*

18.3.3 Detecting Trends and Seasonal Patterns

Thanks to windowed aggregation, we can also detect general trends (when using longish windows). For instance, below we compute (but plot later) 10-year moving averages for the last 50-odd years worth of data:

```
x = spokane.set_index("date").loc["1970-01-01":, "temp"]
x10y = x.rolling(3653, center=True).mean()
```

Based on this, we can compute detrended series:

```
xd = x - x10y
```

Seasonal patterns can be revealed by smoothening out the detrended version of the data, e.g., using a 1-year moving average:

```
xd1y = xd.rolling(365, center=True).mean()
```

See Figure 18.6 for an illustration:

```
plt.plot(x10y, label="trend")
plt.plot(xd1y, "r--", label="seasonal pattern")
plt.legend()
plt.show()
```

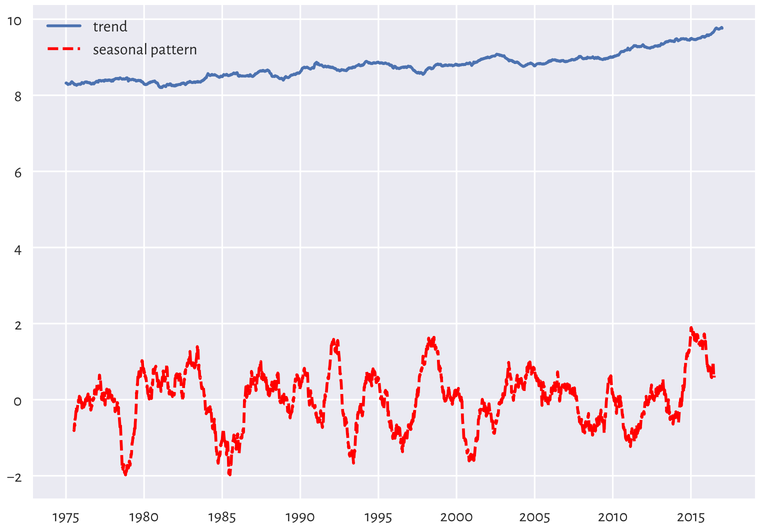


Figure18.6: Trend and seasonal pattern for the Spokane temperatures in recent years

Also, if we know the length of the seasonal pattern (in our case, 365-ish days), we can draw a seasonal plot, where we have a separate curve for each season (here: year) and all series share the same x-axis (here: day of year), see Figure 18.7.

```
from matplotlib import cm
cmap=cm.get_cmap("coolwarm")
x = spokane.set_index("date").loc["1970-01-01":, :].reset_index()
for year in range(1970, 2022, 5):
    y = x.loc[x.date.dt.year == year, :]
    plt.plot(y.date.dt.day_of_year, y.temp,
        c=cmap((year-1970)/(2021-1970)), alpha=0.3,
        label=year if year % 10 == 0 else None)
avex = x.temp.groupby(x.date.dt.day_of_year).mean()
plt.plot(avex.index, avex, "g-", label="Average")
plt.legend()
plt.xlabel("Day of year")
plt.ylabel("Temperature")
plt.show()
```

