Python Simply

Finn Aakre Haugen

12th November 2019

Contents

Preface

This book is written for anyone who wants to learn how to program in Python for calculations – students and academic staff at universities and colleges, students and teachers in schools and professionals in business.

A little about my own background: I am a professor[1] at the University of South-Eastern Norway (USN), campus Porsgrunn. My field of expertise is engineering cybernetics, or automatic control in simpler terms. My education is MSc at the Norwegian Institute of Technology (which is now part of the Norwegian University of Science and Technology) and a PhD at Telemark University College (which is now part of the USN). I have long experience from teaching in universities and in the industry.

Programming is an important part of all the courses I teach. Students use programming in both theoretical and practical assignments. The programming languages that I use in these courses are Python, LabVIEW and MATLAB.

It is great for us, users, that Python is free! So, you can install the Python interpreter – the Python machine – and good programming tools on your own PC - free of charge. (How to get Python is described in Chap. 2.)

Most illustrations in the book are in the form of hand drawings (drawn in Powerpoint on a touch screen PC), not rectilinear drawings. It is a conscious choice. Drawing by hand is more natural and makes it easier to express the meaning of the illustration.

I have chosen to use the same font type and font size for both plain text and Python (Python-specific) names and concepts. It should be clear from the context what is English and what is Python code.

The book contains many numbered examples. Some of the examples *illustrate* a method or approach and follow after the method is explained. Other examples are used to *introduce* a method.

The book contains no exercises, but maybe there will be exercises in a later edition. In any case, a teaching program for learning Python programming must

[1]Norwegian title is “dosent”.

include practice assignments adapted to students or the student's backgrounds and the nature of the subject or course.

I hope the book is easy to understand. If you have comments on the book (negative and positive criticism, or suggestions for changes), please feel free to send them to my email address listed below. The comments will be helpful when revising the book.

Thanks to Marius Lysaker for professional comments, and to Mercedes Noemi Murillo Abril for assistance with translation.

This book is based on Python version 3.7.3 installed on a PC running Windows 10.

The book and program files are freely available on home.usn.no/finnh/books. In case of any revisions, earlier versions of the book and a changelog will be available at the address mentioned above.

This book focuses on Python programming techniques, and not so much on applications. Some good references for applying Python for scientific computing, with applications are (Langtangen 2016) and (Linge & Langtangen 2016). I also mention Haugen (2019) which contains Python examples about simulation, control, optimization, modeling, etc.

Finn Aakre Haugen
Website: home.usn.no/finnh
Email: finn.haugen@usn.no
University of South-Eastern Norway, campus Porsgrunn
August 2019

Chapter 1

Introduction

1.1 About Python

1.1.1 Python - in few words

Python is a programming language developed by Guido van Rossum. Python was launched in 1991 and is constantly evolving. Through this book, you will learn the basics of Python programming and how to apply Python to obtain numerical solutions to mathematical problems. Python can also be used for many applications other than calculations like text processing, file processing, and data communication, but this book is focused on using Python for calculations.

1.1.2 Why choose Python?

There are many alternative programming languages. Why choose Python? Some reasons:

- Python is free.
- Python python can run on different platforms: Windows, Mac and Linux.
- Python python is a powerful language for calculations (on the same level as MATLAB).
- Python makes (really: forces) the programmer to create program code with good visual structure.

- Python is popular and widespread.[1]
- A great amount of material is freely available online.

1.1.3 How popular is Python?

StackOverflow's assessment

StackOverflow[2] runs a popular programming site. If you search for help in programming on Google, StackOverflow is often among the highest on the hit list – *that* can have a variety of causes, of course, you can read:

"June 2017 was the first month that Python was the most visited tag on Stack Overflow within high-income nations. This included being the most visited tag within the US and the UK, and in the top 2 in almost all other high income nations (next to either Java or JavaScript). This is especially impressive because in 2012, it was less visited than any of the other 5 languages, and has grown by 2.5-fold in that time."

Figure 1.1 shows the popularity of well-known programming languages measured in number of searches to StackOverflow during the period 2009-2019. Python is at its peak in 2019. (One might also think that the number of searches is not just due to popularity... :-)

[1] If using a specific programming language is somehow enforced, its deployment may be greater than its popularity, but since Python is a freely available language that has been around for almost 30 years, we may count on popularity and deployment going hand-in-hand.

[2] https://stackoverflow.blog/2017/09/06/incredible-growth-python/

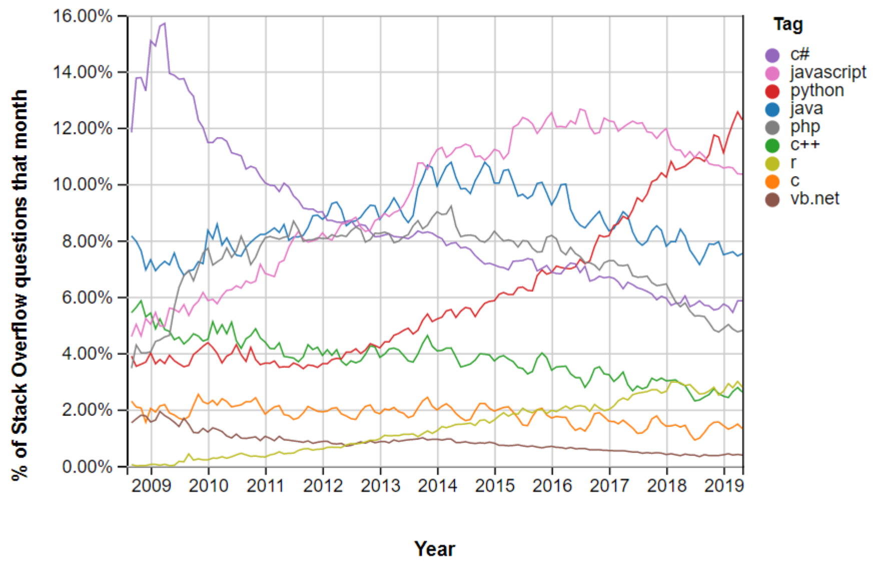
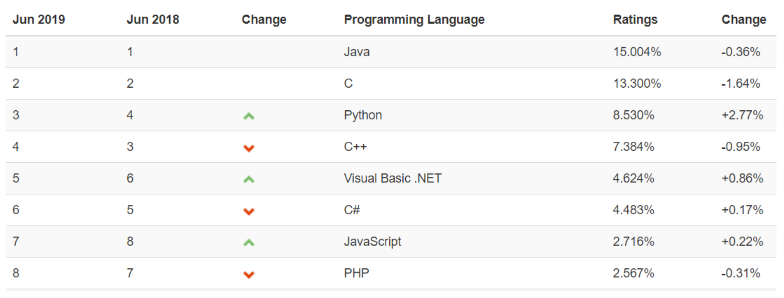


Figure 1.1: The popularity of well-known programming languages measured by the number of searches to StackOverflow during the period 2009-2019

TIOBE's assessment

The company TIOBE ("The Importance Of Being Earnest"), which evaluates and ranks software, has made a statistic that provides a picture of Python's popularity (https://www.tiobe.com/tiobe-index/). The TIOBE index (in percent) is based on the number of professional programmers, courses, and third-party vendors using the various languages. The TIOBE index based on numbers from searches on Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube, and Baidu. Figure 1.2 shows the TIOBE index of well-known programming languages.



Jun 2019	Jun 2018	Change	Programming Language	Ratings	Change
1	1		Java	15.004%	-0.36%
2	2		C	13.300%	-1.64%
3	4	▲	Python	8.530%	+2.77%
4	3	▼	C++	7.384%	-0.95%
5	6	▲	Visual Basic .NET	4.624%	+0.86%
6	5	▼	C#	4.483%	+0.17%
7	8	▲	JavaScript	2.716%	+0.22%
8	7	▼	PHP	2.567%	-0.31%

Figure 1.2: TIOBE scores of well-known programming languages

TIOBE suggests that Python could pass C and Java in popularity in

2022-2023:

“TIOBE Index for June 2019 June Headline: Python continues to soar in the TIOBE index This month Python has reached again an all time high in TIOBE index of 8.5%. If Python can keep this pace, it will probably replace C and Java in 3 to 4 years time, thus becoming the most popular programming language of the world. The main reason for this is that software engineering is booming. It attracts lots of newcomers to the field. Java’s way of programming is too verbose for beginners. In order to fully understand and run a simple program such as "hello world" in Java you need to have knowledge of classes, static methods and packages. In C this is a bit easier, but then you will be hit in the face with explicit memory management. In Python this is just a one-liner. Enough said.”

1.1.4 When is Python *not* useful?

Python is a textuial programming language. Python can not be used for graphical programming. By graphic programming I mean here that function blocks are linked with signal lines in a digital “drawing sheet”, ie the output of one block is connected to input of another block, graphically. (LabVIEW and Simulink are examples of graphical programming tools.)

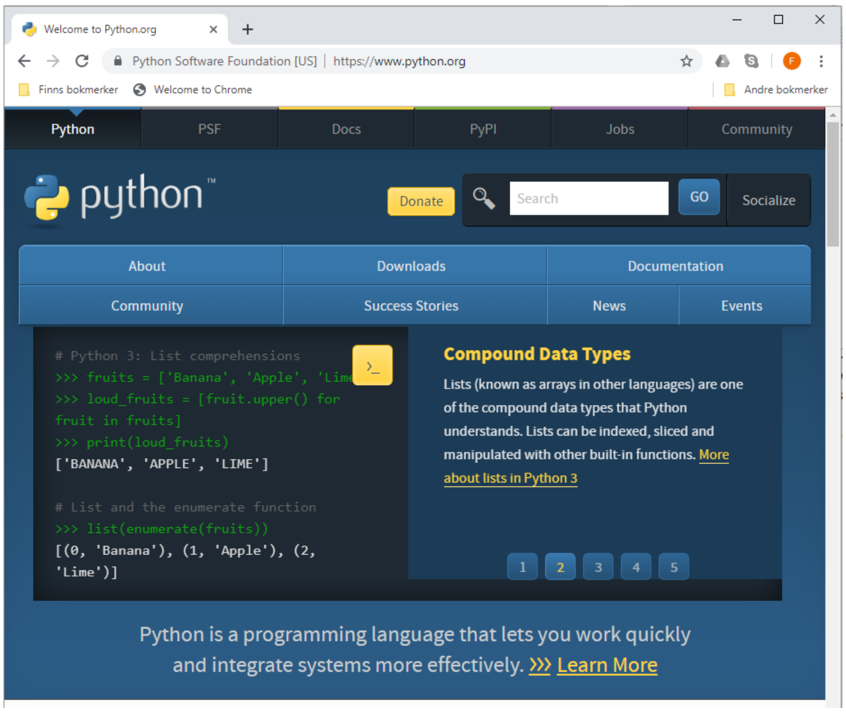
1.1.5 Who holds the threads?

Who makes sure Python is tame and trustworthy? The Python development and publishing organization is The Python Software Foundation (PSF)[3], whose website is

http://python.org

Figure 1.3 shows the PSF website.

[3]From PSFs website: “The Python Software Foundation is an organization devoted to advancing open source technology related to the Python programming language.”



```
# Python 3: List comprehensions
>>> fruits = ['Banana', 'Apple', 'Lime
>>> loud_fruits = [fruit.upper() for
fruit in fruits]
>>> print(loud_fruits)
['BANANA', 'APPLE', 'LIME']

# List and the enumerate function
>>> list(enumerate(fruits))
[(0, 'Banana'), (1, 'Apple'), (2,
'Lime')]
```

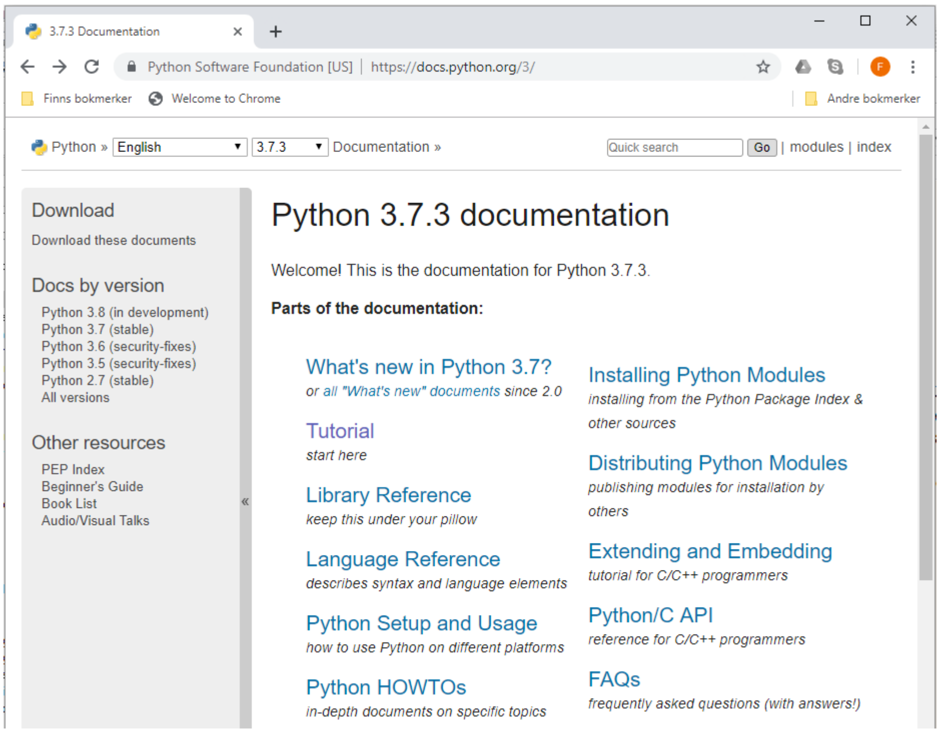
Compound Data Types

Lists (known as arrays in other languages) are one of the compound data types that Python understands. Lists can be indexed, sliced and manipulated with other built-in functions. More about lists in Python 3

Python is a programming language that lets you work quickly and integrate systems more effectively. >>> Learn More

Figure 1.3: Website of Python Software Foundation (PSF) at http://python.org

The PSF website contains Python installation files, documentation, tutorials, overview of Python packages, etc. Figure 1.4 shows the website of the documentation of Python 3.7.3, which per. May 2019 is the latest version of Python.


3.7.3 Documentation
Python Software Foundation [US] | https://docs.python.org/3/
Finns bokmerker
Welcome to Chrome
Andre bokmerker
Python » English 3.7.3 Documentation »
Quick search
Go
modules | index

Python 3.7.3 documentation

Welcome! This is the documentation for Python 3.7.3.

Parts of the documentation:

What's new in Python 3.7?
or all "What's new" documents since 2.0

Tutorial
start here

Library Reference
keep this under your pillow

Language Reference
describes syntax and language elements

Python Setup and Usage
how to use Python on different platforms

Python HOWTOs
in-depth documents on specific topics

Installing Python Modules
installing from the Python Package Index & other sources

Distributing Python Modules
publishing modules for installation by others

Extending and Embedding
tutorial for C/C++ programmers

Python/C API
reference for C/C++ programmers

FAQs
frequently asked questions (with answers!)

Download
Download these documents

Docs by version
Python 3.8 (in development)
Python 3.7 (stable)
Python 3.6 (security-fixes)
Python 3.5 (security-fixes)
Python 2.7 (stable)
All versions

Other resources
PEP Index
Beginner's Guide
Book List
Audio/Visual Talks

Figure 1.4: The website for documentation of Python 3.7.3

Although you can install Python from an installation file on the PSFs website, it is quite common for users of Python to install Python through a so-called Python distribution, which is a collection of Python itself and various additional tools. The most popular distribution is Anaconda, which is described in more detail in Chap.2.1.

1.2 Impatient?

I assume that you are impatient to see what Python programming is all about. We will therefore go through an example of a Python program that contains many of the elements that you can use in your own programs. Although I try to explain the program in some detail, I do not expect you to understand every part of the explanation. The most important thing now is that you get an idea of what Python programming is typically about and get acquainted with some Python concepts and expressions.

Note: I do not expect you to program this example yourself now. The purpose of the example is instead to show a typical program, so that you get a realistic idea of what a typical Python program looks like and how it works.

I will make comments to individual code lines, but let's first take a look at a typical structure of Python programs. Figure 1.5 shows a typical structure of programs which make calculations and plot data. The example we are going to look at has this particular structure, except that the example contains no definitions of functions (defining your own functions is the topic in Chapter 5). Python executes or runs the program code from the top to down. Assuming a structure as shown in Figure 1.5, the Python will run the program code under "Initialization of variables" before the code under "Calculations" are run.

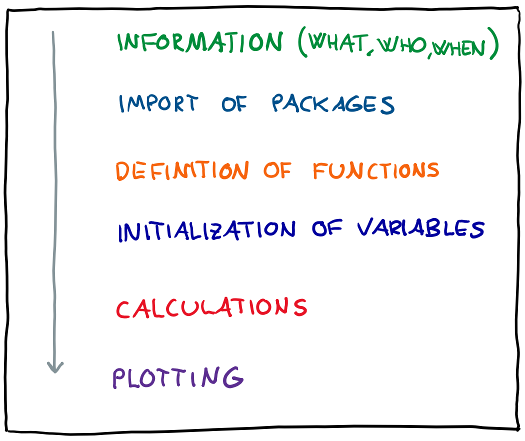


Figure 1.5: A typical structure of a program to perform calculations and plot data

Here is the example.

Example 1.1 *Python program for plotting temperature data*

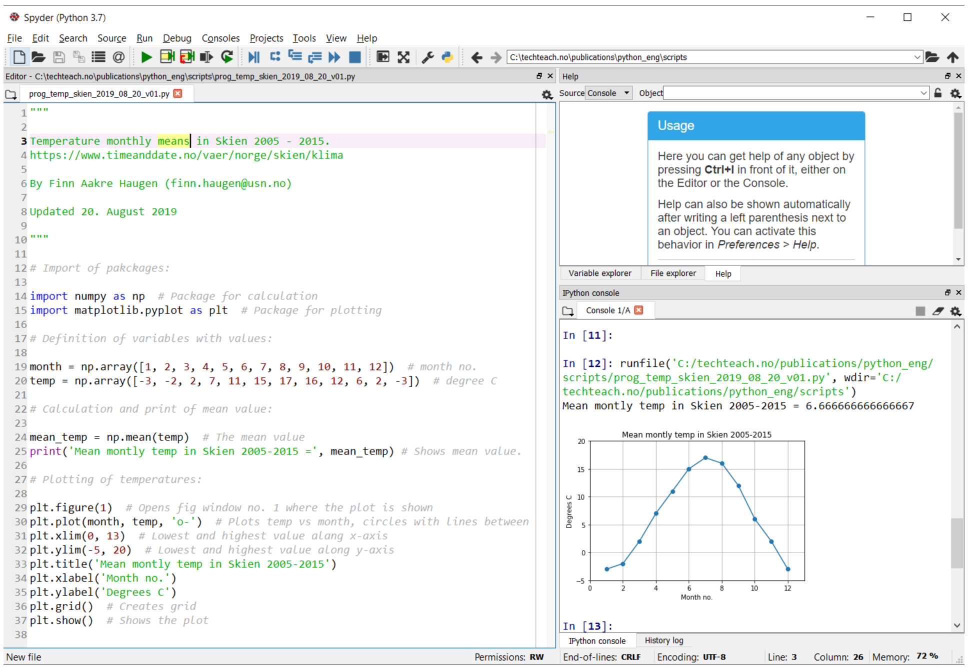
I have created a program that calculates the mean value of the monthly temperatures in Skien over the period 2005 – 2015 and plots these mean temperatures vs. month.[4] The programming environment is Spyder, which is a popular programming environment for Python. You will get to know Spyder in Chap. 2.2.

Figure 1.6 shows the calculated mean and plot generated by the program when it is run in Spyder. The program (or script) is shown in the editor

[4]The data is taken from https://www.timeanddate.no/vaer/norge/skien/klima.

window on the left, while the results of the program run, ie the mean and the plot, are shown in the console in the lower right. (The information in the Help window in the upper left is not interesting here.)

Program name: prog_temp_skien_2019_08_20_v01.py


Spyder (Python 3.7)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\techteach.no\publications\python_eng\scripts
Editor - C:\techteach.no\publications\python_eng\scripts\prog_temp_skien_2019_08_20_v01.py
prog_temp_skien_2019_08_20_v01.py
Help
Source Console Object
Variable explorer
File explorer
Help
IPython console
Console 1/A
IPython console
History log
New file
Permissions: RW
End-of-lines: CRLF
Encoding: UTF-8
Line: 3
Column: 26
Memory: 72 %
Mean montly temp in Skien 2005-2015
Degrees C
Month no.

```
1  """
2
3  Temperature monthly means in Skien 2005 - 2015.
4  https://www.timeanddate.no/vaer/norge/skien/klima
5
6  By Finn Aakre Haugen (finn.haugen@usn.no)
7
8  Updated 20. August 2019
9
10 """
11
12 # Import of pakckages:
13
14 import numpy as np  # Package for calculation
15 import matplotlib.pyplot as plt  # Package for plotting
16
17 # Definition of variables with values:
18
19 month = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])  # month no.
20 temp = np.array([-3, -2, 2, 7, 11, 15, 17, 16, 12, 6, 2, -3])  # degree C
21
22 # Calculation and print of mean value:
23
24 mean_temp = np.mean(temp)  # The mean value
25 print('Mean montly temp in Skien 2005-2015 =', mean_temp) # Shows mean value.
26
27 # Plotting of temperatures:
28
29 plt.figure(1)  # Opens fig window no. 1 where the plot is shown
30 plt.plot(month, temp, 'o-')  # Plots temp vs month, circles with lines between
31 plt.xlim(0, 13)  # Lowest and highest value alang x-axis
32 plt.ylim(-5, 20)  # Lowest and highest value along y-axis
33 plt.title('Mean montly temp in Skien 2005-2015')
34 plt.xlabel('Month no.')
35 plt.ylabel('Degrees C')
36 plt.grid()  # Creates grid
37 plt.show()  # Shows the plot
38
```

Usage

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Help*.

```
In [11]:

In [12]: runfile('C:/techteach.no/publications/python_eng/
scripts/prog_temp_skien_2019_08_20_v01.py', wdir='C:/
techteach.no/publications/python_eng/scripts')
Mean montly temp in Skien 2005-2015 = 6.666666666666667

In [13]:
```

Figure 1.6: The program for calculating the temperature mean and plotting the monthly temperatures in Skien

Below are comments on the program:

- Python runs, or executes, the program code from the top to the bottom.
- The text between lines 1 and 10, ie between the two sequences of three quotation marks, """, are comments – so-called block comments. This text, which we can say constitutes the program head , provides what-who-when information about the program. An informative program header will be very useful, especially a while after the program was created, since details are often forgotten.
- The text after the #-characters in various places in the program are also comments, so-called inline comments. According the so-called

PEP 8 for good code style rules[5], there should be two blank characters in front of #, and one blank after.

