Minimalist Data Wrangling with Python [DRAFTv0.1]

Release [DRAFTv0.1]

Marek Gagolewski

2022-03-27T15:48:34+1100

Contents

Preface **vii**

0.1 The Art of Data Wrangling vii
0.2 Aims and Scope viii
0.3 About the Author x
0.4 Acknowledgements xi

1 Getting Started with *Python* and *JupyterLab* **1**

1.1 Installing Python 1
1.2 What is *JupyterLab*? 2
1.3 Launching *JupyterLab* 3
1.4 First *Jupyter*/*IPython* Notebook 4
1.5 More Cells 5
1.6 Edit vs Command Mode 6
1.7 Markdown Cells 7
1.8 Further Reading 9
1.9 Questions 9

2 Scalar Types in Python **11**

2.1 About Scalar Types 11
2.2 Arithmetic Operators 12
2.3 Named Variables 13
2.4 Character Strings 14
2.5 Calling Built-in Functions 16
  2.5.1 Positional and Keyword Arguments 16
  2.5.2 Modules and Packages 17
2.6 Slots and Methods 17
2.7 Relational and Logical Operators 18
2.8 Controlling Program Flow 20
  2.8.1 The `if` Statement 20
  2.8.2 The `while` Loop 21
2.9 Defining Own Simple Functions 22
2.10 Questions 25

3 Sequential and Other Types in Python **27**
3.1 Sequential Types 27
3.1.1 Lists 27
3.1.2 Tuples 28
3.1.3 Ranges 28
3.1.4 Strings 29
3.2 Working with Sequences 29
3.2.1 Extracting Elements 29
3.2.2 Slicing 30
3.2.3 Modifying Elements 31
3.2.4 Searching Elements 32
3.2.5 Arithmetic Operators 33
3.3 Dictionaries 33
3.4 Iterable Types 34
3.4.1 The `for` loop 35
3.4.2 Tuple Assignment (*) 37
3.5 Object References and Copying (*) 38
3.5.1 Copying References 38
3.5.2 Pass by Assignment 39
3.5.3 Object Copies 39
3.5.4 Modify In-Place or Return a Modified Copy? 40
3.6 Further Reading 41
3.7 Questions 42

Bibliography **43**

Minimalist Data Wrangling with Python is a very-early-and-rough-draft of the forthcoming (ETA 2023) textbook by Marek Gagolewski[1]. It is distributed in the hope that it will be useful. If you detect any bugs or typos, please share them by email[2]. 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.

You can access this book at:

- https://datawranglingpy.gagolewski.com/ (a browser-friendly version)
- https://datawranglingpy.gagolewski.com/datawranglingpy.pdf (PDF)
- https://github.com/gagolews/datawranglingpy (source code)

[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/

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 of how much information you consume and generate when you interact with your 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:

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

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.

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[5]. The 2021 StackOverflow Developer Survey[6] 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

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

data wrangling techniques. This is possible thanks to the famous[7] high quality packages written by the devoted community of open source programmers, including but not limited to *numpy*[8], *scipy*[9], *pandas*[10], *matplotlib*[11], *seaborn*[12], and *sklearn*[13].

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[14] and Julia[15]. There are also some commercial solutions available on the market, but we believe that ultimately all software should be free[16].

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 *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[17]. 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

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

0.3 About the Author

I, Marek Gagolewski[18] (pronounced like Mark Gaggle-Eve-Ski), am 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 Faculty of Mathematics and Information Science, Warsaw University of Technology, Poland.

I'm actively involved in developing *usable* free (libre, independent) and open source software, with particular focus on data science and machine learning. He is the main author and maintainer of stringi[19] – one of the most often downloaded R packages that aims at natural language and string processing as well as the Python and R package genieclust[20] implementing the fast and robust hierarchical clustering algorithm *Genie* with noise point detection.

I'm an author of over 80 publications on machine learning and optimisation algorithms, data aggregation and clustering, statistical modelling, and scientific computing. I taught various courses related to R and Python programming, algorithms, data science, and machine learning in Australia, Poland, and Germany.

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

[18] https://www.gagolewski.com
[19] https://stringi.gagolewski.com
[20] https://genieclust.gagolewski.com

Analysis in Python), [[GBC16]] that I have written with my former (successful) PhD students Maciej Bartoszuk and Anna Cena (in Polish, 2016, published by PWN). The current one is a completely different work, however its predecessor served as a great testbed for many ideas conveyed here. They have also been battle-tested at Warsaw University of Technology, Data Science Retreat (Berlin), and Deakin University (Melbourne).