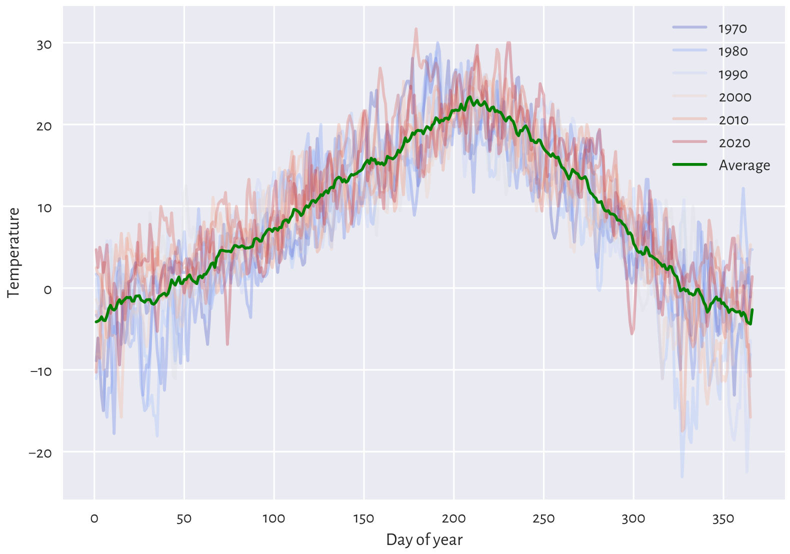


Figure18.7: An example seasonal plot: Spokane temperatures vs day of year between 1970 and 2021

18.3.4 Imputing Missing Values

Missing values in time series can use the information from the neighbouring non-missing observations. After all, it is usually the case that, e.g., today's weather is "similar" to yesterday's and tomorrow's.

The most straightforward ways for dealing with missing values in time series are:

- *forward-fill* – propagate the last non-missing observation,
- *backward-fill* – get the next non-missing value,
- *linearly interpolate between two adjacent non-missing values* – in particular, a single missing value will be replaced by the average of its neighbours.

Example 18.9 *The classic* `air_quality_1973`[174] *dataset gives some daily air quality measurements in New York, between May and September 1973. As an example, let's impute a first few observations in the solar radiation column:*

```
air = pd.read_csv("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/r/air_quality_1973.csv",
    comment="#")
x = air.loc[:, "Solar.R"].iloc[:12]
pd.DataFrame(dict(
    original=x,
    ffilled=x.fillna(method="ffill"),
    bfilled=x.fillna(method="bfill"),
    interpolated=x.interpolate(method="linear")
))
##     original  ffilled  bfilled  interpolated
## 0      190.0    190.0    190.0    190.000000
## 1      118.0    118.0    118.0    118.000000
## 2      149.0    149.0    149.0    149.000000
## 3      313.0    313.0    313.0    313.000000
## 4        NaN    313.0    299.0    308.333333
## 5        NaN    313.0    299.0    303.666667
## 6      299.0    299.0    299.0    299.000000
## 7       99.0     99.0     99.0     99.000000
## 8       19.0     19.0     19.0     19.000000
## 9      194.0    194.0    194.0    194.000000
## 10       NaN    194.0    256.0    225.000000
## 11     256.0    256.0    256.0    256.000000
```

[174] https://github.com/gagolews/teaching_data/blob/master/r/air_quality_1973.csv

Exercise 18.10 *(*) With the `air_quality_2018`[175] dataset:*

1. *Based on the hourly observations, compute the daily mean PM2.5s for Melbourne CBD and Morwell South.*

   For Melbourne CBD, if some hourly measurement is missing, linearly interpolate between the preceding and following non-missing data, e.g., a PM2.5 sequence of `[..., 10, NaN, NaN, 40, ...]` (you need to manually add the `NaN` rows to the dataset) should be transformed to `[..., 10, 20, 30, 40, ...]`.

   For Morwell South, impute the reading as an average of the records in the nearest air quality stations, located in Morwell East, Moe, Churchill, or Traralgon.

2. *Present the daily mean PM2.5 measurements for Melbourne CBD and Morwell South on a single plot. The x-axis labels should be human-readable and intuitive.*

3. *For the Melbourne data, count on how many days the average PM2.5 was greater than in the preceding day.*

4. *Find 5 most air-polluted days for Melbourne.*

18.3.5 Plotting Multidimensional Time Series

Multidimensional time series stored in form of an n-by-m matrix are best viewed as m time series – possibly but not necessarily somehow related to each other – all sampled at the same n points in time (e.g., m different stocks on n consecutive days).