- Python neglects both block comments and inline comments when it runs the program. You can take advantage of this when programming: Suppose you have written some program lines that you do *not* want to include during program execution. Instead of removing the program lines completely, you can hide or "comment off" the code lines.[6]
- According to the PEP 8 rules, comments should be written in English, but I believe that "educational" comments, ie explanatory and "over-detailed" comments, may well be written in Norwegian in Norway and Vietnamese in Vietnam, etc.
- In the program, the temperature values are integers. Had the temperatures been decimal, or floating point, numbers, we should have written e.g. 2.1 (period, not comma, is used as decimal separator in Python).
- Line 14: The command imports numpy as np command imports the package named numpy ("numeric python") into Python and makes the package available in our program through the name np. It is common Python tradition to rename numpy to np. The numpy package contains mathematical functions. In our program we use two different functions from the numpy package, as explained below.
- Line 15: The command imports matplotlib.pyplot as plt imports the matplotlib package with its module (a module is a collection of functions) called pyplot and makes the module available in our program through the name plt. Also this renaming is tradition in Python.
- Line 19: We use the numpy function array to define the array named months consisting of numbers representing the month numbers. Note that we put the package name before the function name, ie np.array. We say that month is a variable of the data type array.
- Line 20: We use the numpy function array to define the array called temp consisting of the temperature value for each month. temp is a variable of the data type array.

[5]PEP is short for Python Enhancement Proposal. PEP8: Style Guide for Python Code, see https://www.python.org/dev/peps/pep-0008/.

[6]You can switch between commenting/uncommenting with the keyboard combination Ctrl+1.

- Line 24: We use the numpy function named mean to calculate the mean of the array temp (which was defined in line 20). The variable mean_temp gets (or is assigned) a value equal to this calculated mean.
- Line 25: The command is used to "print" or present in the Spyder console, the text "Mean montly ..." followed by the value of mean_temp. The console is the window to the bottom right of Spyder, see Figure1.6.
- Lines 29-37 open (make) Figure # 1, and plot the temperatures as a function of the month numbers, with filled circles for each point or pair of pairs and with a straight line between the points. The lowest and highest values of the x and y axes are defined (xlim and ylim, respectively). Figure title (title) and axes (xlabel and ylabel) are generated. The plot gets grid. And finally, the figure with the show function appears. All of these different functions that help create the figure, belong to the plt module. Therefore, plt is in front of each function name.
- Line 38: According to the PEP 8 rules mentioned above, there should be a blank line at the end.

[End of Example 1.1]

We will come back to more details about these topics throughout the book.

1.3 Program input, output and workspace

A Python program is an abstract "thing". You cannot physically touch a program, and so it is often called software. When the computer microprocessor executes the instructions expressed in the program code, the program runs. It is when it runs, that it does something, and it can then be considered a "process". This chapter will develop an understanding of the fundamental aspects of this process, ie the running of a program.

A running Python program produces an output from an input. The program has a workspace which is kind of a worktable of the program. Figure 1.7 illustrates the input and output and workspace.

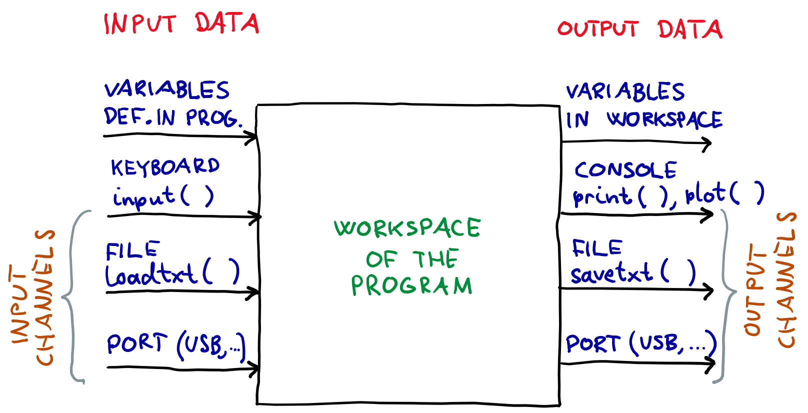


Figure 1.7: A program's inputs and outputs and workspace

Here is a description of Figure 1.7:

- **Workspace** contains all the variables with their values. The content of the workspace is retained after the program is stopped, and still exists when the same or another program is started. The workspace is automatically deleted when you exit the programming tool (for example, Spyder).
- **Input data** is typically numeric data and textual data that the program uses in its data processing. The inputs exist in the work area. The inputs can have several sources:
  - *Variables* with values defined in the program itself, e.g. the array named month in the program shown in Figure 1.6.
  - *The keyboard* where we have written text or numbers. The data can be read into the workspace with Python's input() function, which is described in more detail in Chap. 3.7.
  - *File* with numeric data or textual data. The data can be loaded from the file into the workspace using the general open() function. If the data is numeric data in the form of "text" that we can read with the naked eye, the loadtxt() function in the numpy package is more appropriate than the open() function. The use of loadtxt() is described in Chap. 9.4.
  - *Port*, ie a communication port on the computer. A typical application is continuous reading of measurement data from sensors that generate voltage values which, after being converted to digital values, are read through a USB port.

- **Output data** is typically the numeric or textual data that the program generates when running. The output exists in the workspace. The output may have multiple recipients:
  - *The workspace itself*. Variables that get their values as a result of the program run are available for use in the running program code.
  - *The console* in the current programming tool, e.g. Spyder console shown in the lower right in 1.6. The print() function, which we encounter throughout the book, is used to write numeric data and textual values to the console. The plot() function of the numpy package can be used to plot data in Figures in the console, or alternatively in separate Figures, outside the Spyder window, cf. Section 4).
  - *Fil*e. Data in the workspace can be written to file using. the general open() function. If the data is numeric data in the form of "text", the savetxt() function in the numpy package is more appropriate than the open() function. The use of savetxt() is described in Chap. 9.3.
  - *Port*. A typical application is continuous writing of control signals calculated by our Python program, to so-called actuators such as motors, pumps, valves, heating elements, lamps, switches, etc. via a USB port.

1.4 Why include programming in teaching?

You probably have your own opinion on the extent to which programming is important in teaching STEM courses (science, technology, engineering, mathematics). My opinion is that programming can play an important role in teaching because programming is in line with the core principles of teaching: motivation, concretization, activation, collaboration and individualization:

- ***Motivation***: Programming motivates for theory in mathematics and science because learners see that programming is useful for applying theory. It is also motivating to see that programming is a very important tool for solving practical problems in various disciplines, such as simulation, mathematics, data analysis, statistics, mathematical modeling, monitoring and control of physical systems, etc.

- ***Activation***: Through programming, students will work actively with the theory, and understanding and learning of the theory will be developed. Programming also provides an increased opportunity for experimentation. One can greatly benefit from pre-made programs, but in my experience, it is more instructive to program solutions yourself than to use pre-made programs, as illustrated in Figure 1.8.

- ***Concretization***: Programming is a bridge between theory and applications, see Figure 1.9. With programming, theory can be applied to concrete applications. This develops understanding and learning of the theory, and one can also see to what extent the theory is effective in practice.

- ***Individualization***: Programming provides good opportunities for individual adaptation of the teaching since one can work or try out at one's own pace.

- ***Collaboration***: Programming also provides good opportunities for collaboration by having students or students discuss solutions and/or contribute to common programming tasks.

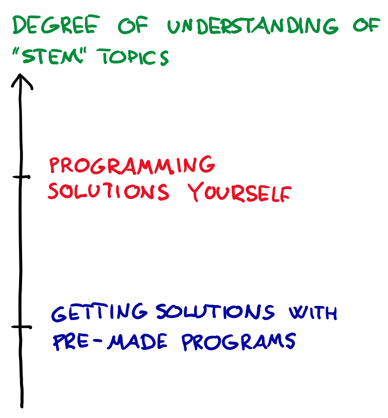

DEGREE OF UNDERSTANDING OF "STEM" TOPICS
PROGRAMMING SOLUTIONS YOURSELF
GETTING SOLUTIONS WITH PRE-MADE PROGRAMS

Figure 1.8: Through programming, students will work actively with theory, and understanding and learning will be developed.

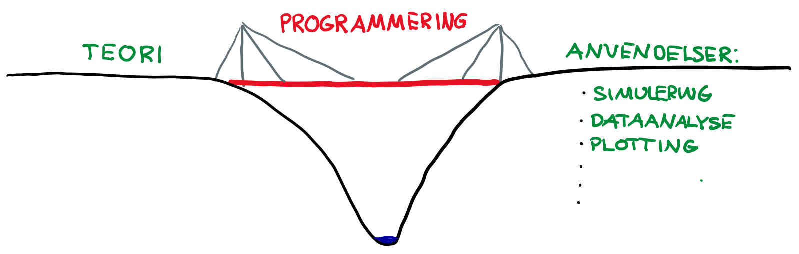


Figure 1.9: Programming is a bridge between theory and applications.

Chapter 2

Programming environments

2.1 Installation of Python

There are several ways to get Python:

- http://python.org, which is the home page of the official Python organization The Python Software Foundation (PSF).
- http://anaconda.com, which is the website of the company Anaconda. From this homepage you can download and install the so-called Anaconda distribution, which is arguably the world's most popular distribution for Python. This distribution contains the "Python machine" itself, which runs the program code, and various useful tools for the programming (writing programs). Lots of useful Python packages are automatically installed with the Anaconda distribution, which may save you some work related to finding and installing packages yourself (but how to do it yourself, is described in Section 2.6.4).

I recommend installing the Anaconda distribution, which is available for Windows, Mac and Linux. Mac and Linux are only available in 64-bit versions. For Windows, you can choose between 32-bit version and 64-bit version. It is common with 64-bit PCs today, so it is natural to choose the 64-bit version.

After installing the Anaconda distribution, Anaconda is on the PC start menu, see Figure 2.1.

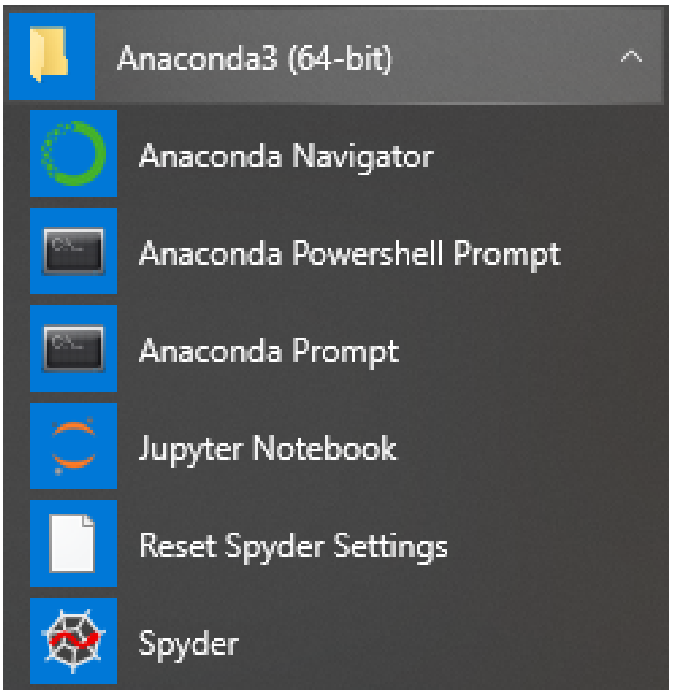
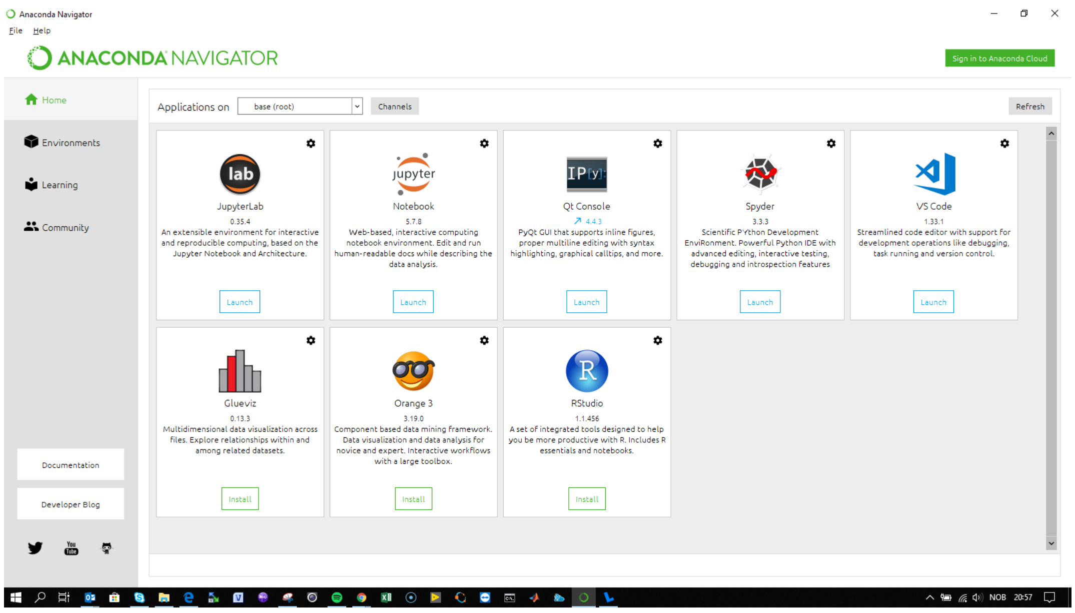


Figure 2.1: Anaconda on the PC start menu

Some information about the various tools available under the Anaconda menu:

- ***Anaconda Navigator***, presenting the various tools included with the Anaconda distribution, see Figure 2.2.
- ***Anaconda Prompt***, which is an Anaconda command window where we can start a (simple) programming environment for Python programming.
- ***Jupyter Notebook***, which is a Python programming environment displayed in a web browser. In Jupyter Notebook we can create so-called Notebook documents, which can contain a mixture of Python program code and information (not program code) in the form of plain text, Latex formatted text (to display mathematical expressions in "book quality"), plots, user interface graphic elements, etc.
- ***Spyder***, which is a thoroughbred programming environment for Python. (I use Spyder in my own programming.)

We will become acquainted with all the three programming environments mentioned above in the subsequent subsections.



JupyterLab
0.35.4
An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.

Notebook
5.7.8
Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

Qt Console
4.4.3
PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.

Spyder
3.3.3
Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features

VS Code
1.33.1
Streamlined code editor with support for development operations like debugging, task running and version control.

Glueviz
0.13.3
Multidimensional data visualization across files. Explore relationships within and among related datasets.

Orange 3
3.19.0
Component based data mining framework. Data visualization and data analysis for novice and expert. Interactive workflows with a large toolbox.

RStudio
1.1.456
A set of integrated tools designed to help you be more productive with R. Includes R essentials and notebooks.

Figure 2.2: Anaconda Navigator

2.2 Spyder

2.2.1 How to open Spyder

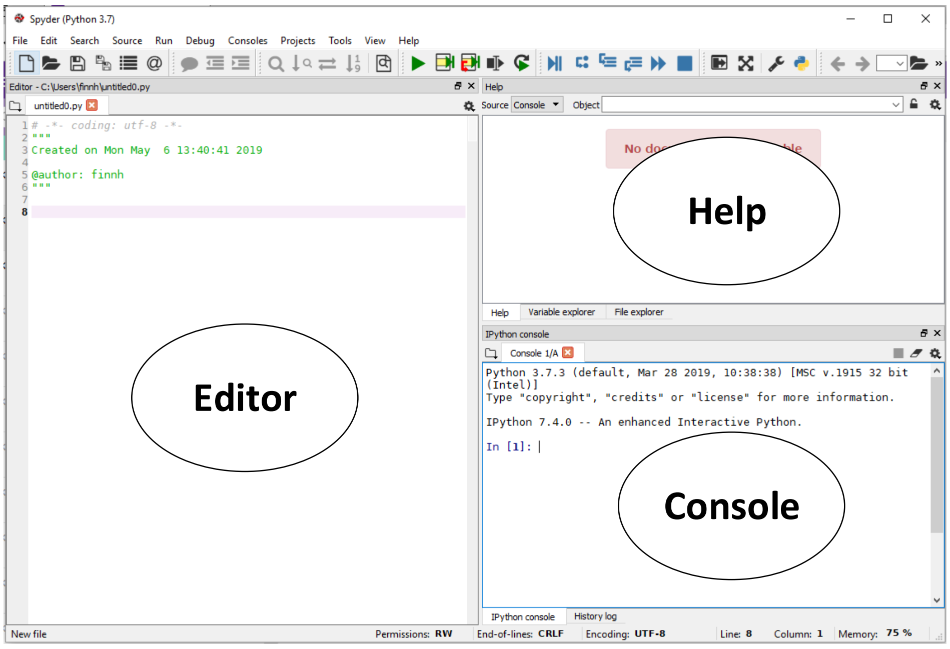
We can open Spyder in two ways:

- Via Spyder button in Anaconda Navigator, see Figure 2.2.
- Via Spyder button under Anaconda on the PC start menu, see Figure 2.1.
  It can also be convenient to make Spyder available also via a button on your PC's taskbar: Right-click the Spyder icon in the Start menu / More / Pin to the taskbar.

Figure 2.3 show Spyder with its three standard windows:

- ***Editor***. The scripts you open will be accessible via their own tab at the top of the editor window.

- ***Console***. The console usually displays the IPython console (interactive Python console). The History log tab shows previous commands.
- ***Help***. The Help window normally displays information about functions. The Variable explorer tab in the Help window displays the variables found in Python's workspace. The File explorer tab shows the files that are in the Pythons workbook, which is the folder where the most recently run program (script) is stored.


Spyder (Python 3.7)
File Edit Search Source Run Debug Consoles Projects Tools View Help
Editor - C:\Users\finnh\untitled0.py
untitled0.py
Help
Source Console Object
Help Variable explorer File explorer
IPython console
Console 1/A
IPython console History log
Editor
Help
Console
New file Permissions: RW End-of-lines: CRLF Encoding: UTF-8 Line: 8 Column: 1 Memory: 75 %

```
# -*- coding: utf-8 -*-
"""
Created on Mon May  6 13:40:41 2019

@author: finnh
"""

```

```
Python 3.7.3 (default, Mar 28 2019, 10:38:38) [MSC v.1915 32 bit
(Intel)]
Type "copyright", "credits" or "license" for more information.

IPython 7.4.0 -- An enhanced Interactive Python.

In [1]:
```

Figure 2.3: Spyder with the three standard windows

2.2.2 How to run Python program code in Spyder

There are two ways to make your PC run Python program code:

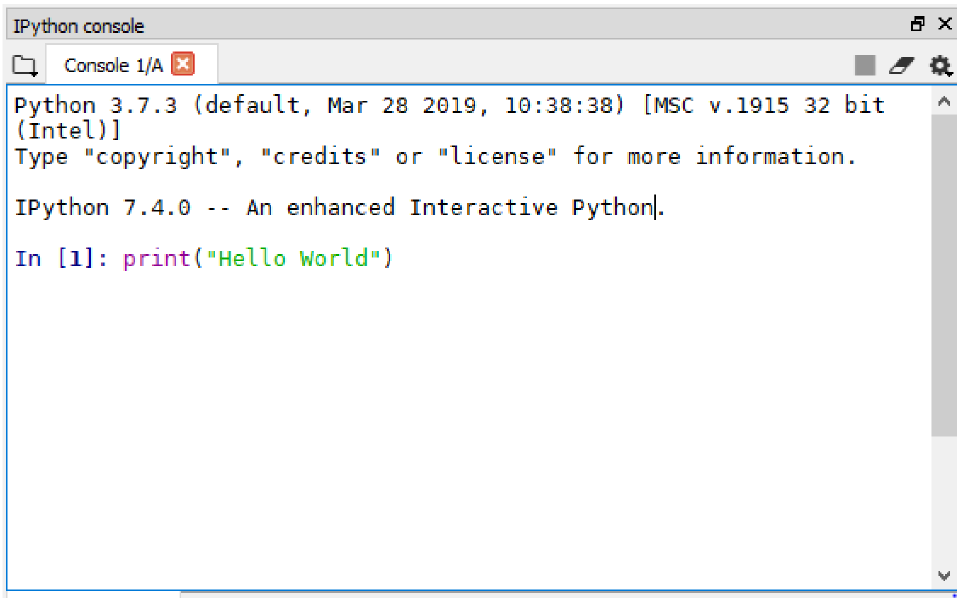
- ***Command line***: You enter the program code on the command line in the console, and execute the code by clicking the Enter key on the keyboard.
- ***Script***: You write the program code in a script in the editor. A script is a text file where you have entered the program code. To run the script, you can either press the F5 key on the keyboard or click

the green Run file button in the toolbar in Spyder. We may say program instead of script, although a program can actually be in the form of code run one by one the command line. (So program is a more general term than script.)