0.4 Acknowledgements

This book has been prepared with TeX (XeLaTeX) and Sphinx. Python code chunks have been processed with the R package *knitr*. A little help of Makefiles and custom shell scripts dotted the *j*'s and crossed the *f*'s.

1

Getting Started with Python *and* JupyterLab

1.1 Installing Python

We will be using the reference implementation of the Python language (called CPython[21]). Python 3.8 (not: 2.x!) or later will be required.

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*, and *jupyter*, etc.).

Exercise. Download and install the latest version of *Anaconda Individual Edition* from https://www.anaconda.com/products/individual. If you run into any troubles, consult the official product documentation available at https://docs.anaconda.com/anaconda/install/.

Side note. Slightly more advanced users of Unix-like operating systems (GNU/Linux, FreeBSD, etc.), just like yours truly, might prefer downloading Python via their native package manager (e.g., `sudo apt install python3` in Debian and Ubuntu). Then, installing[22] additional Py-

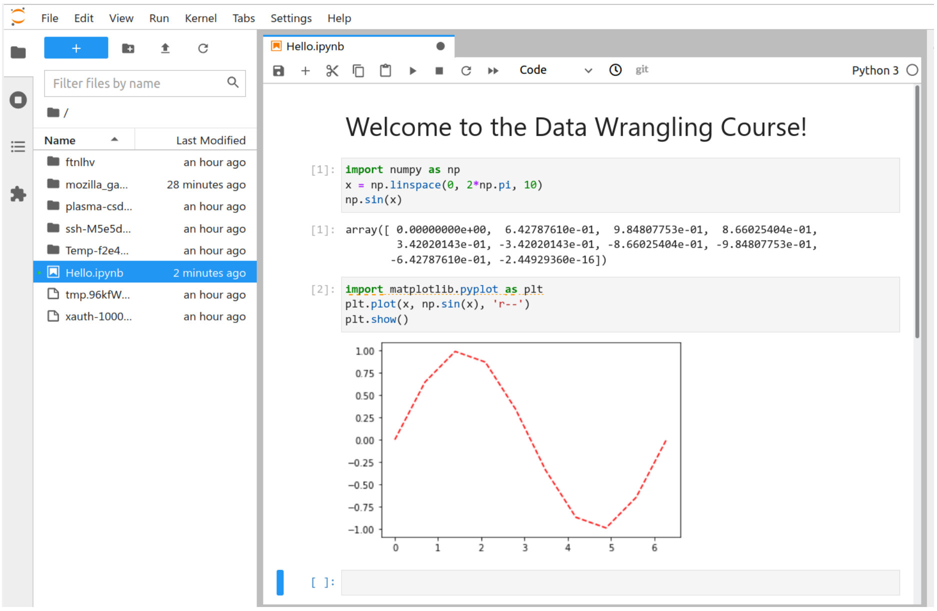
[21] https://www.python.org
[22] https://packaging.python.org/en/latest/tutorials/installing-packages/

thon packages can either be done by the said manager or directly from the Python Package Index (PyPI[23]) via the `pip` tool.

Note that GNU/Linux is the language of choice of machine learning engineers and data scientists both on the desktop and in the cloud. Switching to a free system at some point cannot be recommended highly enough.

1.2 What is *JupyterLab*?

JupyterLab[24] is a web-based development environment supporting numerous[25] programming languages, including of course Python.



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()