Consider the currency exchange rates for the first half of 2020:

```
eurxxx = np.loadtxt("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/eurxxx-20200101-20200630-no-na.csv",
    delimiter=",")
eurxxx[:6, :]  # preview
## array([[1.6006 , 7.7946 , 0.84828, 4.2544 ],
##        [1.6031 , 7.7712 , 0.85115, 4.2493 ],
##        [1.6119 , 7.8049 , 0.85215, 4.2415 ],
##        [1.6251 , 7.7562 , 0.85183, 4.2457 ],
##        [1.6195 , 7.7184 , 0.84868, 4.2429 ],
##        [1.6193 , 7.7011 , 0.85285, 4.2422 ]])
```

This gives EUR/AUD (how many Australian Dollars we pay for 1 Euro), EUR/CNY (Chinese Yuans), EUR/GBP (British Pounds), and EUR/PLN (Polish Złotys), in this order. Let's draw the four time series, see Figure 18.8.

175 https://github.com/gagolews/teaching_data/blob/master/marek/air_quality_2018.csv.gz

```
dates = np.loadtxt("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/euraud-20200101-20200630-dates.txt",
    dtype="datetime64")
labels = ["AUD", "CNY", "GBP", "PLN"]
styles = ["solid", "dotted", "dashed", "dashdot"]
for i in range(eurxxx.shape[1]):
    plt.plot(dates, eurxxx[:, i], ls=styles[i], label=labels[i])
plt.legend()
plt.show()
```

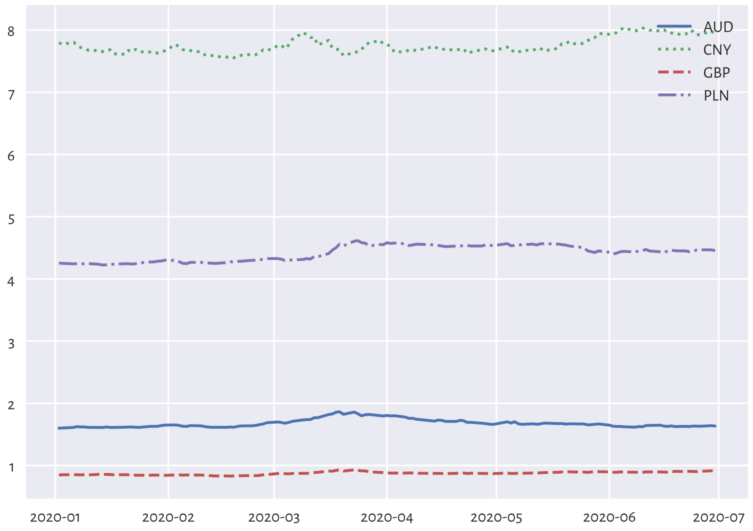


Figure18.8: EUR/AUD, EUR/CNY, EUR/GBP, and EUR/PLN exchange rates in the first half of 2020

Unfortunately, they are all on different scales, hence the plot is not necessarily readable. It would be better to draw these time series on 4 separate plots (compare the trellis plots in Section 13.2.4).

Another idea is to depict the currency exchange rates *relative* to the prices on some day, say, the first one; see Figure 18.9.

```
for i in range(eurxxx.shape[1]):
    plt.plot(dates, eurxxx[:, i]/eurxxx[0, i],
        ls=styles[i], label=labels[i])
plt.legend()
plt.show()
```