Let's try both ways. It is a general tradition in programming that the Hello World example is the very first programming code example, so let's take that example.

2.2.2.1 Run program code on command line

Type the program code print ('Hello World') on the command line in the console, see Figure 2.4. (Do nothing else for now than write this program code. In other words, do not press any keys other than the appropriate text keys.)



```
Python 3.7.3 (default, Mar 28 2019, 10:38:38) [MSC v.1915 32 bit
(Intel)]
Type "copyright", "credits" or "license" for more information.

IPython 7.4.0 -- An enhanced Interactive Python.

In [1]: print("Hello World")
```

Figure 2.4: The program code print ('Hello World') written on the command line in the Spyder's console

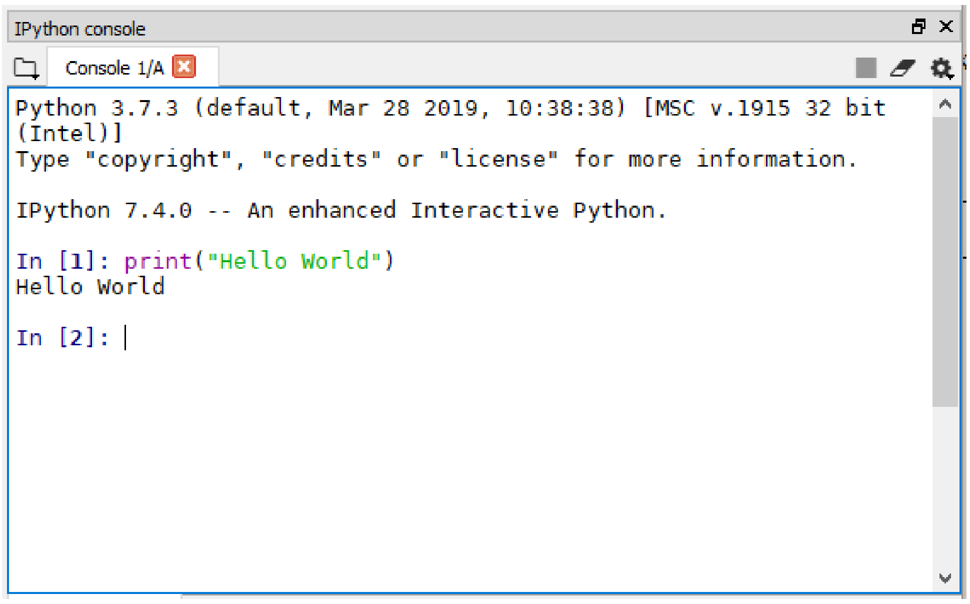
The text In [1]: on the command line indicates that the expression that follows is a so-called command or program code that Python should run or execute. The text In is the abbreviation for input. The number after In, here 1, indicates the command number counted from when this console was opened or started after we opened Spyder.

print(), which is part of this first example of program code, is a function in Python used to display values in the console. As with all other functions, print() needs an input argument which is the “food” or “raw material” that the function is to process. The input argument is specified in the parentheses. In this example, the input argument is the text string ’Hello World’ (remember to include the quotation marks).

So far we have only written the program code

print(’Hello World’)

on the command line, but have not yet run the code. To run the code, we press the Enter key. Python then presents the result in the console, see Figure 2.5.



```
Python 3.7.3 (default, Mar 28 2019, 10:38:38) [MSC v.1915 32 bit
(Intel)]
Type "copyright", "credits" or "license" for more information.

IPython 7.4.0 -- An enhanced Interactive Python.

In [1]: print("Hello World")
Hello World

In [2]: 
```

Figure 2.5: The result is that the program code print(’Hello World’) is run on the command line in the Spyder console.

Figure 2.4 shows how we enter the program code print (’Hello World’) in the console in Spyder and Figure2.5 shows how the result of the program run is presented in the console. If I were to show pictures of the console for all the examples in the book, the book would be unnecessarily large. Instead, the program code that we will write in the console or in a script will be displayed in a box, as shown below.

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

Where applicable, I show the result of the program run at the end of the same text box (possibly in a separate text box), as here:

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

Some useful techniques when using the console command line

- You can enter multiple program expressions one after the other on the command line. They are separated by a semicolon (;) Example: print ('Hello World'); print ('Hello Moon').
- Long program expressions can be broken and placed on new lines with backslash (\\).
  Example:
  1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 \\
  + 11 + 12 + 13 + 14 + 15
- You can delete anything on the command line so that it is blank, with the Esc button on the keyboard.
- You can find old lines that have text beginning with a given character sequence by typing the character sequence and then pressing the up arrow key on the keyboard, such as param↑, as when we show lines that contain e.g. parameter1 and parameter2 and the like. as the first character sequence.
- You can cancel running program code or a command with the Stop the current command button in the upper right corner of the console.

2.2.2.2 Run program code via script

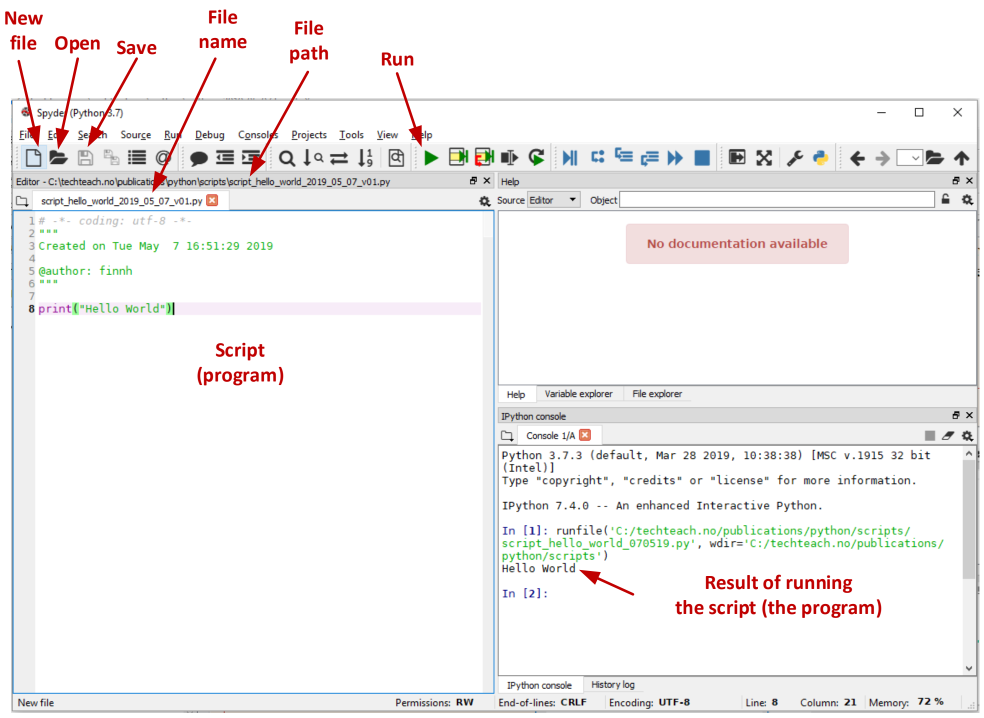
The script is a text file containing program code. (As I already pointed out, we often just say program instead of script.) There are good reasons to create a script and run program code via a script instead of running the individual program expressions from the command line in the console:

- ***Automation***: When the program code is comprehensive and consists of many program lines. When running the script, all program code is executed automatically.[1]

[1]While it is possible to write many and long program lines in the console, it can be impractical to run programs this way.

- ***Storing (filing)***: When you want to store the program in a file, e.g. for later use or for distribution.

We will now create and run a script that will contain the program code print ('Hello World'). Figure 2.6 shows Spyder with the script in the editor. (The Figure is a little ahead of the explanation and also shows the result in the console after the script is run.) To accomplish this, we do the following:



```
# -*- coding: utf-8 -*-
"""
Created on Tue May  7 16:51:29 2019

@author: finnh
"""

print("Hello World")
```

```
Python 3.7.3 (default, Mar 28 2019, 10:38:38) [MSC v.1915 32 bit
(Intel)]
Type "copyright", "credits" or "license" for more information.

IPython 7.4.0 -- An enhanced Interactive Python.

In [1]: runfile('C:/techteach.no/publications/python/scripts/
script_hello_world_070519.py', wdir='C:/techteach.no/publications/
python/scripts')
Hello World

In [2]:
```

Figure 2.6: Spyder with the script (in the editor) and the result of running the script (in the console)

1. ***Open a new script***: In the Spyder menu bar, click the New File button. (Or make the File / New file menu option.) This opens a new script.
   As you can see, there is already some text in the newly opened script. This is a so-called block commentary with informative text about the script (program). You can freely edit this text. There should be information about what the program does, who is the author, contact information in the form of eg. email address, and date of last update. Acc. PEP 8 good code style guidelines, there should be two blank lines after this block comment.

2. ***Write the program code in the script***: You can print ('Hello World') as shown in Figure 2.6. Acc. PEP 8 guidelines, scripts should have a blank line at the end. When you have finished writing, you can save the script (again). You can edit the text in Spyder as in other editors (Word, Notepad, etc.):

   - Copy-and-paste (Ctrl-c followed by Ctrl-v)
   - Cut-and-paste (Ctrl-x followed by Ctrl-v)
   - Delete (Ctrl-x)
   - Undo (Ctrl-z)
   - Redo (Ctrl-Shift-z)

3. ***Save the script***: Although we have not yet written any program code in the script, we can save the script using the Save button or the File / Save menu option or the keyboard combination Ctrl-s. Choose your own folder and file name. The file name will have so-called file extension (extension) py, e.g. minfil.py. As you can see, I have chosen the file name script_hello_world_2019_05_08_v01.py (where v stands for version). In general, it is a good idea to save the script frequently, often with new filenames if there are significant changes between each time you save. One possible strategy for giving file names is to include a brief description and time (date) and the letter v followed by a number (1, 2, 3, ...) that tells which version of the file I am now editing. Example: prog_hello_world_2019_05_08_v01.py.

4. ***Run the script***: There are three alternative ways to run the script, as listed below. The result of the run is shown in Figure 2.6.

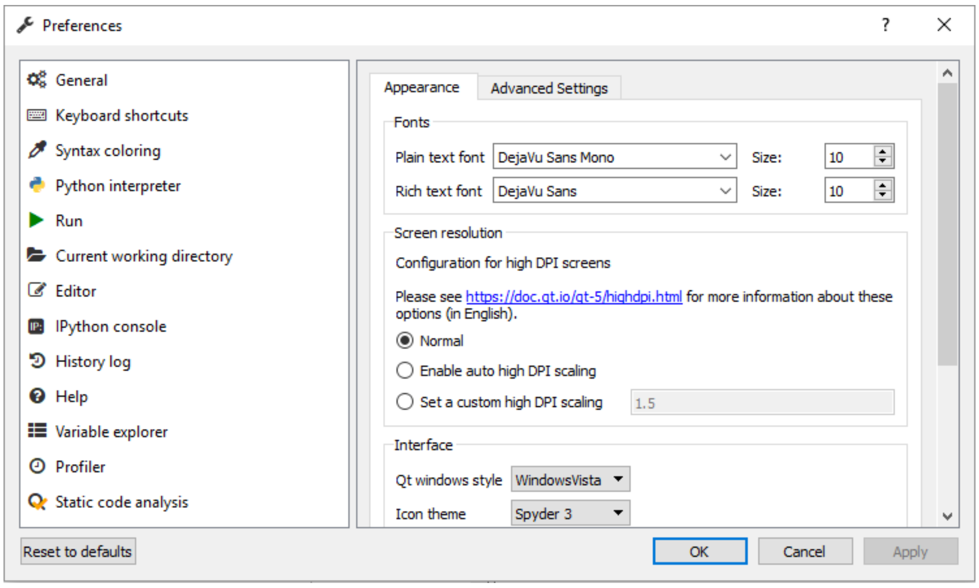
   (a) Run the file button in Spyder's menu bar

   (b) Function key F5 on the keyboard

   (c) With the% run filename.py command, e.g. prog_hello_world_2019_05_08_v01.py, on the console command line. % run is a so-called magic command, which belongs to IPython (abbreviation for Interactive Python), which is the user interface to Python implemented in i.a. Spyder. There are various magic commands. (In terms of plotting, we will meet the magic command% matplotlib that we can use to decide whether a plot should appear inside the Spyder console or in a separate window, outside of Spyder.)

2.2.3 Setting the preferences of Spyder

The Tools / Preferences menu item shows different settings of Spyder, see Figure 2.7.



Fonts

Plain text font: DejaVu Sans Mono Size: 10

Rich text font: DejaVu Sans Size: 10

Screen resolution

Configuration for high DPI screens

Please see https://doc.qt.io/qt-5/highdpi.html for more information about these options (in English).

[✓ Normal] Enable auto high DPI scaling / Set a custom high DPI scaling 1.5

Interface

Qt windows style: WindowsVista

Icon theme: Spyder 3

Figure 2.7: The Tools / Preferences menu item displays various settings of Spyder.

Here is an overview of the settings with comments:

- ***General***: Here you have ability to change font type and font size.
- ***Keyboard shortcuts***: No need to change anything here.
- ***Syntax coloring***: No need to change anything here.
- ***Python interpreter***: No need to change anything here.
- ***Run***: General settings: You may want to enable Remove all variables before execution. Thus, you can be pretty sure that a script that works now will also work if you run the script in another or new Python process (like after you closed and reopened Spyder to run the script).
- ***Current working directory***: No need to change anything here.
- ***Editor***: Display / Highlight occurences after: I suggest 100 ms so that Spyder quickly indicates different instances of a given variable in

the script. With the default setting of 1500 ms, Spyder seems unnecessarily slow in terms of such an indication. I also suggest enabling Real-time code style analysis, which means that a warning triangle will appear in the left part of the editor if the code-style rules as defined in PEP 8[2] wrap.

- ***IPython console***: Graphics / Backend: The default setting Inline means that plots (graphs) are displayed in the console. I usually change to Automatic which allows plots to appear in a separate window outside of Spyder, with various options for manipulating and editing the plot.
- ***History log***: No need to change anything here.
- ***Help***: Here I propose to enable Editor and IPython console, which means that information about a function is displayed in the Help window, see Figure 2.3, when you type a parenthesis after the function name, e.g. print().
- ***Variable explorer***: No need to change anything here.
- ***Profiler***: No need to change anything here.
- ***Static code analysis***: No need to change anything here.

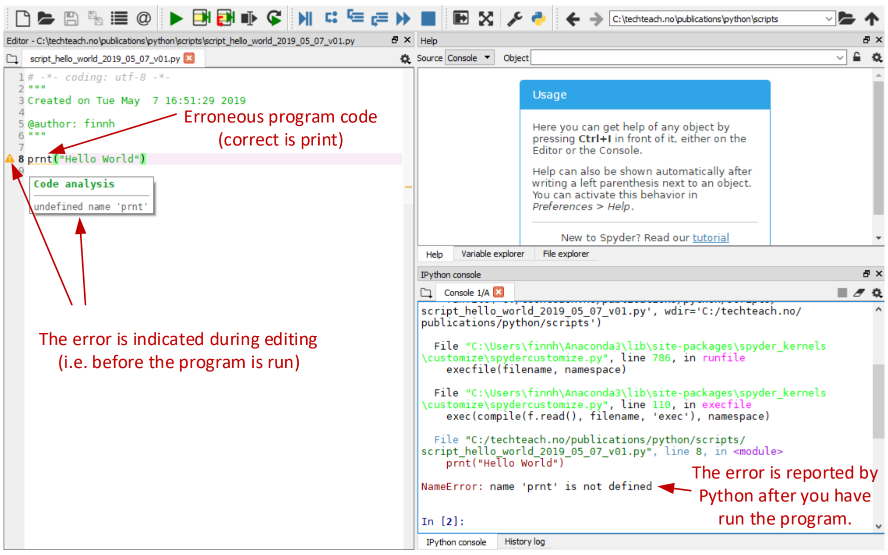
2.2.4 Help in Spyder

You totally depend on help when doing programming:

- ***Help find a suitable way to perform a programming task***, eg. help find a function that can calculate the square root of a number. Some alternative ways to get help are:
  - Textbooks
  - Documentation and tutorials on the internet, eg. on the official website of Python: http://python.org.
  - Google search, which often leads to hits http://stackoverflow.com.
  - Ask people, e.g. fellow students and fellow students and tutors and teachers

[2]PEP 8: Style Guide for Python Code, se https://www.python.org/dev/peps/pep-0008/. (PEP = Python Enhancement Proposals)

- ***Help find syntax errors in your program code***. A syntax error is a program technical error.
  Spyder finds syntax errors. Figure 2.8 shows an example where there is a syntax error in the program code. Unfortunately, I wrote prnt ('Hello World') instead of the correct print ('Hello World'). We see that Spyder already finds the bug in the editor *before* running the program, and also gives an error message in the console *after* the program is run.



Editor - C:\techteach.no\publications\python\scripts\script_hello_world_2019_05_07_v01.py

script_hello_world_2019_05_07_v01.py

```
# -*- coding: utf-8 -*-
"""
Created on Tue May  7 16:51:29 2019

@author: finnh
"""

prnt("Hello World")
```

```
Code analysis
undefined name 'prnt'
```

Usage

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Help*.

New to Spyder? Read our tutorial

```
script_hello_world_2019_05_07_v01.py', wdir='C:/techteach.no/
publications/python/scripts')

  File "C:\Users\finnh\Anaconda3\lib\site-packages\spyder_kernels
\customize\spydercustomize.py", line 786, in runfile
    execfile(filename, namespace)

  File "C:\Users\finnh\Anaconda3\lib\site-packages\spyder_kernels
\customize\spydercustomize.py", line 110, in execfile
    exec(compile(f.read(), filename, 'exec'), namespace)

  File "C:/techteach.no/publications/python/scripts/
script_hello_world_2019_05_07_v01.py", line 8, in <module>
    prnt("Hello World")

NameError: name 'prnt' is not defined


In [2]:
```

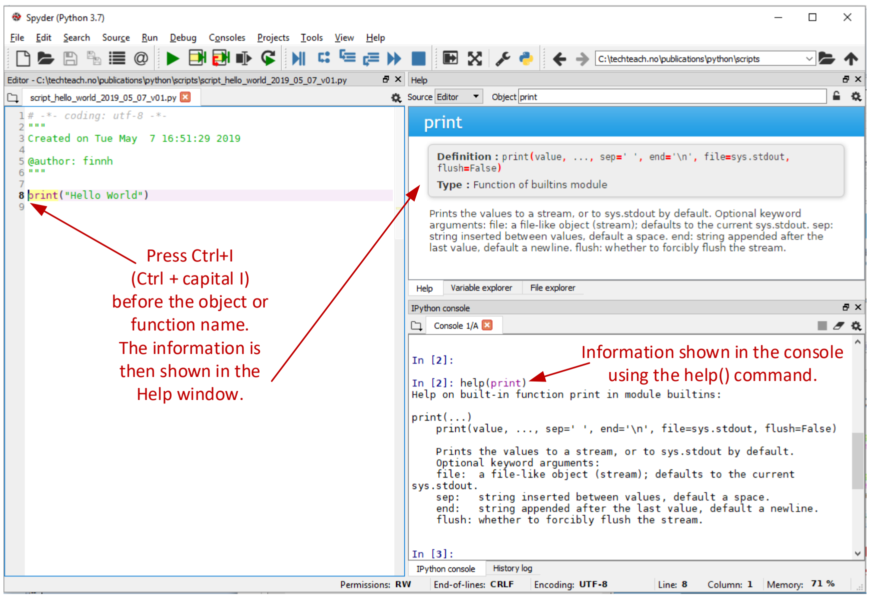
Figure 2.8: Spyder finds syntax errors.

Usually you use some built-in (pre-made) Python function in your programs, e.g. print() function. You can get information on how to use such functions, as mentioned above. Alternatively, you can get information about such functions – or general objects – directly in Spyder in two ways, see Figure 2.9:

- ***By pressing Ctrl + I (capital I) in front of the function name*** in the script (possibly on the command line). The information is then displayed in the Help window.
  Note: If the function you are using comes from a package or module that you have imported into Python using the import command (such import is discussed in Chap. 2.6), you must include the package or module name in front of the function name itself and press Ctrl + I in front of the package or module name. Example: np.sqrt(), where it is assumed that you have previously executed the

import numpy as np command. numpy is a package that contains the sqrt() function. Here you have to press Ctrl + I in front of np.sqrt().

- ***With the help() command executed in the console*** (but this information is not always easy to read).



```
# -*- coding: utf-8 -*-
"""
Created on Tue May  7 16:51:29 2019

@author: finnh
"""

print("Hello World")
```

print

Definition : print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Type : Function of builtins module

Prints the values to a stream, or to sys.stdout by default. Optional keyword arguments: file: a file-like object (stream); defaults to the current sys.stdout. sep: string inserted between values, default a space. end: string appended after the last value, default a newline. flush: whether to forcibly flush the stream.