[ ]:
```

In *JupyterLab*, we can work with:

[23] https://pypi.org/
[24] https://jupyterlab.readthedocs.io/en/stable/
[25] https://github.com/jupyter/jupyter/wiki/Jupyter-kernels

- *Jupyter* notebooks[26] – `.ipynb` documents combining code, text, plots, and other rich outputs; importantly, code chunks can be created, modified, and run interactively; 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.

It is the definitely not 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.).

Exercise. Head to the official documentation of the *JupyterLab* project located at https://jupyterlab.readthedocs.io/en/stable/index.html and watch the introductory video linked in the *Overview* section.

(**) **Note.** If you are a more advanced student, you can consider jupytext[27] as a means to create `.ipynb` files directly from Markdown files.

1.3 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.

[26] https://jupyterlab.readthedocs.io/en/stable/user/notebook.html
[27] https://jupytext.readthedocs.io/en/latest/

1. Users of the *Anaconda* distribution should be able to start *JupyterLab* via the *Anaconda Navigator*[28], which can be accessed through the Start menu/application launcher. Read more in the official manual[29].

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.

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[30] to be in full control over the software environment.

1.4 First *Jupyter/IPython* 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

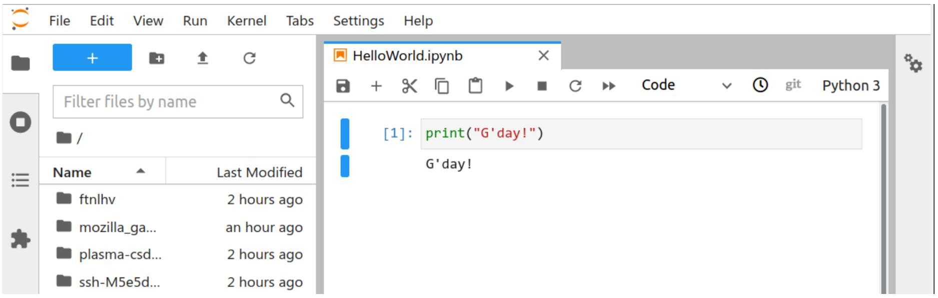
[28] https://docs.anaconda.com/anaconda/navigator/getting-started/#navigator-starting-navigator
[29] https://docs.anaconda.com/anaconda/user-guide/getting-started/#run-python-in-a-jupyter-notebook
[30] https://www.youtube.com/watch?v=Ag1AKIl_2GM

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:

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

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



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

1.5 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:

```python
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` + □/□/□/□ – select text,
- `Ctrl` + `c` – copy,
- `Ctrl` + `x` – cut,
- `Ctrl` + `v` – paste,
- `Ctrl` + `z` – undo,
- `Ctrl` + `]` – indent,
- `Ctrl` + `[` – dedent,
- `Ctrl` + `/` – toggle comment.

1.6 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 □ and □ 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.

1.7 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
    2. bbbb
* three
```

(continues on next page)

(continued from previous page)

~~~
```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* can be modified by entering the *Edit mode* as usual (`Enter` key).

Exercise. Read the official introduction to the Markdown syntax located at https://daringfireball.net/projects/markdown/syntax.

Exercise. Follow the interactive tutorial at https://commonmark.org/help/tutorial/.

Exercise. 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.

1.8 Further Reading

Markdown is one of many markup languages. Other learn-worthy ones include LaTeX (especially for beautiful typesetting of maths, print-ready articles and books, e.g., PDF) and HTML (for the Web).

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

More generally, *Pandoc* 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.

1.9 Questions

1. What is the difference between the Edit and the Command Mode in Jupyter?
2. What is Markdown?
3. How to format a table in Markdown?

2

Scalar Types in Python

Python[31] was designed and implemented by a Dutch programmer Guido van Rossum in the late 1980s. It is a very popular[32] object-oriented programming language that is quite suitable for rapid prototyping. Its name is a tribute to the funniest British comedy troupe[33] ever, therefore we will surely be having a jolly good laugh along our journey.

In this part we quickly introduce the 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 About Scalar Types

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

- `bool` – logical (Boolean) values: `True` and `False`;
- `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 infrequently used in our applications);
- `str` – character strings (which we can also classify as sequential types).

In this part, we'll also cover the `if` statement and ways to define own simple functions.

[31] https://www.python.org/
[32] https://insights.stackoverflow.com/survey/2021#technology-most-popular-technologies
[33] https://en.wikipedia.org/wiki/Monty_Python

Important. Python is case-sensitive. Writing `TRUE` or `true` instead of `True` will result in an error.

Exercise. `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.

2.2 Arithmetic Operators

Here is the list of arithmetic operators available:

```
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[34], e.g., exponentiation has higher priority than multiplication and division which in turn binds more strongly than addition and subtraction. Hence:

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

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

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

2.3 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
```

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

Exercise. Create two named variables `height` and `width`. Based on them, determine your BMI[35].

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

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

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

2.4 Character Strings

Strings, 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, how ya goin\'," he asked.\n"Quite well, thanks," I responded.
 ↪')
## "G'day, how ya goin'," he asked.
## "Quite well, thanks," I responded.
```

Above, "\'" (a way to include an apostrophe in an apostrophe-delimited string) and "\n" (newline character) are examples of *escape sequences*[36].