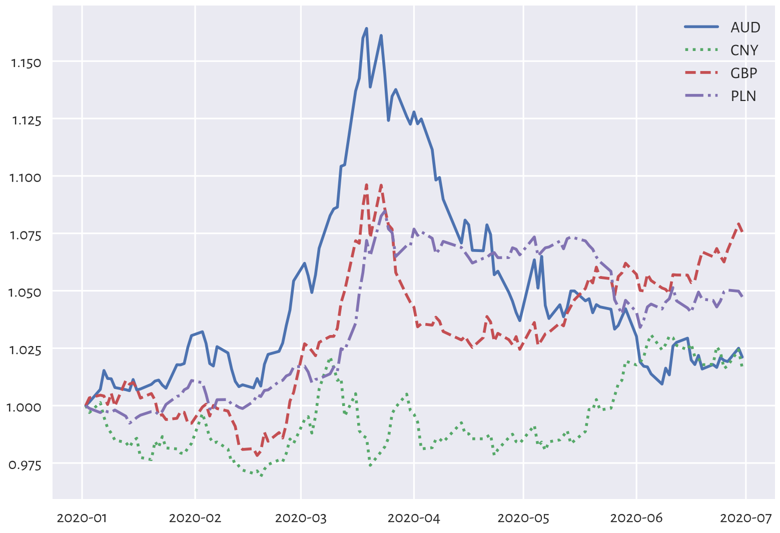


Figure18.9: EUR/AUD, EUR/CNY, EUR/GBP, and EUR/PLN exchange rates relative to the prices on the first day

This way, e.g., relative EUR/AUD rate of ca. 1.15 in mid-March denotes means that if an Australian bought some Euros on the first day and then sold them three-ish months later, they would have 15% more currency (the Euro become 15% stronger relative to AUD).

Exercise 18.11 *Based on EUR/AUD and EUR/PLN rates, compute and plot the AUD/PLN as well as PLN/AUD ones.*

Exercise 18.12 *(*) Draw the EUR/AUD and EUR/GBP rates on a single plot, but with each series having its own*[176] *y axis.*

Exercise 18.13 *(*) Draw the EUR/xxx rates for your favourite currencies over a larger period of time. Use data*[177] *downloaded from the European Central Bank. Add a few moving averages. For each year, identify the lowest and the highest rate.*

18.3.6 Candlestick Plots (*)

Consider the BTC/USD data for 2021:

```
btcusd = np.loadtxt("https://raw.githubusercontent.com/gagolews/" +
    "teaching_data/master/marek/btcusd_ohlcv_2021.csv",
    delimiter=",")
btcusd[:6, :4]  # preview
## array([[28994.01 , 29600.627, 28803.586, 29374.152],
##        [29376.455, 33155.117, 29091.182, 32127.268],
##        [32129.408, 34608.559, 32052.316, 32782.023],
##        [32810.949, 33440.219, 28722.756, 31971.914],
##        [31977.041, 34437.59 , 30221.188, 33992.43 ],
##        [34013.613, 36879.699, 33514.035, 36824.363]])
```

This gives the open, high, low, close prices on the 365 consecutive days (we skipped the marked volume column for readability), which is a common way to summarise daily rates.

The `mplfinance`[178] package (`matplotlib-finance`) features a few functions related to the plotting of financial data. Here, let us briefly mention the candlestick plot.

```
import mplfinance as mpf
dates = np.arange("2021-01-01", "2022-01-01", dtype="datetime64[D]")
mpf.plot(pd.DataFrame(
    btcusd,
    columns=["Open", "High", "Low", "Close", "Volume"]
).set_index(dates).iloc[:31, :], type="candle", style="charles")
# plt.show() # not needed...
```

Figure 18.10 depicts the January 2021 data. Note that this is not a box and whisker plot.

[176] https://matplotlib.org/stable/gallery/subplots_axes_and_figures/secondary_axis.html
[177] https://www.ecb.europa.eu/stats/policy_and_exchange_rates/euro_reference_exchange_rates/html/index.en.html
[178] https://github.com/matplotlib/mplfinance