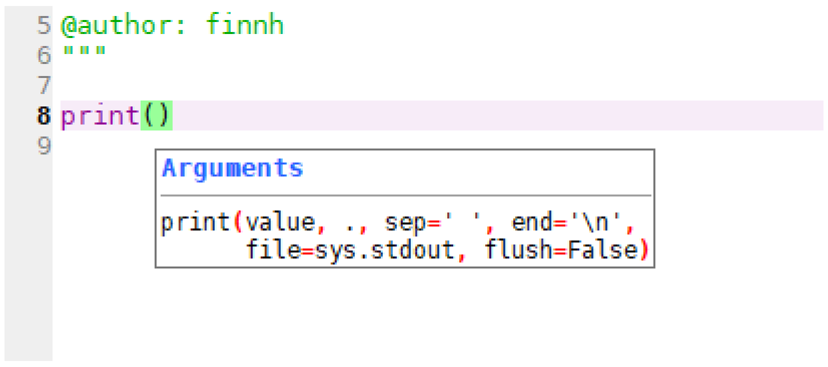
```
In [2]:

In [2]: help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current
sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.


In [3]:
```

Spyder (Python 3.7)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\techteach.no\publications\python\scripts
Editor - C:\techteach.no\publications\python\scripts\script_hello_world_2019_05_07_v01.py
script_hello_world_2019_05_07_v01.py
Help
Source Editor
Object print
Press Ctrl+I (Ctrl + capital I) before the object or function name. The information is then shown in the Help window.
Help Variable explorer File explorer
IPython console
Console 1/A
Information shown in the console using the help() command.
IPython console History log
Permissions: RW End-of-lines: CRLF Encoding: UTF-8 Line: 8 Column: 1 Memory: 71 %

Figure 2.9: Information about built-in functions can be displayed directly in Spyder in two ways: With Ctrl + I in front of the function name or using the help() command.

Instant information in Spyder about the syntax of function

You can get instant information in Spyder about the syntax of the function while writing the program code: As soon as you enter the function name and start typing the parentheses to enter the function arguments, the function syntax appears in a small field in the editor, see Figure 2.10.



```
@author: finnh
"""

print()
```

```
Arguments
print(value, .., sep=' ', end='\n',
      file=sys.stdout, flush=False)
```

Figure 2.10: In Spyder, the function syntax is displayed in a small field in the editor as soon as you enter the function name and begin writing the parentheses for the input arguments.

2.3 Jupyter Notebook

Jupyter Notebook is a Python programming environment that can be used in a web browser. In Jupyter Notebook we can create and run Notebook documents[3], which may contain a mixture of Python program code and information (not program code). Notebook documents can be used locally on the user's PC or over the Internet. Jyputer Notebook runs Notebook documents using. a built-in program called kernel . We will see how we can create Notebook documents running Python, but there is support for other languages as well, e.g. R which is widely used in statistics.

In this chapter we will only look at the very basic use of the Jupyter Notebook.

2.3.1 How to start Jupyter Notebook

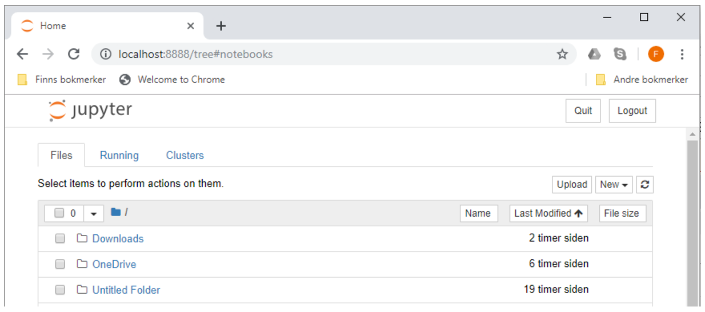
We can start Jupyter Notebook in two ways:

- Via the Jupyter Notebook button in Anaconda Navigator, see Figure 2.2.
- Via the Jupyter Notebook button under Anaconda on the PC's start menu, see Figure 2.1.

[3]We can simply refer to Jupyter Notebook documents as Notebook documents.

> You can also make Jupyter Notebook accessible via a button on your PC's taskbar: Right-click the Jupyter Notebook icon in the Start menu / More / Attach to the taskbar.

When Jupyter Notebook is started, Jupyter Notebook's dashboard appears in a tab of the browser, see Figure 2.11.



Select items to perform actions on them.

Downloads	2 timer siden
OneDrive	6 timer siden
Untitled Folder	19 timer siden

Figure 2.11: The Jupyter Notebook Dashboard in a browser tab

The dashboard has three tabs:

- ***Files***, which works like Window Explorer on a Windows PC.
- ***Running***, which displays Notebook documents that are being processed, i.e. running on the current kernel (Python, R, or another). From here you can stop any (Python) processes running.
- ***Clusters***, which applies to the use of parallel processing (simultaneous processing) on multiple cores with a tool called IPython Parallel, but we will not go into this.

2.3.2 How to create and edit Notebook documents

To create a new Jupyter Notebook document, click the New button on the dashboard, cf. Figure 2.11, and choose the current kernel, which for our use is Python 3 (by the way the only kernel available here). This opens a new Notebook document in a new tab in the browser to the right of the dashboard tab. The notebook is initially given a default name, here Untitled, but you can change this by clicking on the name field or via the File / Rename menu option.

Figure 2.12 Notebook document after renaming hello_world_05_08_v01.

The address in the browser's address bar is:

localhost:8888/notebooks/hello_world_05_08_v01.ipynb

Some comments on this address:

- localhost indicates that Jupyter Notebook is running on a local kernel, ie on the user's PC.
- Notebook document file extension, ie the part of the file name that indicates the file type is ipynb (= interactive python notebook). However, the file extension does not appear in the title field of the Notebook document itself, which is to the right of the Jupyter icon.

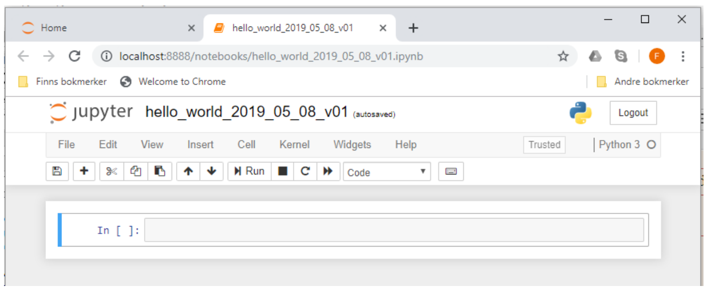


Figure 2.12: Notebook document hello_world_05_08_v01.ipynb (currently free of content) displayed in Notebook tab in browser

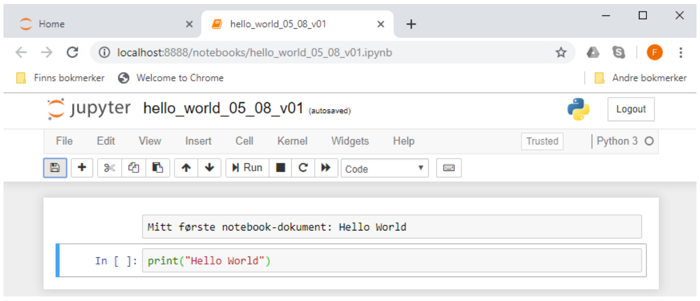
We will now edit the Notebook document by typing into cells in the document:

1. Click in the currently only cell and make the following menu choices: Cell / Cell Type / Markdown, which means that the text we are going to write in this cell is informative text, ie not program code. Note that the characters In []: now no longer appear to the left of the cell.
2. Type the following text in the cell: My first Notebook document: Hello World.
3. Create a new cell during the first one by clicking the + button (Insert Cell Below button) or with the following menu items: Insert / Insert Cell Below.

4. Click in the newly created cell and make the following menu choices: Cell / Cell Type / Code, which means that the text we are going to write in this cell is program code, ie not informative text. The characters In []: are now displayed to the left of the cell.

5. Enter the following text in the newly created cell: print ('Hello World').

6. Save the Notebook document with the Save button (Save and Checkpoint button) or via the menu option File / Save and Checkpoint. Checkpoint represents the latest version of the document that we saved manually. Checkpoint neglects file versions that were saved with auto-save. You can get the Checkpoint version of the file back with the File / Revert to Checkpoint menu option. Figure 2.13 displays the Notebook document as it has now, ready to run.

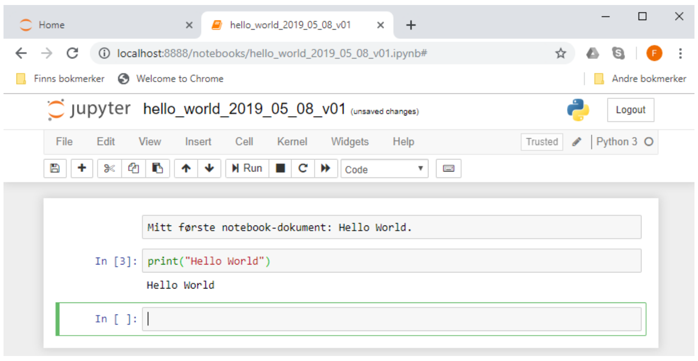

Home
hello_world_05_08_v01
localhost:8888/notebooks/hello_world_05_08_v01.ipynb
Finns bokmerker
Welcome to Chrome
Andre bokmerker
jupyter hello_world_05_08_v01 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
Mitt første notebook-dokument: Hello World

In [ ]: print("Hello World")
```

Figure 2.13: Notebook document ready to run

2.3.3 How to run Notebook documents

We can run the Notebook document shown in Figure 2.13 by clicking the Run button or by making the Run / Run All menu selection. Figure 2.14. shows the result of the run.

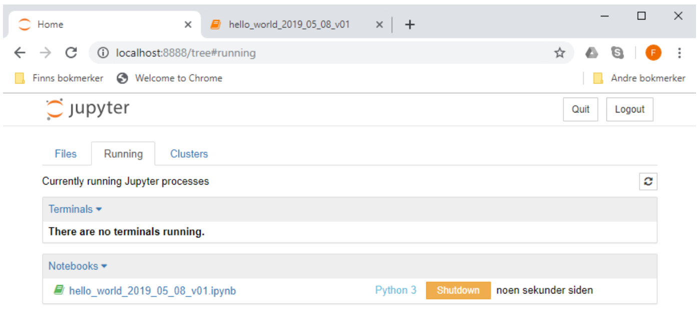

Home
hello_world_2019_05_08_v01
localhost:8888/notebooks/hello_world_2019_05_08_v01.ipynb#
Finns bokmerker
Welcome to Chrome
Andre bokmerker
jupyter hello_world_2019_05_08_v01 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
Mitt første notebook-dokument: Hello World.

In [3]: print("Hello World")

Hello World

In [ ]:
```

Figure 2.14: The result of the Notebook document running

Once we have started running a Notebook document, the process will continue to run - until we manually stop it, which is done as follows:

1. Open the Dashboard (tab to the left of the Notebook document tab) in the browser.
2. Open the Running tab.
3. Click the Shutdown button, see Figure 2.15.



Currently running Jupyter processes

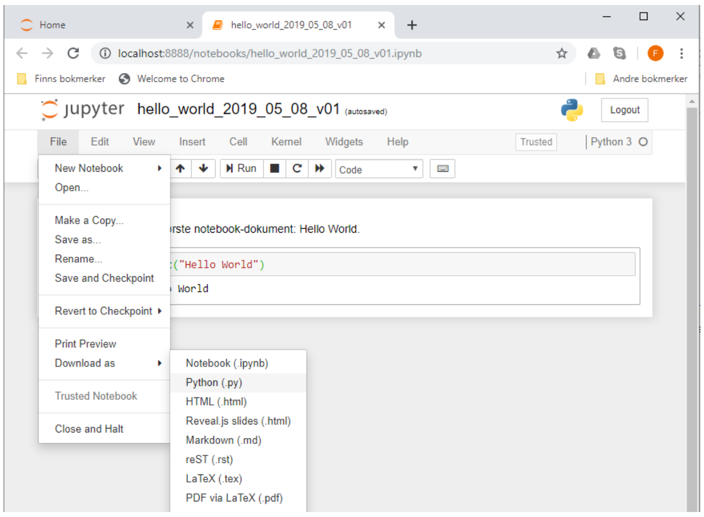
There are no terminals running.

hello_world_2019_05_08_v01.ipynb   Python 3   Shutdown   noen sekunder siden

Figure 2.15: The dashboard that displays the Notebook document hello_world_05_08_v01.ipynb, which is running.

2.3.4 How to Save Notebook Documents

Notebook documents can be stored - or "downloaded" (download) from the server running the Python process - in a variety of alternative file formats. This is done via the menu option File / Download As ..., see Figure 2.16.



rste notebook-dokument: Hello World.

```
:("Hello World")
```

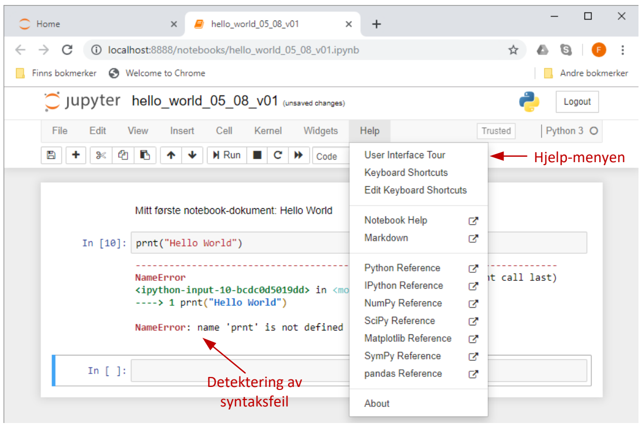
```
World
```

Figure 2.16: Notebook documents can be saved - or "downloaded" (in a variety of alternative file formats).

2.3.5 Help in Jupyter Notebook

There is help getting in the Jupyter Notebook:

- ***Help find a suitable way to perform a programming task***. The possibilities for such assistance are the same as for Spyder, cf. 2.2.4 (they are not repeated here). There is also a rich Help menu in the Jupyter Notebook, see Figure 2.17.
- ***Help find syntax errors*2.2.4 *in your program code***, as illustrated in Figure 2.17 where the syntax error in prnt ("Hello World") is detected.



Mitt første notebook-dokument: Hello World

```
In [10]: prnt("Hello World")

         ---------------------------------------          ------------
         NameError                                        t call last)
         <ipython-input-10-bcdc0d5019dd> in <mo
         ----> 1 prnt("Hello World")

         NameError: name 'prnt' is not defined

In [ ]:
```

Figure 2.17: Different types of help in Jupyter Notebook: Programming Help (Guide) under the Help menu and Detecting Syntax Errors

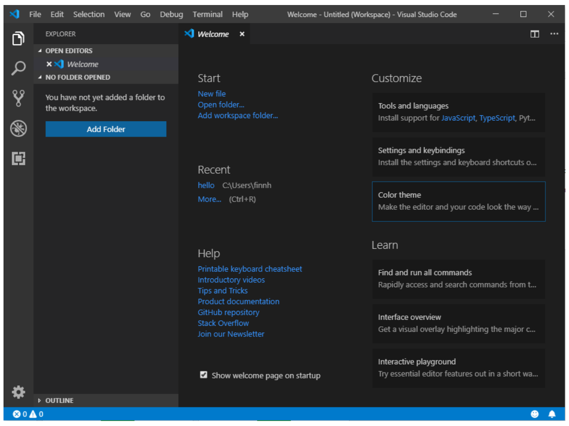
2.4 Visual Studio Code

2.4.1 How to install and launch Visual Studio Code

Visual Studio Code or just VS Code (Microsoft) is a freely available programming environment with support for a variety of programming languages, including Python. VS Code can be downloaded from https://code.visualstudio.com, but is also included with the Anaconda distribution and is then available via a separate button in Anaconda Navigator, see Figure 2.2.

If you have installed the Anaconda distribution, VS Code is also available through the PC Start menu. I recommend starting VS Code via Anaconda Navigator because all Python packages that come with the Anaconda distribution will then be available in Python via the import command executed in the VS Code terminal (see below) or in your Python script.

Figure 2.18 shows the VS Code startup window. (The window may look a little different on your PC.)



You have not yet added a folder to the workspace.

Install support for JavaScript, TypeScript, Pyt...

Install the settings and keyboard shortcuts o...

Make the editor and your code look the way ...

Rapidly access and search commands from t...

Get a visual overlay highlighting the major c...

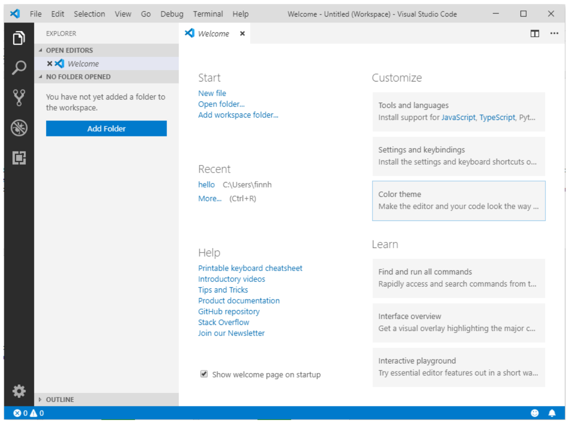
Try essential editor features out in a short wa...

Figure 2.18: Startup window in VS Code

You may disagree, but I think it's a bit dull with such a dark programming environment. I prefer more light and therefore make this menu selection:

File / Preferences / Color Theme / Light (Visual Studio)

Figure 2.19 shows again the startup window in VS Code, now with the Light color them.



You have not yet added a folder to the workspace.

Install support for JavaScript, TypeScript, Pyt...

Install the settings and keyboard shortcuts o...

Make the editor and your code look the way ...

Rapidly access and search commands from t...

Get a visual overlay highlighting the major c...

Try essential editor features out in a short wa...

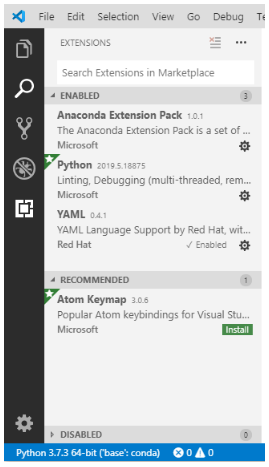
Figure 2.19: The VS Code startup window after the File / Preferences / Color Theme / Light (Visual Studio) menu option

2.4.2 Connect Python to VS Code

VS Code can be used to program and run programs in a variety of languages. We will now connect Python to VS Code so that we can run

Python programs from VS Code:

1. Select the View Extensions menu option (or click the Extensions button in the button bar shown on the left of Figure 2.19), which automatically opens and selects Anaconda Extensions and Python and YAML.[4] Python is now entered in the status bar at the bottom of the VS Code window, see Figure.



Anaconda Extension Pack 1.0.1
The Anaconda Extension Pack is a set of ...
Microsoft

Python 2019.5.18875
Linting, Debugging (multi-threaded, rem...
Microsoft

YAML 0.4.1
YAML Language Support by Red Hat, wit...
Red Hat ✓ Enabled

Atom Keymap 3.0.6
Popular Atom keybindings for Visual Stu...
Microsoft

Python 3.7.3 64-bit ('base': conda) 0 0

Figure 2.20: The View Extensions menu option opens and selects Anaconda Extensions and Python and YAML. (We keep the choices.)

2.4.3 Open (create) workspace

We are going to create a simple Python program, which we will store in a folder that belongs to a workspace. Various settings of VS Code are stored in the current workspace.

We start by opening a new workspace with the menu selection

File / Save Workspace As

This opens a file explorer window called Save Workspace. In this window, create or select an existing folder. Then create a workspace associated with this folder by entering a self-selected Workspace name (I chose the name

[4]YAML = YAML Ain't Markup Language, which is a recursive name definition that can be difficult to understand :-) YAML is used to create configuration files of various types.

vs_workspace_finnh) and clicking the Save button. Figure 2.21 shows a section of the VS Code window where the newly created workspace is displayed under the Explorer view. (You can access the Explorer view using the View / Explorer menu option or by clicking the Explorer button in the upper left of the VS Code window.)

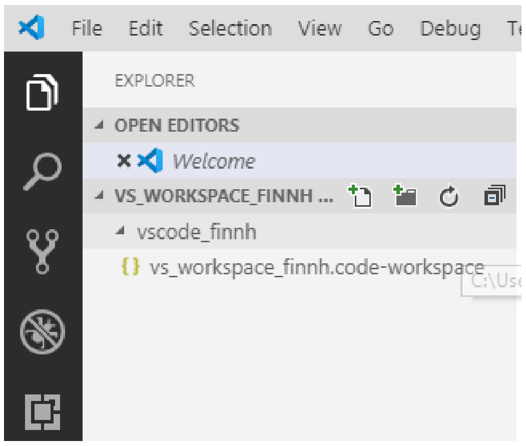
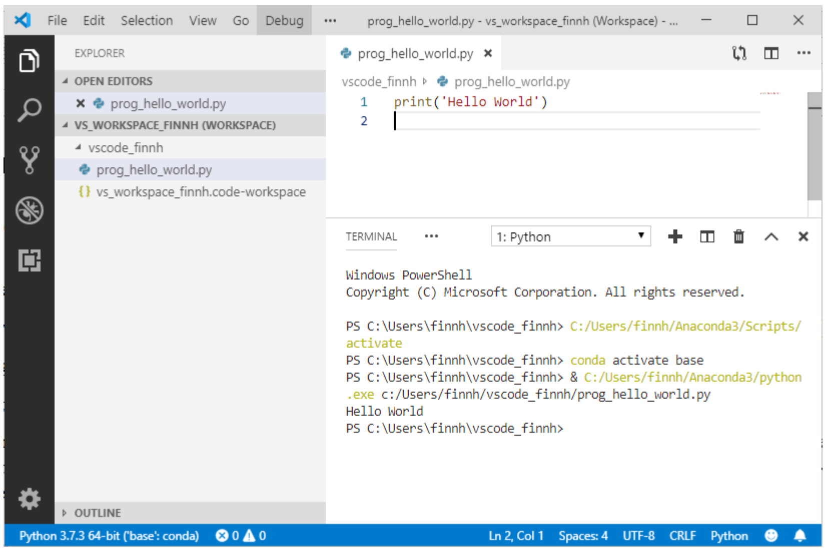


Figure 2.21: VS Code window where the newly created workspace (vs_workspace_finnh) is displayed under the Explorer view

2.4.4 How to create and run a Python program

We can create a Python program as follows, cf. Figure 2.21:

1. Make the menu view View / Explorer.
2. Right-click on the folder name (vscode_finnh) that is listed under the workspace name (vs_workspace_finnh).
3. Velg New File i menyen som åpnes.
4. Enter a file name. I have chosen prog_hello_world.py.
5. Run the program by right-clicking somewhere in the editor window of the current Python program and selecting Run Python File in Terminal. (Alternatively, you can right-click the current Python program in the Explorer window to the left of the editor window.) Figure 2.22 shows the result of the run.