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'
```

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

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

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

```
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[37] 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.

Important. There are many string operations available – related for example to formatting, pattern searching, extracting chunks; they are very

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

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 a bit later.

2.5 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.

Exercise. Call `help("round")` or `?round` (this is 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.

2.5.1 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
```

(continues on next page)

(continued from previous page)

```
## 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.

2.5.2 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 https://docs.python.org/3/library/math.html for its official documentation and 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.

2.6 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'>
```

Classes define the following kinds of *attributes*:

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

Exercise. 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)*.

2.7 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[38], `not (1 > 2 and 2 < 3)` is true if and only if `1 <= 2 or 2 >= 3` holds.

Exercise. 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)`.

[38] https://en.wikipedia.org/wiki/De_Morgan%27s_laws

2.8 Controlling Program Flow

2.8.1 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!")
## 0.9909723137466452
```

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
## eggs!
```

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
## ham!
## I'd rather eat spam!
## more spam!
```

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. For a given BMI[39] (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 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. (*) 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.

2.8.2 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:
```

(continues on next page)

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

(continued from previous page)

```
    count = count + 1
print(count)
## 120
```

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

2.9 Defining Own Simple 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)
## 0.336
```

Exercise. Write a function named `bmi` which computes and returns BMI[40] given 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:
```

(continues on next page)

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

(continued from previous page)

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

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

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

2.10 Questions

1. What does `import xxxxxx as x` mean?
2. What is the difference between `if` and `while`?
3. Name the scalar types we have introduced here.
4. What is a docstring and how to create and access it?
5. What are keyword arguments?

3

Sequential and Other Types in Python

3.1 Sequential Types

Sequential objects store data items that we can access 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:

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

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.

Lists are *mutable* and hence their state may be changed arbitrarily:

```
x = [
        [1, 2, 3, 4],
        [5, 6, 7, 8]
```

(continues on next page)

(continued from previous page)

```
]  # a list of lists
x.append([9, 10, 11, 12])
print(x)
## [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

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 often we can drop the round brackets and still get a tuple:

```
1, 2, 3  # the same as `(1, 2, 3)`
## (1, 2, 3)
1,  # i.e., `(1,)` - note it's not the same as simply `1` or `(1)`
## (1,)
```

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.).

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

Recall that we have discussed character strings in the previous part.

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

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.

Note. 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
```

(continues on next page)

(continued from previous page)

```
## ['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.

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. 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. 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!'}
```

(*) Here is a more advanced example: in the sequel, we will frequently be reading JSON[41] 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']
```

(continues on next page)

[41] https://en.wikipedia.org/wiki/JSON

(continued from previous page)

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

Exercise. 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. (*) 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
```

(continues on next page)

(continued from previous page)

```
## ['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.
    """
    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. Write your own basic versions (using the `for` loop) of the built-in `max`, `sum`, `any`, and `all` functions.

(*) **Exercise.** 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
```

For example, knowing that the `items` method for a dictionary 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 also 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']
```

(**) 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]
```

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, 4, 3, 2, 5]
```

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])
```

(continues on next page)

(continued from previous page)

```
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

This 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.

The official Python 3 tutorial is available at https://docs.python.org/3/tutorial/index.html. We encourage at some point to skim through at least the following chapters:

- 3. An Informal Introduction to Python[42],
- 4. More Control Flow Tools[43],
- 5. Data Structures[44].

3.7 Questions

1. Name the sequential objects we have introduced.
2. Is every iterable object sequential?
3. Is `dict` an instance of a sequential type?
4. What is the meaning of `+` and `*` operations on strings and lists?
5. 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?
6. (*) How to make an object copy and when we should do so?
7. What is the difference between `x[0]` and `x[:0]`, where `x` is a sequential object?

[42] https://docs.python.org/3/tutorial/introduction.html
[43] https://docs.python.org/3/tutorial/controlflow.html
[44] https://docs.python.org/3/tutorial/datastructures.html

Bibliography

[GBC16] Marek Gagolewski, Maciej Bartoszuk, and Anna Cena. *Przetwarzanie i analiza danych w języku Python (Data Processing and Analysis in Python)*. Wydawnictwo Naukowe PWN, Warsaw, Poland, 2016. ISBN 978-83-01-18940-2. in Polish. URL: https://github.com/gagolews/Analiza_danych_w_jezyku_Python.