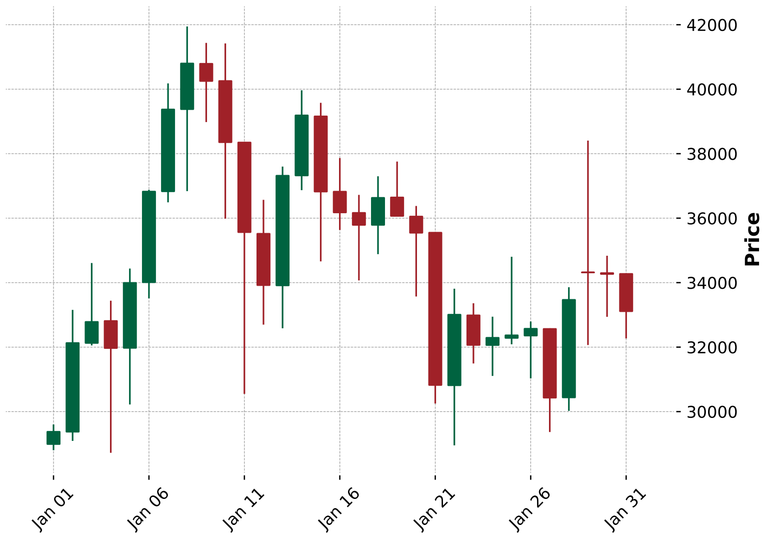


Figure18.10: Candlestick plot for the BTC/USD exchange rates in January 2021

The candlestick body denotes the difference in the market opening and the closing price. The wicks (shadows) give the range (high to low).

Green candlesticks represent bullish days – where the closing rate is greater than the opening one (uptrend). Red candles are bearish (decline)

Exercise 18.14 *Draw the BTC/USD rates for the whole year and add the 10-day moving averages.*

Exercise 18.15 *(*) Draw a candlestick plot manually, without using the* **`mplfinance`** *package.* **`matplotlib.pyplot.fill`** *might be helpful.*

Exercise 18.16 *(*) Using* **`matplotlib.pyplot.fill_between`** *add a semi-transparent polygon that fills the area bounded between the Low and High prices on all the days.*

18.4 Notes on Signal Processing (Audio, Image, Video) (**)

Data science classically deals with information that is or can be represented in tabular form and where particular observations (which can be multidimensional) are usually independent from but still somewhat similar to each other. We often treat them as samples from different larger populations which we would like to describe or compare at some level of generality (think: health data on patients being subject to two treatment plans that we wish to evaluate).

From this perspective, time series which we have briefly touched upon above are already quite distinct, because there is some dependence observed in the time domain: a price of a stock that we observe today is influenced by what was happening yesterday. There might also be some seasonal patterns or trends under the hood. Still, for data of this kind, employing statistical modelling techniques (*stochastic processes*) still can make sense.

Signals such as audio, images, and video, are slightly different, because *structured randomness* does not play dominant role there (unless it's noise that we would like to filter out). Instead, what is happening in the frequency (think: perceiving pitches when listening to music) or spatial (seeing green grass and sky on a photo) domain will play key role there.

Signal processing thus requires a quite distinct set of tools, e.g., Fourier analysis and finite impulse response (discrete convolution) filters. This course obviously cannot be about everything (also because it requires some more advanced calculus skills that we didn't assume the reader to have at this time). Therefore, see, e.g., [[Smi02], [Ste96]].

Nevertheless, we should keep in mind that these are not completely independent domains. For example, we can extract various features of audio signals (e.g., overall loudness, timbre, and danceability of each recording in a large song database) and then treat them as tabular data to be analysed using the techniques described in this course. Moreover, machine learning (e.g., deep convolutional neural networks) algorithms may also be used for tasks such as object detection on images or optical character recognition.

Note: Some Python packages for signal processing worth inspecting include:

- **`scipy.signal`** and **`scipy.ndimage`**,

- **pillow**[179] (**PIL**) ,
- **scikit-image**[180],
- **OpenCV**[181] (a C++ library for image and video processing with Python bindings).

18.4.1 Representing Images

In order to demonstrate that signals are *some kinds of data* after all and that – at some basic level – there is nothing particularly mysterious behind them, let us briefly discuss how images are usually represented.

The **pillow** package provides a number of functions for basic image processing. Let's load the very small example image that we can download from our example data repository[182] (the file needs to be fetched manually).

```
from PIL import Image
filename = "/home/gagolews/Projects/teaching_data/images/grayscale_demo_8x8.
↪png"
im = Image.open(filename)  # download it manually
im.size
## (8, 8)
```

It is a bitmap of height 8 and width 8 pixels.