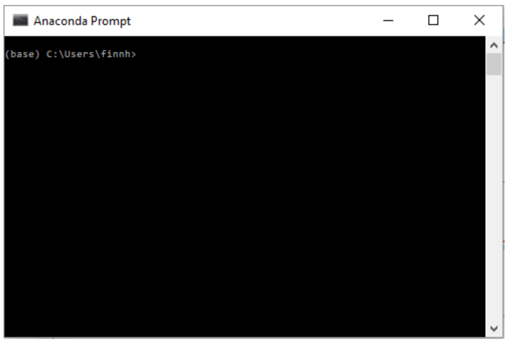
```
print('Hello World')
```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\finnh\vscode_finnh> C:/Users/finnh/Anaconda3/Scripts/
activate
PS C:\Users\finnh\vscode_finnh> conda activate base
PS C:\Users\finnh\vscode_finnh> & C:/Users/finnh/Anaconda3/python
.exe c:/Users/finnh/vscode_finnh/prog_hello_world.py
Hello World
PS C:\Users\finnh\vscode_finnh>
```

Figure 2.22: The result of running the Python program prog_hello_world.py via the menu item Run Python File in Terminal.

2.5 The Python command line in the Anaconda command window

Figure 2.23 displays the Anaconda command window, which can be accessed via the Anaconda menu on the PC start menu.

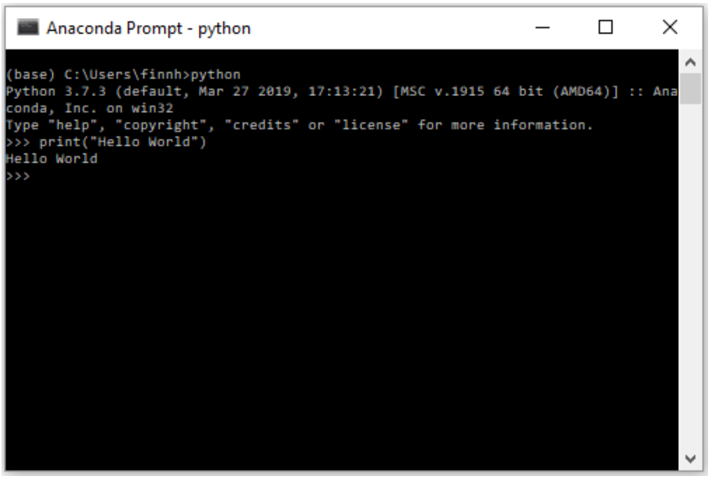


```
(base) C:\Users\finnh>
```

Figure 2.23: The Anaconda command window

We can open the Python command line by typing python (and ending with the Enter key). We can then write the program code on the command line and execute the code by pressing the Enter key on the keyboard.

Figure 2.24 shows the result of executing the program code print ('Hello World') on the Python command line in the Anaconda command window.



```
(base) C:\Users\finnh>python
Python 3.7.3 (default, Mar 27 2019, 17:13:21) [MSC v.1915 64 bit (AMD64)] :: Ana
conda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World")
Hello World
>>>
```

Figure 2.24: The result of executing the program code print ('Hello World') on the Python command line in the Anaconda command window.

Generally, it is more appropriate to use Spyder or Jupyter Notebook than the Python command line as a programming environment. But even if we do not use the Anaconda command window for programming itself, we can greatly benefit from the Anaconda command window for eg. to manage so-called packages of Python functions. By package management is meant listing of already installed packages (on your PC), searching for packages that are not yet installed, and installing packages. We will discuss this in more detail in Chap. 2.6.

2.6 Import and use of Python packages and modules

2.6.1 Packages management with conda or pip

Packages

functions that we can use in our Python programs, e.g. print(), sqrt(), sum(), etc., are aggregated into packages for special purposes. Here are a few examples of such packages:

- numpy, which contains elementary mathematical functions

- matplotlib, which contains data plotting functions
- scipy, which contains advanced mathematical functions for eg. optimization and signal processing, etc.
- pygame, which contains functions for programming animations

The packages come in different shapes:

- One package, which we can call the standard package, is automatically installed when the Anaconda deployment is installed, and the functions there are immediately available for use in the programs we create.
- Many packages are installed automatically when the Anaconda distribution is installed. We must also *import* them to Python in order to use them.
- And there are many packages "out there" that are not included with the Anaconda distribution. If we need functions in some such packages, we must install them ourselves and then import them into Python.

Package Management

There are tools available for package management or package management. Such package management can be listing, installation, uninstalling packages, etc. Once we install the Anaconda distribution, we have two package management tools available:

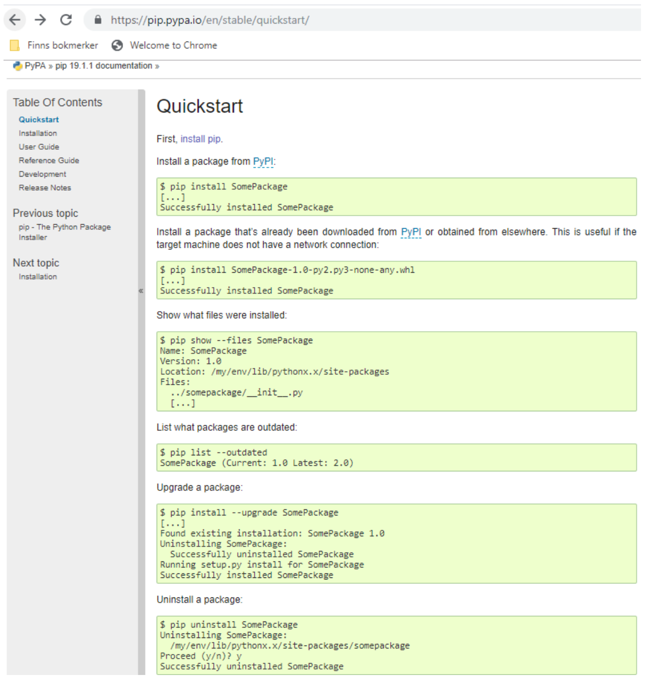
- ***conda*** , which is an Anaconda product.
- ***pip*** , which is quite similar to conda, but is a tool developed by PyPA - The Python Packaging Authority - which is a working group that develops and maintains tools for Python packages. Also pip is part of the Anaconda distribution.[5]

[5]

– Pip is supposedly an abbreviation for “pip installs packages”, ie Pip is a so-called recursive acronym (abbreviation). This is not a particularly clear definition, I think :-) Alternatively, we can consider pip as an abbreviation for “Python installation package” or “package management system used to install and manage software packages written in Python” (Wikipedia).

Both conda and pip can be used in Anaconda command window, cfr. Ch.2.5. They can also be used from the command line in Spyder, but I would not recommend this as there is sometimes little or no running information displayed in Spyder while pip or conda is running (eg installing or uninstalling a package), and I have also experienced that commands seemingly hanging. The Anaconda command window is better in that way, but should we use conda or pip there? My experience is that conda has not always succeeded with the job, while pip always worked. Then I land on: pip in the Anaconda window.

Figure 2.25 displays some basic pip commands. (On the Anaconda command line, we do not need to type the § character.)



Quickstart

First, install pip.

Install a package from PyPI:

```
$ pip install SomePackage
[...]
Successfully installed SomePackage
```

Install a package that's already been downloaded from PyPI or obtained from elsewhere. This is useful if the target machine does not have a network connection:

```
$ pip install SomePackage-1.0-py2.py3-none-any.whl
[...]
Successfully installed SomePackage
```

Show what files were installed:

```
$ pip show --files SomePackage
Name: SomePackage
Version: 1.0
Location: /my/env/lib/pythonx.x/site-packages
Files:
  ../somepackage/__init__.py
  [...]
```

List what packages are outdated:

```
$ pip list --outdated
SomePackage (Current: 1.0 Latest: 2.0)
```

Upgrade a package:

```
$ pip install --upgrade SomePackage
[...]
Found existing installation: SomePackage 1.0
Uninstalling SomePackage:
  Successfully uninstalled SomePackage
Running setup.py install for SomePackage
Successfully installed SomePackage
```

Uninstall a package:

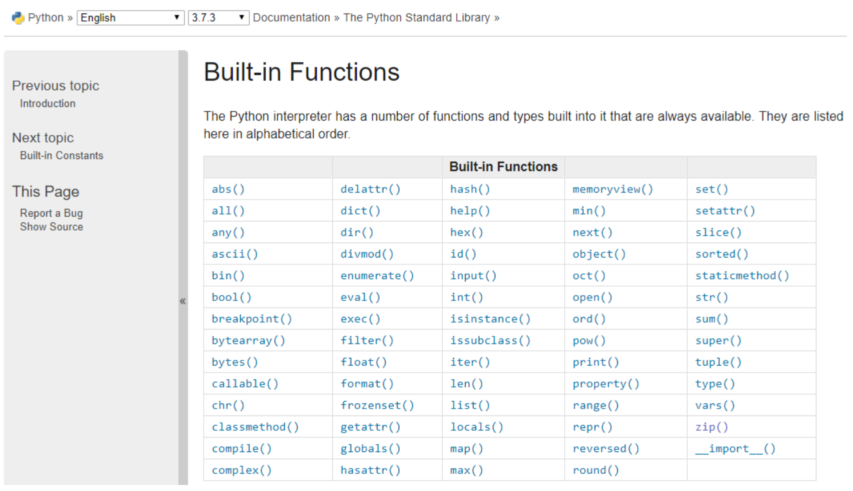
```
$ pip uninstall SomePackage
Uninstalling SomePackage:
  /my/env/lib/pythonx.x/site-packages/somepackage
Proceed (y/n)? y
Successfully uninstalled SomePackage
```

Figure 2.25: Some basic pip commands. (Https://pip.pypa.io/en/stable/quickstart/)

We will use pip commands in some of the following sections.

2.6.2 Built-in functions in Python (standard package)

Python is said to come with battery included, which means that in Python there is a collection of built-in functions that you can use in your programs. We can consider this collection of built-in functions as Python's standard package of functions. An overview of the built-in functions in Python version 3.7.3 is available at https://docs.python.org/3/library/functions.html, see Figure 2.26. We see that the well-known print() function is in the standard package.


Python » English 3.7.3 Documentation » The Python Standard Library »

Previous topic
Introduction

Next topic
Built-in Constants

This Page
Report a Bug
Show Source

Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

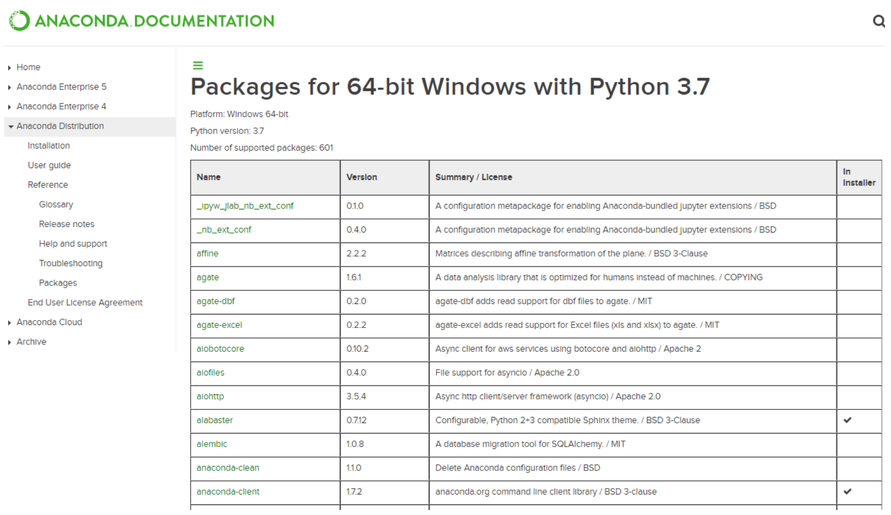
		Built-in Functions		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Figure 2.26: Pythons standard package of built-in functions (Python version 3.7.3)

2.6.3 Import of packages included with the Anaconda distribution

When we install the Anaconda distribution, a large number of function packages are automatically installed on your PC. These packages are (automatically) installed in addition to the standard package discussed in Chap. 2.6.2.

Figure 2.27 shows an excerpt of an overview of the 601 packages available for Python version 3.7.3, Windows 64 bits. (Source: https://docs.anaconda.com/anaconda/packages/py3.7_win-64/.) Of these packages, slightly less than half (that is, a few hundred) of packages are automatically installed when the Anaconda distribution is installed. The packages that are installed are marked with a check mark in the overview.



ANACONDA DOCUMENTATION

- Home
- Anaconda Enterprise 5
- Anaconda Enterprise 4
- Anaconda Distribution
  - Installation
  - User guide
  - Reference
    - Glossary
    - Release notes
    - Help and support
    - Troubleshooting
    - Packages
  - End User License Agreement
- Anaconda Cloud
- Archive

Packages for 64-bit Windows with Python 3.7

Platform: Windows 64-bit

Python version: 3.7

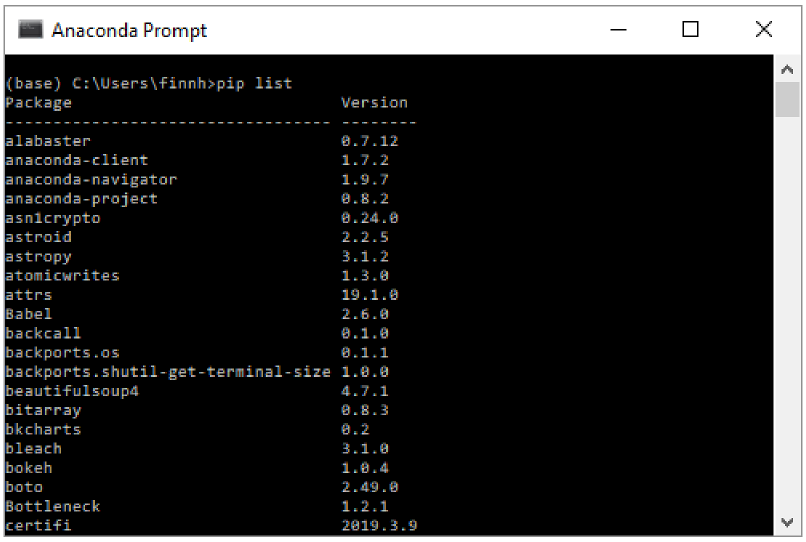
Number of supported packages: 601

Name	Version	Summary / License	In Installer
_ipyw_jlab_nb_ext_conf	0.1.0	A configuration metapackage for enabling Anaconda-bundled jupyter extensions / BSD	
_nb_ext_conf	0.4.0	A configuration metapackage for enabling Anaconda-bundled jupyter extensions / BSD	
affine	2.2.2	Matrices describing affine transformation of the plane. / BSD 3-Clause	
agate	1.6.1	A data analysis library that is optimized for humans instead of machines. / COPYING	
agate-dbf	0.2.0	agate-dbf adds read support for dbf files to agate. / MIT	
agate-excel	0.2.2	agate-excel adds read support for Excel files (xls and xlsx) to agate. / MIT	
aiobotocore	0.10.2	Async client for aws services using botocore and aiohttp / Apache 2	
aiofiles	0.4.0	File support for asyncio / Apache 2.0	
aiohttp	3.5.4	Async http client/server framework (asyncio) / Apache 2.0	
alabaster	0.7.12	Configurable, Python 2+3 compatible Sphinx theme. / BSD 3-Clause	✓
alembic	1.0.8	A database migration tool for SQLAlchemy. / MIT	
anaconda-clean	1.1.0	Delete Anaconda configuration files / BSD	
anaconda-client	1.7.2	anaconda.org command line client library / BSD 3-clause	✓

Figure 2.27: Packages that are installed with the Anaconda distribution for 64-bit Windows are marked with a check mark.
(https://docs.anaconda.com/anaconda/packages/py3.7_win-64/)

List of packages installed on your PC

The pip command list can be used to list packages that are installed on the PC. Figure 2.28 shows the result of the pip list command executed on the Anaconda command line.

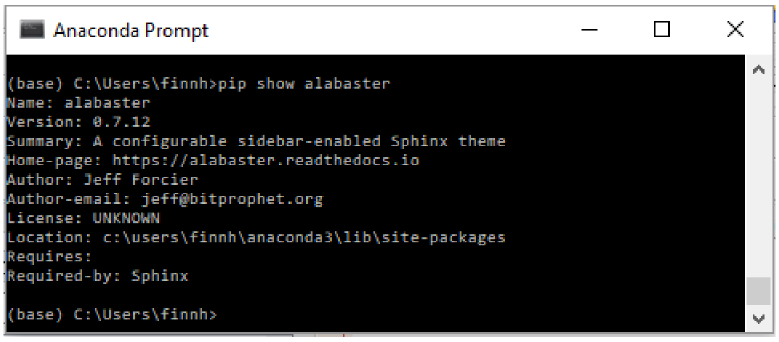


Anaconda Prompt

```
(base) C:\Users\finnh>pip list
Package                            Version
---------------------------------- --------
alabaster                          0.7.12
anaconda-client                    1.7.2
anaconda-navigator                 1.9.7
anaconda-project                   0.8.2
asn1crypto                         0.24.0
astroid                            2.2.5
astropy                            3.1.2
atomicwrites                       1.3.0
attrs                              19.1.0
Babel                              2.6.0
backcall                           0.1.0
backports.os                       0.1.1
backports.shutil-get-terminal-size 1.0.0
beautifulsoup4                     4.7.1
bitarray                           0.8.3
bkcharts                           0.2
bleach                             3.1.0
bokeh                              1.0.4
boto                               2.49.0
Bottleneck                         1.2.1
certifi                            2019.3.9
```

Figure 2.28: The result of the pip list command executed on the Anaconda command line

To check if a specific package is installed and possibly get information about the package, you can use the command show show. Figure 2.29 shows, for example, the result of the command pip show alabaster (alabaster is at the top of the package overview shown in Figure 2.28).



```
(base) C:\Users\finnh>pip show alabaster
Name: alabaster
Version: 0.7.12
Summary: A configurable sidebar-enabled Sphinx theme
Home-page: https://alabaster.readthedocs.io
Author: Jeff Forcier
Author-email: jeff@bitprophet.org
License: UNKNOWN
Location: c:\users\finnh\anaconda3\lib\site-packages
Requires:
Required-by: Sphinx

(base) C:\Users\finnh>
```

Figure 2.29: The result of the pip command show affine after it is executed on the Anaconda command line

How to import packages installed on your PC

Although a package is installed on the PC, the functions of the package are not immediately available in the programs we create. They become available by so-called importing that package. The usual way to import is to use the import command in your Python script.

An example: Suppose we calculate the square root of a number represented by the variable name x. The square root is represented by the variable name y and must be calculated by the sqrt() function included in the numpy package. The result (which is 1.5811388300841898) is displayed in the console by typing y on the command line followed by the Enter key. The code lines in the box below realize this (end each code line with the Enter key).

```
>>> import numpy as np
>>> x = 2.5
>>> y = np.sqrt(x)
>>> y
1.5811388300841898
```

Comments on the program code above:

- The numpy package is imported into Python and in this context is given the short name np (it is Python tradition to use the short name np for numpy).

- The import command should be followed by two blank lines, cf. the recommendation in PEP 8.
- The package name, here np, must be set as the prefix to the sqrt() function, cf. the code np.sqrt (x).
- If we had imported numpy with the code import numpy, we should have written numpy.sqrt (x) instead of np.sqrt (x). Actually, we can choose whether or not to rename a package. import, but I think we should follow name traditions in Python programming.

Although the main focus here is really the import of packages, it fits with some comments that do not have to do with packages:

- x is a variable, given value 2.5. (We shall take a closer look at the term variable in Chap. 3.3.)
- y is variable, which gets value equal to the square root of x.

How to import modules included in packages

In some cases, we need to import so-called modules which is included in packages. A module is in principle a collection of functions. A package can then contain a number of modules, which in turn consists of a number of functions.

Although we will look more closely at the plotting of data in Chap. 4, here it may be appropriate to use a concrete example of plotting to illustrate the import of modules: We will import the pyplot module from the matlplotlib package and use the plot (plot) function included in the pyplot module to plot data. The code lines in the box below do this.

```
>>> import matplotlib.pyplot as plt
>>> x = [0, 1, 2]
>>> y = [0, 10, 20]
>>> plt.plot(x,y,'-o')
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.grid(which='both',color='grey')
>>> plt.show()
```

Comments on the code above:

- The code line import matplotlib.pyplot as plt imports the pyplot module into the matplotlib library and renames pyplot to plt in that regard (it is Python tradition to use the short name plt for pyplot).

- The code lines x = [0, 1, 2] and y = [0, 10, 20] define the lists x respectively y.
- The pyplot function calls plt.plot (x, y, '- o') generate the plot itself. Note that plt is in front of the plot() function. The code states:
  - y is plotted vs. x
  - Each data point, ie each (x, y) point is marked with a circle.
  - A straight line is drawn between the data points.
- The pyplot function calls plt.xlabel ('x') and plt.ylabel ('y') specify mark text along the axes.
- The pyplot function call plt.grid (which = 'both', color = 'gray') generates grids in the plot.
- The Pyplot function call plt.show() ensures that the plot is displayed. (In Spyder, this function shell can actually be dropped. The plot is displayed anyway.)

Figure 2.30 shows the plot.

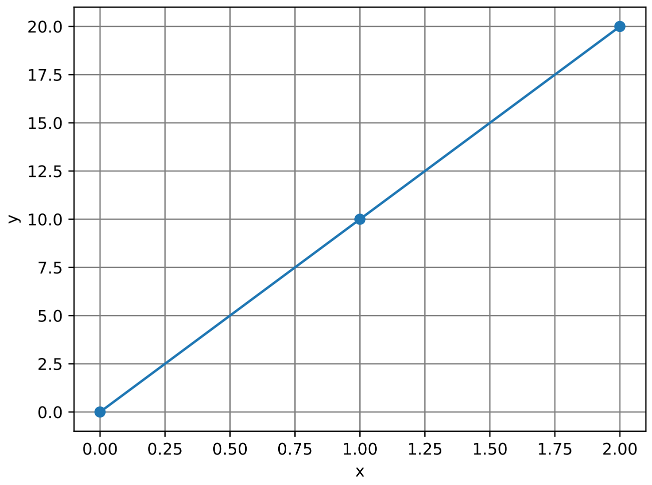
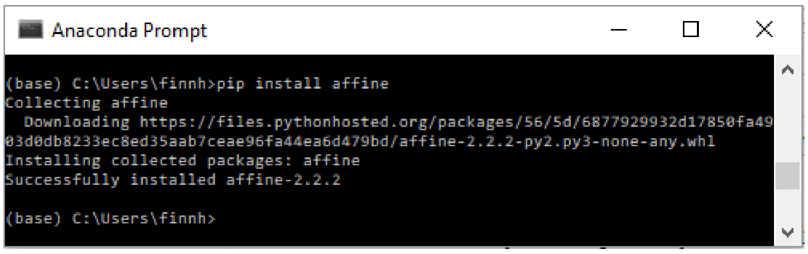


Figure 2.30: Plot of data with plot() function of the plt module (pyplot module) in the matplotlib package

2.6.4 Installation and import of packages not included with the Anaconda distribution

If you know the name of a Python package you need, and that package does not come with the Anaconda distribution, you can install the package with the pip install command, see Figure 2.25. Once you have installed the package, you can import it with the import command, cf. 2.6.3.

As an example, let's install affine (although we actually don't need it now). Figure 2.31 shows the result of the pip install command executed in the Anaconda command window.



```
(base) C:\Users\finnh>pip install affine
Collecting affine
  Downloading https://files.pythonhosted.org/packages/56/5d/6877929932d17850fa49
03d0db8233ec8ed35aab7ceae96fa44ea6d479bd/affine-2.2.2-py2.py3-none-any.whl
Installing collected packages: affine
Successfully installed affine-2.2.2

(base) C:\Users\finnh>
```

Figure 2.31: The result of the pip install command executed in the Anaconda command window

Chapter 3

Variables and data types

3.1 Introduction

You have to understand what variables and data types are, in order to program. Of course, programming requires knowledge of many other things as well, but variables and data types are fundamental concepts. This chapter provides sufficient knowledge of variables and data types.

3.2 How to run the code samples?

The chapter, and the rest of the book, contains many examples of program code that you can / should run yourself. The code examples are set in frames. In general, you can choose whether to run the commands from the command line in e.g. Spyder or via a script that you create. Small examples, which I suppose can be tested on the command line (without creating scripts), I type after the so-called Python prompt (or the Phyton command sign) >>>.

In the programming environments Jupyter Notebook and Spyder are prompted

In [n]

(there is a running command number) instead

>>>

If you use one of these programming environments, you enter the code after the In [n] prompt.

After you enter the code on the command line, run the code by pressing the Enter key on the keyboard.

3.3 Variables

3.3.1 What is a variable?

A variable is a data element with a name. We can create or define a variable and assign it a value by using equals. If the number value has a unit, you must keep track of the unit yourself.

Example: We define the variable m (for "mass") and assign the number value 10.2, which we assume has unit kg. We can then write the following in the current script or on the console command line:

```
>>> m = 10.2  # kg
```

Comments:

- In Python we must use decimal point, not decimal comma.
- We can specify the unit of value as text after the # sign, which indicates line comments. All characters after this character are interpreted by Python as a comment and neglected as program code.
- After the code is executed, the variable m exists in Python's workspace (workspace), as shown in the Variable explorer in the Spyder Help window, see Figure 3.1.
- We can delete the variable m with the command del m. We can delete all the variables in the work area with the eraser button in the console or in the Variable explorer, see Figure 3.1.

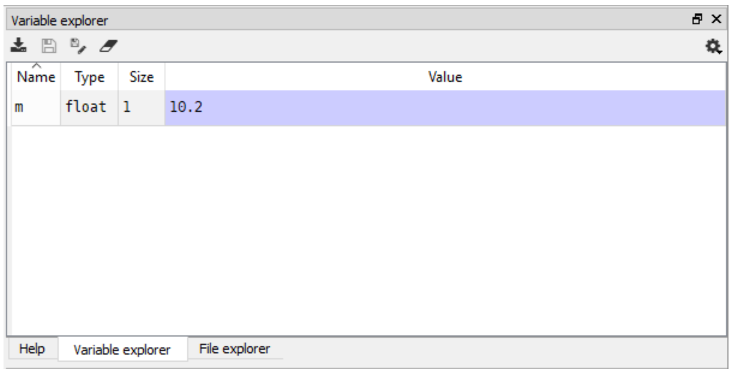
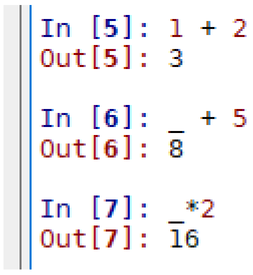


Figure 3.1: After executing the code, the variable m exists in Python's workspace, as shown in the Variable explorer in the Spyder help window.

The built-in variable _ (underscore) has value from the last calculation

For example, suppose you performed the 1 + 2 calculation on the Python command line (in the console). The value of the calculation is 3. Python has a built-in variable named _, which we can call the underscore variable. The underscore variable gets a value here of 3. You can then use the underscore variable, which has a value of 3 here, in the subsequent calculation. Figure 3.2 illustrates the use of the underscore variable.



```
In [5]: 1 + 2
Out[5]: 3

In [6]: _ + 5
Out[6]: 8

In [7]: _*2
Out[7]: 16
```

Figure 3.2: The built-in underscore variable has value equal to the result of the last calculation.

3.3.2 Why use variables when you can always use *values*?

Stating the conclusion right away: It is wise to use variables in programs! For example, suppose we have a program where the constant (constant) value is 1.23 is included in 10 places in the program. Two alternative ways to use this value in the program are:

1. We can write the value 1.23 at all 10 places in the program.
2. We can define a variable let's say with name a and assign that value 1.23 one place in the program, ie we write the code a = 1.23, we also use a at all 10 places in the program.

It is almost as easy to write a as 1.23, so then it doesn't matter which of the two options we choose? Assume that, for some reason, the value is changes to 2.46. With method 1 we must make 10 changes in the program, while with method 2 we can do with only 1 change, namely a = 2.46! This is illustrated in Figure 3.3.

Another benefit of using variables instead of just values is that the program code becomes easier to maintain, ie that we are less likely to create the wrong code. In addition, the program code becomes easier to read.

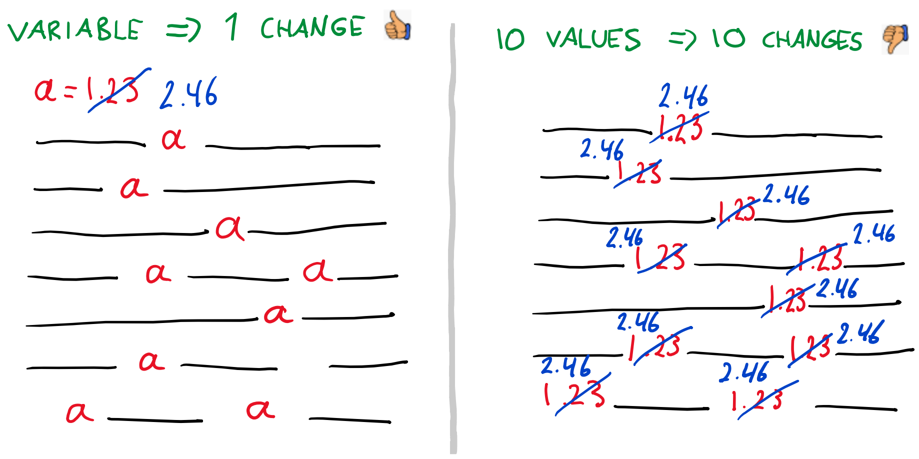


Figure 3.3: It is much better to use variables than values in the program code. (The lines represent program code here.)

3.3.3 How to choose variable name

There are actually only a few absolute rules for variable names. Below are some absolute rules and some recommendations.

- We should not use the characters l (lowercase l), o (lowercase o), O (capital o), I (capital i) alone or first in the name of variables or functions because these characters can so easily be misunderstood.
- It is ok to use both lowercase and capital letters, e.g. m or M for mass, but remember that m and M are different names.

- In names or parts of multi-letter names, lowercase letters should be used, e.g. mass, not MASS, nor Mass.
- The names may consist of a combination of letters and numbers, but not special characters such as. %, (, + and], etc., but underline, _, is ok. Numbers should not be first characters in a name. Examples:
  - 3 usable names: mass_bil_1, m_bil_1, M_bil_1
  - 2 illegal names: 1_car_mass,% _ rate
- Names of built-in functions in Python should not be used as names of variables because the built-in function then loses its original meaning. Example: If you, unfortunately, have given a variable the name "print", then the built-in function print() loses its original meaning. An unfortunate name selection can be reversed by deleting the variable with the del print command, and then, the print() function is again available as normal.

3.4 A little about functions

In *mathematics*, variables and functions (or formulas) are fundamental concepts. Just think of square root calculation:

$$y = \sqrt{x}$$

there

- x is the function's input or input argument,
- y is the output or output argument of the function
- $\sqrt{\ }$ (square root calculation) is the function.

Figure 3.4 illustrates this function.

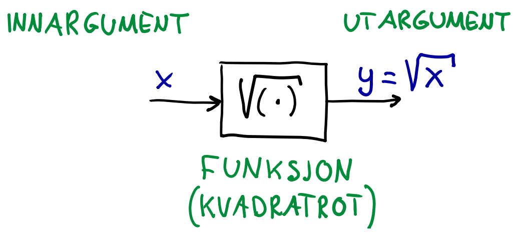


Figure 3.4: Mathematical function (square root calculation)

Similarly, it is in *programming*: Variable and Functions are fundamental concepts. This chapter focuses on variables and the various data types that variables can have (such as integers, floating numbers, text, etc.). However, since variables and functions often appear together, it is reasonable to introduce the concept of functions and some different forms of functions here. However, functions are treated in much more detail in Chapter 5.

A function "does something" with the input, or input argument, to the function. The result of what the function does is called the function output, or output argument or return argument. Figure3.5 illustrates the concept of functions in Python.

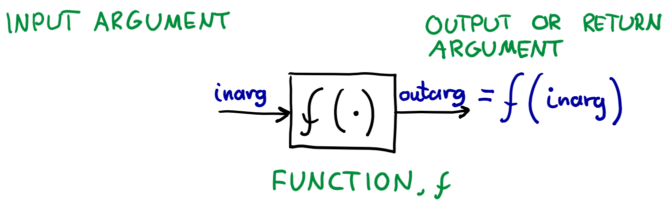


Figure 3.5: Function with input argument and output argument, or return argument

Functions come in different forms:

- Function
- Method

You will become more familiar with these forms through some simple examples.

Function

The built-in numpy package contains the sqrt() function for calculating square root. It is used as follows:

```
>>> import numpy as np
>>> x = 2.0
>>> y = np.sqrt(x)
>>> y
1.4142135623730951
```

The first line of code causes the numpy package to be imported into Python and renamed in that connection to np (which is a python tradition). Python also has some functions that come with its standard package. They can be used directly, without importing any package, cf. 2.6.2.

Method

Simply put, all variables in Python are so-called objects. And with the objects, there are a number of so-called methods, which are functions that "belong" to the object and operate on it. The syntax is

objekt.metode()

In the example below, we first create a variable named L of data type list, that consists of the two numbers 0 and 1 (we will learn more about lists in Ch. 3.9). Then we expand the list with an element of value 2 using the append() method which can be applied to list objects. We see that the append() method basically works as a function.

```
>>> L = [0, 1]
>>> L.append(5)
>>> L
[0, 1, 5]
```

Above I wrote L (+ enter) on the command line to display the value of L. I could also have used the print() function:

```
>>> L = [0, 1]
>>> L.append(5)
>>> print(L)
[0, 1, 5]
```

3.5 Numbers and basic mathematical operations

Here we will look at different ways of representing numbers and basic mathematical operations. More advanced calculations, and the special but very effective calculation method called vectorized calculation, are described in Ch. 3.12.7.

3.5.1 Numbers types

In Python we can use different types of numbers:

- ***Integer***.
  Example: 2
  Example: -10
- ***Floating point*** , or decimal number. In Python, dots, not commas, are used as decimal separators.
  Example: 1.23
- ***Complex numbers***. The symbol j is used as a complex unit.
  Example: 1 + 1j (just 1 + j will give an error message).

Numbers can be expressed with powers of 10 as follows:

- Example: 1.234e2 that is the number $1.234 \cdot 10^2 = 123.4$. The letter e stands for 10's exponent.
- Example: 1.234e$-$2 that is the number 0.01234.

3.5.2 How to format numbers in print() function

The print() function is probably the most frequently used function in Python. It is one of Python's built-in functions, cf. Figure 2.26. We have used this function many times already. When we use it to print numbers, the numbers are displayed by default with a large number of digits (14) after the decimal point. Often we want fewer digits, and we can obtain this by specifying the format (number) to print.

There are several ways to format the arguments of the print() function. The program (script) below demonstrates the most relevant ways. The program prints text, and numbers mixed with the text. Comments on the individual code lines and their results are given below.

Program name: prog_print_format.py.

```
x1 = x2 = x3 = x4 = x5 = x6 = x7 = 200/3
y2 = y4 = y6 = 100/3

print('x1 =', x1)
print('x2 =', x2, 'and y2 =', y2)
print('x3 =', f'{x3:.3f}')
print('x4 =', f'{x4:.3f}', ' and y4 =', f'{y4:.1f}')
print('x5 = %.3f' % x5)
print('x6 = %.3f and y6 = %.1f' % (x6, y6))
print('x7 =', f'{x7:.2e}')
```

Here is the result of running the above code lines as shown in the console:

```
x1 = 66.66666666666667
x2 = 66.66666666666667 and y2 = 33.333333333333336
x3 = 66.667
x4 = 66.667 and y4 = 33.3
x5 = 66.667
x6 = 66.667 and y6 = 33.3
x7 = 6.67e+01
```

Comments on the individual code lines and their results:

- print('x1 =', x1):
  No special formatting is used here, and x1 is displayed with a full 14 digits after decimal point!
- print('x2 =', x2, and y2 =', y2):
  Here, two numeric values are printed in the text – without any special formatting of the numbers. There will be lots of digits all in all. (Seems like there is a numerical inaccuracy in the last digit, by the way.)
- print('x3 =', f'{x3:.3f}'):
  Note the letter f in front of the formatting text string, {x3:.3f}. f stands for "formatted string literal" or "f-string".
  The term {x3: .3f} means that the value of x3 should be written as floating point with 3 digits after decimal point. Note the colon!
  I like f-string formatting, because the numbers are in their natural place within the text string to be printed.
- print('x4 =', f'{x4:.3f}', ' and y4 =', f'{y4:.1f}'):
  Same as in the above paragraph, but now two numbers are printed with each f-string formatting.
- print('x5 = %.3f' % x5):

Note where the% sign is.
This is a traditional way to format the print. On python.org[1] it is referred to as "old-string formatting", and it is stated that there may be certain technical problems with this formatting. Of course, it is used in many applications.
I think it is natural that we drop using this formatting method, but it is useful to know about it because it is probably used in existing code.

- print('x6 = %.3f and y6 = %.1f' % (x6, y6)):
  Same as the point above, but now two number values are printed. (x6, y6) is a tuple. The order of the elements in the tuple corresponds to the order of formatting in the text string to be printed.

- print('x7 =', f'{x7:.2e}'):
  This is f-string formatted printing, as for x3 above. The formatting character e, which stands for "exponential," indicates that the number is printed in 10s exponential form. The number 2 in front of e indicates that it must be 2 digits after the decimal point. Printing 6.67e+01 means $6.67 \cdot 10^1$.

I admit there are many details in the above, but I would say it's worth the effort since we – and others – use the print() function often.

I would like to repeat the request from python.org to Python programmers to use print with f-string formatting in the print() function, as for x3 and x4 and y4 above.

3.5.3 Mathematical operators

The basic mathematical operators are:

$+$ (addition)

$-$ (subtraction)

* (multiplication)

/ (division)

** (power), e.g. 2**0.5 ($= 2^{0,5} = \sqrt{2}$)

[1]https://docs.python.org/3/library/stdtypes.html#old-string-formatting

Example 3.1 *Basic mathematical operations*

The following commands written on the command line demonstrate basic mathematical operations (the results are also shown):

```
>>> 2 + 5.1
7.1
>>> 2 - 5.1
-3.1
>>> 1.2*1.5
1.7999999999999998
>>> 1.2/1.5
0.7999999999999999
>>> 2**0.5
1.4142135623730951
```

[End of Example 3.1]

The relative precedence of the operator

The operators have relative precedence or rank as follows:

1. **
   Example:
   ```
   >>> 2**3 + 4
   12
   >>> 2**(3+4)
   128
   ```
2. * and / (ie equal precedence, and the calculation is done from left to right).
   Example:
   ```
   >>> 8/4*2
   4.0
   ```
3. +and - (ie equal precedence, and the calculation is done from left to right).
   Example:
   ```
   >>> 3 + 2 −2
   3
   ```

Note: In a series of ** operators, the calculation is performed - from normal - from right to left.

With parentheses, you remove any doubts about the precedence of the operators. Recommended!

Example:

```
>>> 2**3**2
518
>>> 2**(3**2)
518
```

3.6 Text strings (strings)

In addition to numbers, Python can handle data in the form of text strings (pythonsk: strings), eg. 'Donald'. Alternatively, double quotes can be used: 'Donald' and “Donald” are thus equivalent. (Simple quotes are used in this book.)

Some things to note:

- Numbers are ok, e.g. '1' and '0.23'.
- Native character, like Norwegian characters, are ok, ie 'Æ', 'Ø', 'Å', 'æ', 'ø', 'å'.
- Special characters ok, e.g. "&" and "%".
- The character sequence \' is used to represent apostrophe as a character.
- The character sequence \n means new line.
- The + operator can be used to put together, or concatenate), text.

Example 3.2 *Text strings and operations on such*

The following examples use the print() function to display the text strings in the console.

```
>>> print('1'+'2')
12
```

```
>>> print('I don\'t know.')
I don't know.
```

```
>>> print('Line 1.\nLine 2.')
Line 1.
Line 2.
```

[End of Example 3.2]

3.7 From numbers to text and from text to numbers

A short rehearsal: Three basic data types in Python are

- float (floating point), e.g. 59.8[2]
- int (integer), e.g. 2
- str (text string), e.g. '59.8'

Python requires that you use the correct data type in current program expressions. It may therefore be necessary to convert mbetween some of the data types. Python has built-in functions that perform type conversion (pythonsk: datatype casting), cf. Figure 2.26:

- float() converts from str or int *to* float.
- int() converts from str or float *to* int with any rounding against null.[3]
- str() convert from float or int *to* str.

Example 3.3 *Conversion from float to str*

```
>>> vekt_float = 59.8 # Flyttall
>>> info = 'Hun veier ' + str(vekt_float) + ' kg.'
>>> info
Hun veier 59.8 kg.
```

The code above uses the + operator to merge text strings.

[2]

– I Python brukes desimalpunktum, ikke desimalkomma.

[3]numpy.ceil() rounds up to the nearest integer.

If we slurp by dropping the type conversion for the variable weight from float to size, Python gives clear message:

```
>>> vekt_float = 59.8
>>> info = 'Hun veier ' + vekt_float + ' kg.'
TypeError: can only concatenate str (not "float") to str
```

[End of Example 3.3]

Then we have an example of the opposite conversion:

Example 3.4 *Conversion from str to float*

The float() function is used here to convert text string to float number in preparation for a mathematical operation, namely multiplication:

```
>>> vekt_str = '59.8' # Tekststreng
>>> vekt_i_gram_float = float(vekt_str)*1000
>>> vekt_i_gram_float
59800.0
```

[End of Example 3.4]

Fortunately, the print() function is quite flexible

In the examples above, we had to convert between floating point and text string and vice versa. But it can be useful to keep in mind that the print() function itself is quite flexible since it can print (present in the console) a mix of text and numbers.

Example 3.5 *The print() function accepts a mix of different data types*

print() function prints both text and numbers:

```
>>> weight_float = 59.8 # Flyttall
>>> print('She would like to weigh', weight_float, 'kg.')
She would like to weigh 59.8 kg.
```

[End of Example 3.5]

Type conversion regarding the input() function

Suppose you have created a program where the floating point variable x is to be included in a calculation. Of course, you can assign x a value in the program code itself. Alternatively, the user can enter a value in the console *while the program is running* using the input() function, which is a built-in function in Python, see Figure 2.26. The input() function returns text. If x is to have data type floating point, we need to converte the type from text to number. This is demonstrated in the following example.

Example 3.6 *Type conversion ifm. input() - function*

The program in the box below uses the input() function to ask the user to enter a number on the command line in the console. The input() function considers this number as text. It is therefore necessary to convert the text to an actual number (data type float), and we do so with the float() function.