Converting to a **numpy** matrix:

```
im = np.array(im.getdata()).reshape(im.size)
im
## array([[255, 255, 255,   0,   0, 255, 255, 255],
##        [255, 255, 255, 255, 255, 255, 255, 255],
##        [119, 255, 119, 255, 119, 255, 119, 255],
##        [255, 119, 255, 119, 255, 119, 255, 119],
##        [194, 119, 194, 119, 194, 119, 194, 119],
##        [194, 194, 194, 194, 194, 194, 194, 194],
##        [255, 255, 255, 255, 255, 255, 255, 255],
##        [  0, 255,   0, 255,   0,   0, 255,   0]])
```

[179] https://pillow.readthedocs.io
[180] https://scikit-image.org/
[181] https://www.opencv.org
[182] https://raw.githubusercontent.com/gagolews/teaching_data/master/images/grayscale_demo_8x8.png

The image is in 8-bit greyscale: the values represent 256 shades of grey between 0 (black) and 255 (white), see Figure 18.11.

```
plt.rcParams["axes.grid"] = False
plt.imshow(im, cmap="gray")
plt.rcParams["axes.grid"] = True
plt.show()
```

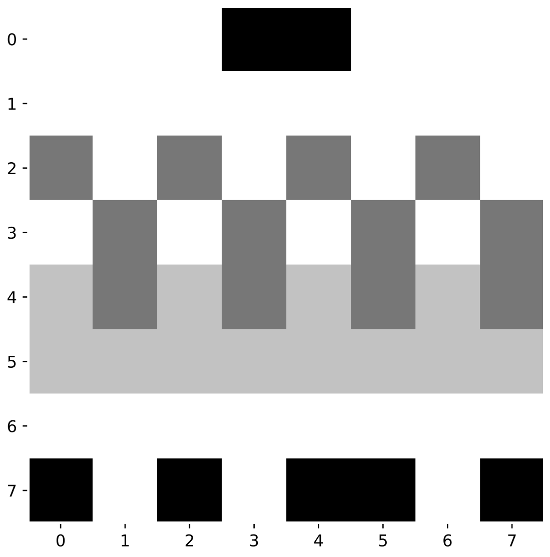


Figure18.11: Example bitmap

Note: Machine learning algorithms will often be expecting flat vectors on inputs – instead of an 8x8 matrix, they had rather be fed with a sequence of 64 elements. We can call `im.flatten()` to get such a form.

Exercise 18.17 *Scale the brightness values from 0-255 to the* [0, 1] *interval, which is another common option (as we can read from the manual of* ***`matplotlib.pyplot.imshow`****).*

Exercise 18.18 *(**) Write a program that imports and visualises chosen example images in the* *famous MNIST[183] and Fashion-MNIST[184] datasets which are frequently used for benchmarking classification algorithms.*

18.4.2 Colour Models

There are quite a few colour models[185] in use, including:

- RBG (red-green-blue),
- RGBA (RGB with the alpha channel governing the pixel transparency/opaqueness),
- HSV (hue-saturation-value),
- CMY (cyan-magenta-yellow) – usually used for printing.

They require each pixel be described by 3 (or 4 if the alpha channel is included) values. Thus, instead of a $h{\times}w$ matrix, we well be dealing with three- dimensional arrays of size $h{\times}w{\times}3$ (or 4).

Exercise 18.19 *Load and display some larger PNG or JPEG file downloaded from the internet.*

18.5 Further Reading

Note that due to limited spacetime, we merely touched upon the most basic methods for dealing with time series. The reader is encouraged to take a look at the broad literature concerning statistical analysis of time series (e.g., issues in forecasting, [[HA21], [OFK17]]) as well as on signal processing (e.g., [[Smi02], [Ste96]]); their toolkits somewhat overlap.

For probabilistic modelling, see textbooks on stochastic processes, e.g., [[Tij03]].

18.6 Exercises

Exercise 18.20 *Let's say we have a time series with* n *observations. What is a* 1*- and an* n*-moving average? Which one is smoother, an* (0.01n)*- or a* (0.1n)*- one?*

183 http://yann.lecun.com/exdb/mnist/
184 https://github.com/zalandoresearch/fashion-mnist
185 https://en.wikipedia.org/wiki/Color_model

Exercise 18.21 *What is the Unix Epoch?*

Exercise 18.22 *How can we recreate the original series when we are given its* **`numpy.diff`***-transformed version?*

Exercise 18.23 *(*) In your own words, describe the key elements of a candlestick plot.*

Exercise 18.24 *(**) What colours do the following RGB hex triples[186] represent?* `#ff0000`*,* `#808000`*,* `#cccccc`*,* `#00aaff`*?*

Exercise 18.25 *(**) What are some other[187] popular colour models?*

[186] https://en.wikipedia.org/wiki/Web_colors
[187] https://en.wikipedia.org/wiki/Color_model

Part VI

Appendix

A

Changelog

Important: This book is still a work in progress. It is already readable, but there will be more, especially related to the processing of multidimensional data. Stay tuned.

Below please find a list of the most noteworthy changes.

- **…**: …ongoing…
  - More lightweight mathematical notation.
  - A way to compute the most correlated pair of variables.
  - A note on modifying elements in an array and adding new rows and columns.
  - An example seasonal plot in the time series chapter.
  - Solutions to the SQL exercises added; using **`pd.testing.assert_frame_equal`** instead of **`DataFrame.equals`** to ignore small round-off errors.
  - More details on file paths.
  - …
- **2022-04-12 (v0.2.1)**:
  - Many chapters merged or relocated.
  - Added captions to all figures.
  - Improved formatting of elements (information boxes such as *note*, *important*, *exercise*, *example*).
- **2022-03-27 (v0.1.1)**:
  - First public release – all chapters drafted.
  - Using Sphinx for building.
- **2022-01-05 (v0.0.0)**:

– Project started.

Bibliography

[Bil95] P. Billingsley. *Probability and Measure*. John Wiley & Sons, 1995.

[CH91] J.M. Chambers and T.J. Hastie. *Statistical Models in S*. Wadsworth & Brooks/Cole, 1991.

[CSN09] A. Clauset, C.R. Shalizi, and M.E.J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009. doi:10.1137/070710111[188].

[DJ03] T. Dasu and T. Johnson. *Exploratory Data Mining and Data Cleaning*. John Wiley & Sons, 2003.

[DKLM05] F.M. Dekking, C. Kraaikamp, H.P. Lopuhaä, and L.E. Meester. *A Modern Introduction to Probability and Statistics: Understanding Why and How*. Springer, 2005.

[FD81] D. Freedman and P. Diaconis. On the histogram as a density estimator: L_2 theory. *Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete*, 57:453–476, 1981.

[Fri06] J.E.F. Friedl. *Mastering Regular Expressions*. O'Reilly, 2006.

[Gag15a] M. Gagolewski. *Data Fusion: Theory, Methods, and Applications*. Institute of Computer Science, Polish Academy of Sciences, 2015. ISBN 978-83-63159-20-7.

[Gag15b] M. Gagolewski. Spread measures and their relation to aggregation functions. *European Journal of Operational Research*, 241(2):469–477, 2015. doi:10.1016/j.ejor.2014.08.034[189].

[Gag22] M. Gagolewski. stringi: Fast and portable character string processing in R. *Journal of Statistical Software*, 2022. in press. URL: https://stringi.gagolewski.com.

[188] https://doi.org/10.1137/070710111
[189] https://doi.org/10.1016/j.ejor.2014.08.034

[GBC16] M. Gagolewski, M. Bartoszuk, and A. Cena. *Przetwarzanie i analiza danych w języku Python (Data Processing and Analysis in Python)*. PWN, 2016. ISBN 978-83-01-18940-2. in Polish.

[Gen03] J.E. Gentle. *Random Number Generation and Monte Carlo Methods*. Springer-Verlag, 2003.

[Gen09] J.E. Gentle. *Computational Statistics*. Springer-Verlag, 2009.