```
x = float(input(Enter the number, x: '))
y = x + 3.4
print('y =', y)
```

When the above program is run, the program pauses while the text 'Enter the number x:' appears on the command line. When the user (I) has entered a "number" (actual text) – and I entered “1.2”, the program proceeds with code succeding the input() code, which here is the addition y = x + 2.3, which gives the result 4.6. The box below shows the information that is presented in the console.

```
Enter the number x: 1.2
y = 4.6
```

If I had dropped the type conversion with the float() function, like this:

```
x = input('Enter the number x: ')
y = x + 3.4
print('y =', y)
```

then, Python would react negatively and give an error message:

```
⋮
    y = x + 3.4
TypeError: can only concatenate str (not "float") to str
```

[End of Example 3.6]

3.8 Boolean variables, logical operators and comparison operators

Programming is often about logical operations. This section introduces the basics about programming logical operations in Python.

3.8.1 Introduction

Suppose that you have created a program that contains two alternative program parts, program part 1 and program part 2. Suppose that program part 1 is to be executed if the floating point variable A is larger than the floating point variable B, otherwise program part 2 will be executed. in Figure 3.6. The program flow is thus determined by the comparison between A and B. More specifically: It is the result of the comparison

$$A > B$$

that determines the program flow. This comparison gives results either logically true or logically false, or in Python: True or False, as the two possible logical values in Python.

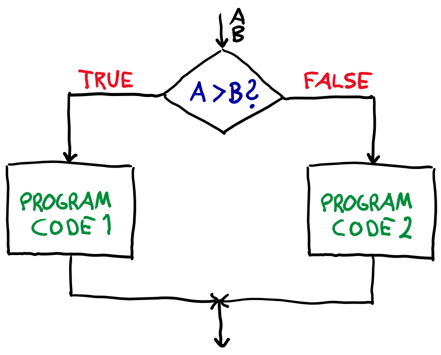


Figure 3.6: Program flowchart expressing that program flow is determined by the logical value of equation A > B.

The content of the diamond square, ie the test A> B?, can be realized with a so-called if expression. We will get closer to the if-expression in Section 7, but let us indulge in a little taste of how this can be expressed with Python code:

```
if A > B:
    Program code 1
else:
    Program code 2
```

Above we have become familiar with some terms that we will look at in the sections that follow:

- Logical values
- Comparisons

3.8.2 Boolean variable

Boolean or logical variables are variables that can have only one of two possible values:

- True
- False

Note: In contexts other than Python programming, e.g. in other programming languages, True and False may have other names:

- True can be named as "on"or "high" or "1" (logical one).
- False can be named as "off" or "low" or "0" (logical zero).

Boolean variables are useful for comparisons, etc.

3.8.3 Logical operators

In Python there are three logical operators which you can apply to Boolean variables:

- and
- or
- not

The significance of these logical operators can be represented in so-called truth tables , see the Tables 3.1 (for the and operator), 3.2 (or) og 3.3 (not).

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

Table 3.1: Truth table for the and operator

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

Table 3.2: Truth table for the or operator

A	not A
True	False
False	True

Table 3.3: Truth table for the not operator

Example 3.7 *Logical operators*

Code lines below demonstrates the logical operators and, or, not and how they can be combined into a composite logical operation.

Note that we can assign the value of a logical operation to a variable, which then gets a boolean value according to it according to the logical operation.

```
>>> A = True
>>> B = False
>>> C = A and B
>>> C
False
>>> A or B
True
>>> not A
False
>>> A and (not B) # Composite logical operation
True
```

[End of Example 3.7]

3.8.4 Comparison operators

Table 3.4 shows the most current comparison operators in Python.

Operator	Navn
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
==	Equal
!=	Unequal

Table 3.4: The most current comparison operators in Python

Note that the equality operator is ==. It is not =, which is the assignment operator.

Here is an example that demonstrates the comparison operators.

Example 3.8 *Logical operators*

```
>>> 1 < 2
True
>>> 1 <= 2
True
>>> 1 > 2
False
>>> 1 >= 2
False
>>> 1 == 2
False
>>> 1 != 2
True
```

[End of Example 3.8]

3.9 Lists

3.9.1 What are lists?

One of the data types in Python is the so-called list .

Here is an example of a list of 4 elements of floating point numbers (I have chosen to give the list name L1 here):

```
>>> L1 = [0.1, 2.3, 4.5, 6.7]
>>> L1
[0.1, 2.3, 4.5, 6.7]
```

Note:

- The elements are separated by commas (with or without spaces after the comma).
- The brackets are brackets (square brackets).

Each element has a so-called element index, which is an integer. The first element always has index 0 (not 1)[4]. The 4 elements in L1 then have the element indices 0, 1, 2 respectively. 3.

[4]In MATLAB, the first index of vectors or arrays, which in some ways are comparable with lists in Python, is 1

Figure 3.7 illustrates, based on the example above, the data type list.

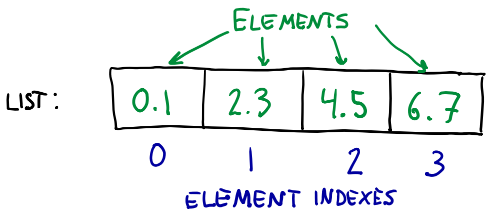


Figure 3.7: Illustration of a list. Note that the first element has index 0.

List elements can have data type text strings. An example:

```
>>> L2 = ['zero', 'one', 'two']
```

And here's a list of mix drops - numbers, text and list:

```
>>> L3 = [0.1, 'one', [1.0, 'two']]
```

How long is the list?

The built-in Python function len(), cf. Figure 2.26, find the length of a list. Example:

```
>>> L1 = [0.1, 2.3, 4.5, 6.7]
>>> n = len(L1)
>>> n
4
```

Sequences

Lists are sequences of data. There are other data types than lists that consist of sequences of data, namely arrays and tuples, which are mentioned in their respective sections. All of these data types fall under the general Python terms sequences and iterables.

3.9.2 Operations on lists

3.9.2.1 Reading list elements

Reading one list element

Let's assume we have a list named L (but we could of course have assumed another name). We can then assign a variable, here named E, the value of the list element with index i with the code

E = L[i]

As mentioned earlier, the first element has index 0. But what is the index of the last element? It is -1, which will apply to all lists. It sounds strange, but Figure3.8 shows logic: We consider the list as a closed sequence of elements! For a specific list of n elements, the last element index can alternatively be specified as $n - 1$.

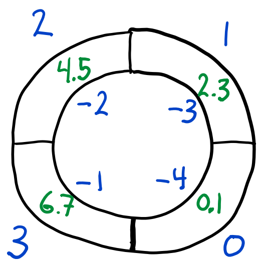


Figure 3.8: A list considered as a closed sequence of elements.

Example 3.9 *Read one element in a list*

Below are some examples of reading items in list L = [0.1, 2.3, 4.5, 6.7], as we defined above and illustrated in Figure 3.7.

```
>>> L = [0.1, 2.3, 4.5, 6.7]
>>> L[0]
0.1
>>> L[2]
4.5
>>> L[3]
6.7
>>> L[-1]
6.7
```

[End of Example 3.9]

Read a series of list items

To read item values in a list, we must access (or index or address) the relevant elements by specifying their indexes. We use the operator: (colon) to access a series of list items (python for serial accessing: slicing). The term L [i: j] accesses the elements starting with the element i to – *but not with* – element j. This is illustrated in the figure 3.9. In particular, we note that element j in list L is not included in list extract L2, and therefore this element is marked in gray. (The white elements are not included in the list excerpt either.)

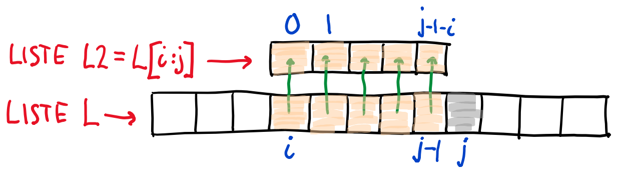


Figure 3.9: List extract L2 = L [i: j]. We especially notice that element j in list L is not included in the list excerpt.

If the start index is 0, we can drop typing 0. For example, L [0: 2] is equivalent to L [: 2].

Example 3.10 *Read a series of items in a list*

We are going to read the series of elements from index 0 to (but not with) index 2 in list L as we defined above:

```
>>> L = [0.1, 2.3, 4.5, 6.7]
>>> L[0:2]
[0.1, 2.3]
>>> L[:2] # Ekvivalent med L[0:2]
[0.1, 2.3]
```

[End of Example 3.10]

If we are going to access the items from index i and "out list", ie even the last item, we can write L [i:].

Example 3.11 *Read a series of items through the last item in the list*

Here we read the items from index 1 and out list:

```
>>> L = [0.1, 2.3, 4.5, 6.7]
>>> L[1:]
[2.3, 4.5, 6.7]
```

[End of Example 3.11]

3.9.2.2 How to update list items with new values

The examples below demonstrate how we can update list items in an existing list of new values. In this connection, the list elements are accessed in the same way as when we read from a list, cf. 3.9.2.1.

Oppdatere ett element

Example 3.12 *Update one item in a list*

Below, item 2 of the original list L = [0.1, 2.3, 4.5, 6.7] is updated with the new value -4.5 as follows:

```
>>> L = [0.1, 2.3, 4.5, 6.7]
>>> L[2] = -4.5
>>> L
[0.1, 2.3, -4.5, 6.7]
```

[End of Example 3.12]

Update a series of list items

When updating a series of list items from element i to - but not with - element j, we access the relevant part of the list (to be updated) with the code L [i: j].

Example 3.13 *Update a series of items in a list*

Returns list L = [0.1, 2.3, 4.5, 6.7]. We can update the series of elements 0, 1 and 2 with the values 0.5, 2.5 and respectively. 3.5 as shown in the box below. Note: We access the current list items with the code L [0: 3], not L [0: 2].

```
>>> L = [0.1, 2.3, 4.5, 6.7]
>>> L[0:3] = [0.5, 2.5, 3.5]
>>> L
[0.5, 2.5, 3.5, 6.7]
```

[End of Example 3.13]

3.9.2.3 Expand lists with new items

We can extend an existing list with one or more new elements with the extend() function or method for lists. Pythonically, extend() is a method that belongs to lists considered as objects. (The concepts of object and method are explained in Chap. 3.4.)

The extend() method syntax is

L_original.extend(L_extension)

where L_original is the original list and L_extension is the list with which L_original is expanded.

Example 3.14 *List extension*

Returns list L = [0.1, 2.3, 4.5, 6.7]. We can expand the list with a new element with value 8.9 as follows:

```
>>> L = [0.1, 2.3, 4.5, 6.7]
>>> L.extend([8.9])
>>> L
[0.5, 2.5, 3.5, 6.7, 8.9]
```

Comment on the code above:

1. Python gives the error message " float 'object is not iterable' if we drop the bracket around 8.9. This is because the extend() method must be extended with a list. 8.9 is in this context is not a list - therefore the error message. [8.9] is a list - therefore no error message.

We can also use the extend() method to extend by more than one element, as long as these elements are in the form of a list. Here we extend the original list with list [8.9, 10.0]:

```
>>> L = [0.1, 2.3, 4.5, 6.7]
>>> L.extend([8.9, 10.0])
>>> L
[0.5, 2.5, 3.5, 6.7, 8.9, 10.0]
```

[End of Example 3.14]

What about the append() method?

Above we have used the extend() method. Alternatively, where we extended L by one element, we could have used append() as follows: L.append (8.9) - without parentheses, with the same result as with L.extend ([8.9]).

Note: When expanding by more than one element, the append() method produces a result that can be unexpected and possibly problematic. In the example above, let's use append() instead of extend():

```
>>> L = [0.1, 2.3, 4.5, 6.7]
>>> L.append([8.9, 10.0])
>>> L
[0.1, 2.3, 4.5, 6.7, [8.9, 10.0]]
>>> len(L)
5
```

The result is thus a list (L) consisting of 4 floating numbers and a list of two floating numbers. The number of items in L becomes 5, not 6 as expected! Here there can be misunderstandings and possible mistakes. If we had used extend(), the result would be a list of 6 floating numbers and with length 6, as expected. Based on this, I would say that extend() is much better than append() for list extension with more than one element.

How about one item list extension? We have already seen that it is hip to happ if we do list extension with append() or extend() - the resulting list will be the same. But maybe one of them is much faster to drive? I have created a program where I have detected the time[5] it takes to expand a list of one item with each of the methods. The test shows that append() is about 10% faster than extend(). That difference is not much to brag about!

Based on this, I would recommend the extend() method over the append() method when listing extensions with both one and more elements.

[5]The program uses the time.time() method to calculate the time it takes to make one million extensions. These repeated expansions take place in a so-called loop, which is the theme elsewhere in the book.

3.9.2.4 Remove list items

We can remove items from a list with the command del (abbreviation for delete). We access the elements to be removed in the same way as when reading list items, cf. 3.9.2.1. The syntax for removing an indexed item from list L is

del L[i]

Syntax to remove items from index i to (but not with) index j from list L is

del L[i:j]

Example 3.15 *Remove items from a list*

Returns list L = [0.1, 2.3, 4.5, 6.7]. Let's remove item 2 from the list:

```
>>> L = [0.1, 2.3, 4.5, 6.7]
>>> del L[2]
>>> L
[0.1, 2.3, 6.7]
```

Let's start again with list L = [0.1, 2.3, 4.5, 6.7] and remove the two elements with indexes 1 and 1 respectively. 2 from the list:

```
>>> L = [0.1, 2.3, 4.5, 6.7]
>>> del L[1:3]
>>> L
[0.1, 6.7]
```

[End of Example 3.15]

3.9.2.5 List manipulation with + and *

Usually the + and * operators are used for mathematical operations. But when these operators are used on lists, they do not work mathematically. Instead, they are used to manipulate the list. So here we are in a minefield: -o

If you want to use the + operator to add a number value to all the list items and the * operator to multiply all the list items by a number value, you must convert the list to array, and then use the operators on the array. This is explained in Chap. 3.12.

The + operator used between two lists is used to merge (python: concatenate) listae, as demonstrated here:

```
>>> L = [10, 20, 30]
>>> L1 = [40, 50, 60]
>>> L + L1
[10, 20, 30, 40, 50, 60]
```

The * operator applied to a list is used to create a new list consisting of a sequence of the original list, e.g. 3 times the original list, as in the following example:

```
>>> L = [10, 20, 30]
>>> L*3
[10, 20, 30, 10, 20, 30, 10, 20, 30]
```

3.10 Tuples

Tuples are used to group data elements of the same or different data types. Tuples are very similar to lists. Two important differences are:

- The elements in tuples are gathered in regular parentheses, ie(), while lists are gathered in square brackets, [].
- Tuples cannot be manipulated, ie they can be considered static lists.

When using tuples, they are often the starting arguments of functions you create yourself:

(ut_arg_1, ut_arg_2) = en_eller_annen_funksjon(inn_arg)

where (ut_arg_1, ut_arg_2) is a tuple with two elements (Programming your own functions is covered in Ch. 5.)

How to define tuples

Here is an example of defining a tuple, which consists of the floating point number 1.0 and the text string ’a’:

```
>>> T = (1.0, 'a')
>>> T
(1.0, 'a')
>>> type(T)
tuple
```

It is not really necessary to use parentheses when defining tuples, but Python nevertheless displays parentheses around. (I prefer to use parentheses because I think the code then appears clearer.)

```
>>> T = 1.0, 'a'
>>> T
(1.0, 'a')
```

As I said, you cannot change the value of one or more tuple elements (as you can with lists), but you can (of course) give a tuple a new value and in that way change its content. You can also extend a tuples with the + operator:

```
>>> T1 = (1.0, 'a')
>>> T2 = (2.0, 'b')
>>> T = T1 + T2
>>> T
(1.0, 'a', 2.0, 'b')
```

How to unpack tuples

Reading the values of one or more tuple elements is called unpacking the tuple. You can

- unpack the entire tuple in one operation (and you can choose to include or not include the parenthesis() around the result of the unpacking)
- access the individual elements with indexing, just as you do with lists.

This is demonstrated in the example below.

```
>>> T = (1.0, 'a')
>>> (num1, text1) = (1.0, 'a') # Unpacking the whole tuple using paren-
theses
>>> num1
1.0
>>> text1
'a'
>>> num1, text1 = (1.0, 'a') # Unpacking the whole tuple without using
parentheses
>>> num1
1.0
>>> text1
'a'
>>> text1 = T[1] # Unpacking of one of the elements
>>> text1
'a'
```

3.11 Dictionary

A dictionary is a collection of data elements that can have different data types, and the data elements are indexed (accessed) with keys in the form of text strings. A nice feature of dictionaries is that the keys (indexes) can be made "readable" (such as "teams"'). Lists, cf. 3.9, also consists of a collection of data elements, but in lists the indices are just integers.

Below you will see how to create a dictionary. Comments on the code:

- 'Team' is key, and 'ManU' is its value .
- 'Points' is also key, and 100 is its value.

```
>>> D = {} # Creates an empty dictionary
>>> D['Team'] = 'ManU' # Adds an element to the dictionary
>>> D['Points'] = 100 # Adds another element
>>> D
{'Team': 'ManU', 'Points': 100} # In their dreams
```

You can read the value of a key as shown below:

```
>>> D['Team']
'ManU'
>>> D['Points']
100
```

You can remove an element with the del command:

```
>>> del D['Points']
>>> D
{'Team': 'ManU'}
```

3.12 Arrays

3.12.1 Introduction

Arrays are very similar to lists. Both consist of sequences of data, which can be numbers or text. The big difference between them is that a sea of mathematical functions is open to arrays of numbers, while there are hardly any mathematical functions for lists. *This means that you have to represent the data in arrays if you are going to do calculations with the data.*

However, the starting point for an array may well be a list, in which case you need to convert the list to an array. List-to-array conversion, and vice versa, is described in Section 3.12.2.

Arrays is a data type that "belongs" to the numpy package. Therefore, to create and operate on arrays, you must import the numpy library. It is common python to rename numpy to np when importing:

```
import numpy as np
```

You perform the import either from the command line or by running a script that contains this command.

In the upcoming sections, we will look at some common ways to create arrays and some common operations and calculations on arrays.

Complete documentation on arrays is available at[6]

https://docs.scipy.org/doc/numpy/reference/arrayer.html

The code examples in this subchapter are

[6]Ihht. scipy.org is Scipy "... a collection of open source software for scientific computing in Python".

import numpy as np

as the first command. You do not need to execute this command in each instance if you are constantly in the programming environment (such as spears), but if you exit and re-enter, you must execute the command.

3.12.2 How to convert lists to arrays and vice versa

3.12.2.1 Conversion from list to array

How do we convert a numeric list to an array? We can use the array() function in the numpy package. Suppose the list has name L and the array should have name A. We can then convert L to A with:

A = np.array(L)

Example 3.16 *Conversion of a list to an array*

The following code converts the list L = [0.1, 2.3, 4.5, 6.7] to an array here called A:

```
>>> import numpy as np
>>> L = [0.1, 2.3, 4.5, 6.7]
>>> A = np.array(L)
>>> A
array([0.1, 2.3, 4.5, 6.7])
```

[End of Example 3.16]

Note: Even if we say that we convert a list to an array, the original list will still exist after the conversion.

3.12.2.2 Conversion from array to list

If you need to convert an array, called A, to a list, called L, you can use Python's built-in list() function:

L = list(A)

Example 3.17 *Conversion of list to array*

The following code converts the list L = [0.1, 2.3, 4.5, 6.7] to array A, which is then converted to list L1:

```
>>> import numpy as np
>>> L = [0.1, 2.3, 4.5, 6.7]
>>> A = np.array(L)
>>> L2 = list(A)
>>> L2
[0.1, 2.3, 4.5, 6.7]
```

[End of Example 3.17]

3.12.3 Create arrays of special design

3.12.3.1 Arrays with equal element values

A weird array

An empty array:

```
>>> import numpy as np
>>> A = np.array([])
>>> A
array([], dtype=float64)
```

dtype = float64 expresses the data type, which is here set to float64 (floating point with represented 64 bits).

You can expand an empty array of new elements. However, it's probably rare that you need an empty array.

Array of only zeros

We can create an array of n elements that all have a value of 0, using the zeros() function in numpy:

$$A = \text{np.zeros}([n])$$

Example:

```
>>> import numpy as np
>>> A = np.zeros([3])
>>> A
array([0., 0., 0.])
```

Array of only ones

We can create an array of n items that all have value 1, using the the ones() function:

A = np.ones([n])

Example:

```
>>> import numpy as np
>>> A = np.ones([3])
>>> A
array([1., 1., 1.])
```

Array with all elements having the same value

We can create an array of n elements that all have value k (whatever you set it to) by starting with an array of only 0's, which we then add k to:

A = np.zeros([n]) + k

The result is that k is added to each of the elements. (It may seem strange that one can add one number, that is, a scalar, to an array of numbers, but it is thus allowed.)

Alternatively, we can use the ones() function, which gives the same result:

A = k*np.ones([n])

Example:

```
>>> import numpy as np
>>> A = np.zeros([3]) + 5.2
>>> A
array([5.2, 5.2, 5.2])
```

Note that here the array differs from lists, since the + operator applied to a list means to *expand* list, not elemental addition as for arrays. Check the result of adding an element, here 5.2, to a list:

```
>>> import numpy as np
>>> L = list(np.zeros([3])) + [5.2]
>>> L
[0.0, 0.0, 0.0, 5.2]
```

3.12.3.2 Array with fixed increment between elements

It is often necessary to create an array with fixed increment (distance) between the element values, e.g. an array of times with fixed time steps between times. The linspace() function in the numpy package can be done as follows:

A = np.linspace(start, stopp, antall)

there

- start is a given start value, like *are included* in the array.
- stop is a given stop value, like *are included* in the array.
- number is the number of elements in the array.
- A is the resulting – or returned – array. (Of course, you can choose a name other than A on the array.)

Example 3.18 *Using np.linspace() to create an array of points of time*

The code below creates an array of points of time, from (included) the start time t_start = 0 s to (included) the end time t_stop = 1.0 s, with time step Ts = 0.1 s between each point of time.