[Gen20] J.E. Gentle. *Theory of Statistics*. book draft, 2020. URL: https://mason.gmu.edu/~jgentle/books/MathStat.pdf.

[GMMP09] M. Grabisch, J.-L. Marichal, R. Mesiar, and E. Pap. *Aggregation Functions*. Cambridge University Press, 2009.

[Gum39] E.J. Gumbel. La probabilité des hypothèses. *Comptes Rendus de l'Académie des Sciences Paris*, 209:645–647, 1939.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[Hun07] J. D. Hunter. Matplotlib: a 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[HA21] R.J. Hyndman and G. Athanasopoulos. *Forecasting: Principles and practice*. OTexts, 2021. URL: https://otexts.com/fpp3/.

[HF96] R.J. Hyndman and Y. Fan. Sample quantiles in statistical packages. *American Statistician*, 50(4):361–365, 1996. doi:10.2307/2684934[190].

[Kle51] S.C. Kleene. Representation of events in nerve nets and finite automata. Technical Report RM-704, The RAND Corporation, Santa Monica, CA, 1951. URL: https://www.rand.org/content/dam/rand/pubs/research_memoranda/2008/RM704.pdf.

[Kuc22] A.M. Kuchling. *Regular Expression HOWTO*. 2022. URL: https://docs.python.org/3/howto/regex.html.

[LR02] R.J.A. Little and D.B. Rubin. *Statistical Analysis with Missing Data*. John Wiley & Sons, 2002.

[McK17] W. McKinney. *Python for Data Analysis*. O'Reilly, 2017.

[MKK16] M. Modarres, M.P. Kaminskiy, and V. Krivtsov. *Reliability Engineering and Risk Analysis: A Practical Guide*. CRC Press, 2016.

[190] https://doi.org/10.2307/2684934

[New05] M.E.J. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, pages 323–351, 2005. doi:10.1080/00107510500052444[191].

[O+21] T. Oetiker and others. *The Not So Short Introduction to LaTeX 2ε*. 2021. URL: https://tobi.oetiker.ch/lshort/lshort.pdf.

[OFK17] J.K. Ord, R. Fildes, and N. Kourentzes. *Principles of Business Forecasting*. Wessex Press, 2017.

[RS59] M. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3:114–125, 1959.

[RT70] D.M. Ritchie and K.L. Thompson. QED text editor. Technical Report 70107-002, Bell Telephone Laboratories, Inc., 1970. URL: https://wayback.archive-it.org/all/20150203071645/http://cm.bell-labs.com/cm/cs/who/dmr/qedman.pdf.

[RC04] C.P. Robert and G. Casella. *Monte Carlo Statistical Methods*. Springer-Verlag, 2004.

[Rub76] D.B. Rubin. Inference and missing data. *Biometrika*, 63(3):581–590, 1976.

[Smi02] S.W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. Newnes, 2002. URL: https://www.dspguide.com/.

[Ste96] K. Steiglitz. *A Digital Signal Processing Primer: With Applications to Digital Audio and Computer Music*. Pearson, 1996.

[Tij03] H.C. Tijms. *A First Course in Stochastic Models*. Wiley, 2003.

[vB18] S. van Buuren. *Flexible Imputation of Missing Data*. CRC Press, 2018. URL: https://stefvanbuuren.name/fimd/.

[vdLdJ18] M. van der Loo and E. de Jonge. *Statistical Data Cleaning with Applications in R*. John Wiley & Sons, 2018.

[Was21] M.L. Waskom. Seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021, 2021. doi:10.21105/joss.03021[192].

[Wic11] H. Wickham. The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, 40(1):1–29, 2011. doi:10.18637/jss.v040.i01[193].

[Wic14] H. Wickham. Tidy data. *Journal of Statistical Software*, 59(10):1–23, 2014. doi:10.18637/jss.v059.i10[194].

[191] https://doi.org/10.1080/00107510500052444
[192] https://doi.org/10.21105/joss.03021
[193] https://doi.org/10.18637/jss.v040.i01
[194] https://doi.org/10.18637/jss.v059.i10