```
>>> import numpy as np
>>> t_start = 0.0
>>> t_stop = 1.0
>>> Ts = 0.1
>>> n = int((t_stop - t_start)/Ts) + 1 # int() rounds downwards.
>>> t = np.linspace(t_start,t_stop,n)
>>> t
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
>>> n
11
```

[End of Example 3.18]

np.arange() instead of np.linspace()?

As an alternative to np.linspace (start, stop, number), we have np.arange (start, stop, step). One difference between these functions is that they use, respectively, number and steps as arguments to specify the distance (step size) between the element values. Another difference – and that is the most important difference – is that linspace includes the stop value in the array, while arange() *does not include* the stop value.[7]

Example 3.19 *Using arange() to create an array of points of time*

This example is pretty much the example 3.18 based on linspace(), but let's use arange() instead:

```
>>> import numpy as np
>>> t_start = 0.0
>>> t_stopp = 1.0
>>> Ts = 0.1
>>> t = np.arange(t_start,t_stopp,Ts)
>>> t
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
>>> n
10
```

The number of elements becomes (of course) one less for arange() than for linspace(), ie 10 and 11 respectively, given that the time step is the same (set to Ts = 0.1 s in both cases).

[End of Example 3.19]

3.12.3.3 Multidimensional or n-dimensional arrays

So far we have used one-dimensional arrays. In general, Python/numpy supports multidimensional arrays, or n-dimensional arrays where n is an integer: 1 or 2 or 3, etc. More specifically:

- One-dimensional arrays, or 1D arrays, have the elements (whatever their value) located along one dimension or axis. Such arrays are also

[7] I prefer linspace() over arange().

called vectors. Visually, 1D arrays are similar to horizontal or vertical lines (you can choose to consider 1D arrays as horizontal or vertical lines).

- Two-dimensional arrays, or 2D arrays, have the elements located along two dimensions or axes. Such arrays are also called matrix. Visually, 2D arrays are similar to surfaces.
- Three-dimensional arrays, or 3D arrays, have elements placed along three dimensions or axes. Such arrays are also called tensors. Visually, 3D arrays resembles cubes.
- Etc.

Figure 3.10 illustrates 1D, 2D and 3D arrays. The Figure shows examples of addressing specific elements (but I have not specified any the value of the elements (numbers or text) in these examples).

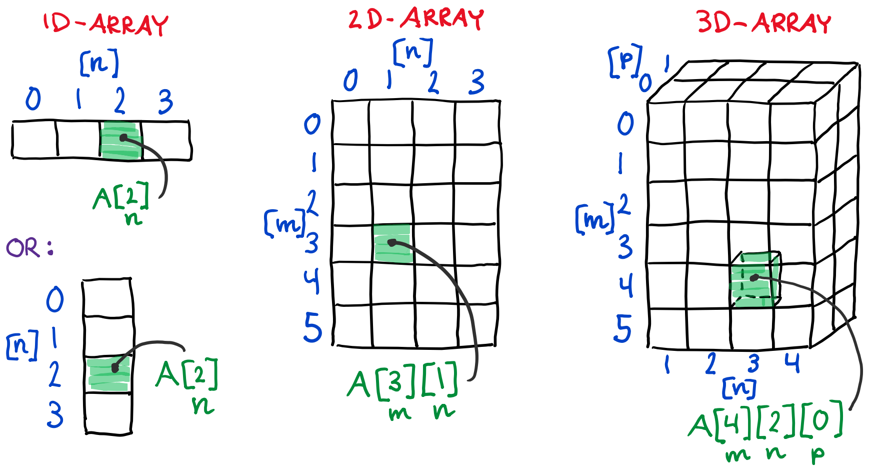


Figure 3.10: 1D, 2D and 3D arrays

Here is an example where we create a matrix (2D array).

Example 3.20 *Matrix (2D array)*

We create a 2D array with 2 rows and 3 columns:

```
>>> import numpy as np
>>> A = np.array([[0, 10, 20], [30, 40, 50]])
>>> A
array([[ 0, 10, 20],
[30, 40, 50]])
```

[End of Example 3.20]

We can generally create n-dimensional arrays, e.g. 19-dimensional arrays, but we will not look into this.

3.12.4 Array operations

3.12.4.1 Introduction

In Chap. 3.9.2 we learned about various operations on lists, namely

- Find the length
- Read element values
- Update element with new values
- Expand with new elements
- Remove element
- Find maximum values and minimum values

The same operations can be performed on arrays and with the same syntax as for lists. This is shown in the sections below.

3.12.4.2 The size of an array

Example 3.21 *The length of an array*

The length or number of elements in an array can be found with Python's built-in len() function:

```
>>> import numpy as np
>>> A = np.array([0.1, 2.3, 4.5, 6.7])
>>> n = len(A)
>>> n
4
```

[End of Example 3.21]

What about multidimensional arrays? It is not easy to say what the length of such arrays is. Although the len() function produces a result, we should use the shape attribute of the array object instead of len(). The shape attribute gives us the dimension of the array, ie the number of rows, columns, etc.

Example 3.22 *The dimension of a 2D array, i.e., an array*

The code below finds the dimension of a 2D array, which is an array. Note that the shape() method specifies the dimension in the form of a tuple, here (m, n)!

```
>>> import numpy as np
>>> A = np.array([[1, 2], [3, 4], [5, 6]])
>>> A
array([[1, 2],
[3, 4],
[5, 6]])
>>> (m, n) = A.shape
>>> m
3
>>> n
2
```

[End of Example3.22]

3.12.4.3 Read element values in an array

We can read element values in an array by accessing the elements just as for lists, and if possible assign the read element values to variables, cf.3.9.2.1.

Example 3.23 *Read element values in an array*

```
>>> import numpy as np
>>> A = np.array([0.1, 2.3, 4.5, 6.7])
>>> x = A[1]
>>> x
2.3
```

[End of Example 3.23]

Read a series of elements

Now we will read the series of elements fom. index 0 to (but not including) index 2 in array A as we defined in example 3.23:

Example 3.24 *Read a series of element values in an array*

```
>>> import numpy as np
>>> A = np.array([0.1, 2.3, 4.5, 6.7])
>>> A[0:2]
array([0.1, 2.3])
```

[End of Example 3.24]

Read element values in a 2D array (array)

Example 3.25 *Read element values in a 2D array (array)*

We create a 2D array with 2 rows and 3 columns, which is the same array as in the example 3.20:

```
>>> import numpy as np
>>> A = np.array([[0, 10, 20], [30, 40, 50]])
>>> A
array([[ 0, 10, 20],
[30, 40, 50]])
```

Then we can read e.g. the element in row 1 and column 2 in the array - remember that the lowest index for both rows and columns is 0, and note the addressing method with square brackets in series:

```
>>> x = A[1][2]
>>> x
50
```

[End of Example 3.25]

3.12.4.4 Update elements in an array

Update one element

Example 3.26 *Update one element in a list*

Below, element of index 2 of the original list L = [0.1, 2.3, 4.5, 6.7] is updated with the new value -4.5 as follows:

```
>>> import numpy as np
>>> A = np.array([0.1, 2.3, 4.5, 6.7])
>>> A[2] = -4.5
>>> A
array([ 0.1, 2.3, -4.5, 6.7])
```

[End of Example 3.26]

Update a series of list elements

To update a series of array elements from element i to – but not included – element number j, we access the relevant part of the array (to be updated) with the code A [i: j].

Example 3.27 *Update a series of elements in an array*

Here we update the series of elements 0, 1 and 2 with the values 0.5, 2.5 and respectively. 3.5. Note: We access the current list elements with the code A [0: 3], not A [0: 2].

```
>>> import numpy as np
>>> A = np.array([0.1, 2.3, 4.5, 6.7])
>>> A[0:3] = [0.5, 2.5, 3.5]
>>> A
array([0.5, 2.5, 3.5, 6.7])
```

[End of Example 3.27]

3.12.4.5 Expand arrays with new elements

Remember how we can expand lists of new elements at the end of the list? With the extend() or append() method of the list object. There are no such methods for array objects! Instead, we can use the np.append() function, as demonstrated in the following example.

Example 3.28 *Update a series of elements in one array*

```
>>> import numpy as np
>>> A = np.array([0.1, 2.3, 4.5, 6.7])
>>> A = np.append(A,[8.9, 10.0])
>>> A
array([ 0.1, 2.3, 4.5, 6.7, 8.9, 10. ])
>>> len(A)
6
```

Note: With the code only

np.append (A, [8.9, 10.0])

A does not expand. It is necessary to assign A the value of np.append (A, [8.9, 10.0]), as done in the code above.

[End of Example3.28]

You can add new elements anywhere in an array with the np.insert() function, but we will not look into it here.

3.12.4.6 Remove elements from arrays

We remove elements from lists with the del command, cf. 3.9.2.4. It does not work on arrays! Instead, we can use the np.delete() function, as demonstrated in the following example.

Example 3.29 *Remove elements from an array*

Below, we remove elements 1 and 2 from array A, and assign A the value of the updated (reduced) array:

```
>>> import numpy as np
>>> A = np.array([0.1, 2.3, 4.5, 6.7])
>>> A = np.delete(A,[1,2])
>>> A
array([0.1, 6.7])
```

[End of Example 3.29]

3.12.4.7 Find the maximum and minimum in arrays

We can find the index of the element that has the maximum value in an array with the np.argmax() function. The same applies to the minimum value; just replace argmax() with argmin(). Once this index is found, the maximum element value can be found with regular addressing.

Example 3.30 *How to find maximum value in an array*

The following code finds the index of the maximum value element in the list, as well as the value of this element:

```
>>> import numpy as np
>>> A = np.array([0.1, 2.3, 4.5, 6.7])
>>> i_max = np.argmax(A)
>>> i_max
3
>>> A[i_max]
6.7
```

[End of Example 3.30]

3.12.5 Mathematical operations on arrays, including matrices

3.12.5.1 Scalar addition and scalar multiplication

In many contexts we need

- to add a scalar to each array element
- to multiply each array element by a scalar

This can be done very easily in Python with, respectively, the + and * operator (below, A is the array and s is the scalar):

$$\text{A + s}$$

and

A*s

You may say that the syntax does not make sense, because we cannot really add an array and a number, but you just have to bow our thanks.

Example 3.31 *Addition of scalars to each element of an array*

Here, we add 1 to an array:

```
>>> import numpy as np
>>> A = np.array([10, 20, 30])
>>> A + 1
array([11, 21, 31])
```

[End of Example 3.31]

Example 3.32 *Multiplying each array element by a scalar*

Here, we multiply by 3:

```
>>> import numpy as np
>>> A = np.array([10, 20, 30])
>>> A*3
array([30, 60, 90])
```

[End of Example 3.32]

3.12.5.2 How to create row vectors and column vectors and arrays

In mathematics (linear algebra) we distinguish between row vectors and column vectors. You can create row vectors and column vectors in Python for 2D arrays only. In a sense, 1D arrays are neither row vector nor column vector – just vector. You can also create matrices from given row vectors and/or column vectors. We need to take a closer look at this.

How to make a row vector

Row vector (note that there are two square brackets in code line 2):

```
>>> import numpy as np
>>> R = np.array([[1.1, 2.2, 3.3]])
>>> R
array([[1.1, 2.2, 3.3]])
>>> R.shape
(1, 3)
```

If you *drops* one of the square brackets, you get an array, but you can't tell if the array is a row vector or column vector, see the following example:

```
>>> import numpy as np
>>> R = np.array([1.1, 2.2, 3.3])
>>> R
array([1.1, 2.2, 3.3])
>>> R.shape
(3,)
```

The fact that R.shape gives (3,) means that R is only a vector with 3 elements, and cannot be classified as neither a row vector nor a column vector.

How to create a column vector

We can create column vectors as follows:

```
>>> import numpy as np
>>> K = np.array([[1.1], [2.2], [3.3]])
>>> K
array([[1.1],
[2.2],
[3.3]])
>>> K.shape
(3, 1)
```

Alternatively, we can create a column vector a the transpose of a row vector. To transpose is to reflect on an imaginary axis (diagonal), see Figure 3.11. The figure shows, for example, the transposition of an array in the form of a (2, 6) array, but the principle is the same regardless of the number of rows and columns.

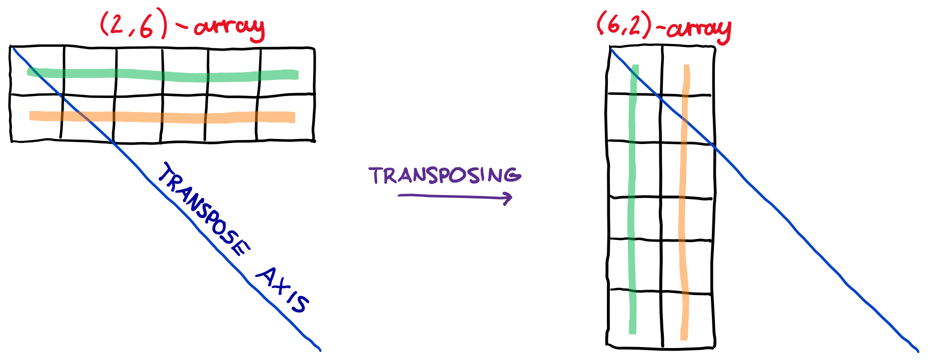


Figure 3.11: Matrix transposition (2D Array)

Transposing can be achieved with the T method of the array object, which is here the row vector R:

```
>>> import numpy as np
>>> R = np.array([[1.1, 2.2, 3.3]])
>>> K = R.T
>>> K
array([[1.1],
[2.2],
[3.3]])
>>> K.shape
(3, 1)
```

So, we got exactly the same K-vector as earlier.

Note: Trying to transpose a 1D array in Python is useless, as you only get back the original 1D array. Transposing only works in higher dimensional arrays.

How to create a matrix with given 1D arrays as column vectors

Suppose you create an array where the columns are to be formed from existing 1D arrays. Eg. we can have two arrays like this:

- A 1D array named t that contains times for a series of measurements.
- A 1D array named m containing the measurement values of a particular sensor (measurement element).

All in all we have two 1D arrays, which we would like to collect in a data array in the form of a 2D array named say d, where the two columns originate. When we create such a matrix, we get a challenge because the 1D arrays cannot be considered as neither row vectors nor column vectors – they are just vectors. The challenge can be solved by first creating a matrix where the 1D arrays become row vectors, and then creating the final matrix as the one transposed by this matrix. This is demonstrated by example 3.33 below.

Example 3.33 *An array of column vectors from given 1D arrays*

Below are the t and m 1D arrays. The matrix d is made as the transpose of the 2D array that has t and m as rows.

The 2D array d1 is generated as an intermediate result for the sake of illustration. Of course, you can drop generating d1, and type d = np.array ([t, m]).T directly.

```
>>> import numpy as np
>>> t = np.array([0, 1, 2, 3, 4])
>>> m = np.array([30.0, 30.1, 30.2, 30.3, 30.4])
>>> d1 = np.array([t, m])
>>> d1
array([[ 0. , 1. , 2. , 3. , 4. ],
       [30. , 30.1, 30.2, 30.3, 30.4]])
>>> d1.shape
(2, 5)
>>> d
array([[ 0. , 30. ],
       [ 1. , 30.1],
       [ 2. , 30.2],
       [ 3. , 30.3],
       [ 4. , 30.4]])
>>> d.shape
(5, 2)
```

[End of Example 3.33]

3.12.5.3 Vector and matrix multiplications

In linear algebra we encounter both addition and multiplication of vectors and matrices always. Python actually has the ability to transform arrays into arrays – or rather: array objects – with the np.mat() function, which works like this:

M = np.mat(A)

so that one can use the syntax for addition and multiplication with + and * on the matrix object (just like in the known MATLAB calculation tool).

But now the Python developers have decided that the matrix object will soon be out of life. Therefore, it is just as easy to forget matrix objects:

~~M = np.mat(A)~~

Therefore:

We should use array objects in vector and matrix multiplications because matrix objects have an uncertain future!

How do we then add and multiply vectors and matrices? There are several ways to do this, but I think the neatest thing is to make sure that the vectors and matrices are 2D arrays, and then perform the mathematical operation on these 2D arrays:

Addition:

A1 = A1 + A2

That is, the syntax is as for matrix objects.

But for multiplication, there are news:

A3 = A1 @ A2

So instead of *, we use @ (the at character).

Here is an important rarity when it comes to multiplication: To multiply a matrix in the form of a 2D array with a *scalar*, then the @ operator unfortunately does not work; then you must use the * operator just like in Section 3.12.5.1:

A * s

About the use of words: As a result of what has been said above, I hereby stop talking about matrices in the form of *matrix objects* in Python. But I will continue to talk about matrices, assuming they are 2D arrays in Python.

Multiplication of vectors

From mathematics we know that the scalar product (also called the inner product or the dot product) between two vectors a and b where both are assumed to be row vectors, are[8]

$$p = a \cdot b^T$$

In Python we can calculate this with the @ operator for multiplication:

p = a @ b.T

Example 3.34 *Scalar product*

Here we create two row vectors, then calculate their scalar product, which actually becomes in the form of a 2D array. We should have liked the result as a floating point, and we get that with Python's built-in float() function:

```
>>> import numpy as np
>>> a = np.array([[0, 1, 2]])
>>> b = np.array([[3, 4, 5]])
>>> p = a @ b.T
>>> p
array([[14]])
>>> f = float(p)
>>> f
14.0
```

Python can actually calculate the scalar product with a simpler syntax, see below. Maybe you like this syntax better than the “rigid” syntax above.

```
>>> import numpy as np
>>> a = np.array([0, 1, 2])
>>> b = np.array([3, 4, 5])
>>> p = a @ b
>>> p
14.0
```

[8]If a and b are one and the same vector, i.e. a = b, the scalar product is the square of the length of the vector.

Question: Can we use the "natural" product operator * to calculate the scalar product, ie, can we write p = a * b? Just try it![9]

Finally, I mention that numpy offers the vdot() function to calculate the scalar product (dot product) of vectors:

```
>>> import numpy as np
>>> a = np.array([0, 1, 2])
>>> b = np.array([3, 4, 5])
>>> p = np.vdot(a, b)
>>> p
14
```

[End of Example 3.34]

Multiplication of matrices and vectors

Matrices and vectors are naturally multiplied by the @ operator, as demonstrated in the following example.

Example 3.35 *Multiplication of matrix and vector*

Given the matrix

$$M = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$$

and the vector

$$v = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

The product of M and v is

$$p = Mv = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}$$

The following code calculates this product:

```
>>> import numpy as np
>>> M = np.array([[1, 0], [0, 2]])
>>> v = np.array([[3], [4]]) # Column vector
>>> p = M @ v
>>> p
array([[3],
       [8]])
```

[9]The answer is no.

Want to see if p = M * v gives the same result?

[End of Example 3.35]

Multiplication of Matrices

Matrices are naturally multiplied by the @-operator, just as when multiplying matrices and vectors. I skip an example on this.

3.12.6 Matrix functions for linear algebra

The numpy package with its linealg module has a large collection of matrix functions for linear algebra. The examples below illustrate its use.

Example 3.36 *A couple of examples of matrix functions in linear algebra*

Here, the determinant of a matrix is calculated:

```
>>> import numpy as np
>>> M = np.array([[1, 0], [0, 2]])
>>> det_M = np.linalg.det(M)
>>> det_M
2.0
```

Here, an array is inverted:

```
>>> import numpy as np
>>> M = np.array([[1, 0], [0, 2]])
>>> inv_M = np.linalg.inv(M)
>>> inv_M
array([[1. , 0. ],
       [0. , 0.5]])
```

[End of Example 3.36]

3.12.7 Vectorized calculations

Suppose you use a given function in Python on each of the n elements of an array, which we call vector. Initially, you must then apply the function once to each of the n elements, ie n times. It can be cumbersome. Fortunately, Python developers have thought of this. They have facilitated so-called vectorization, which implies that *vector* – and not each of the elements – is specified as the function's input argument. Thus, it keeps entering the function once in the program. Figure 3.12 illustrates vectorization vs. non-vectorization. In the example, the sqrt() function is the function of the numpy package.

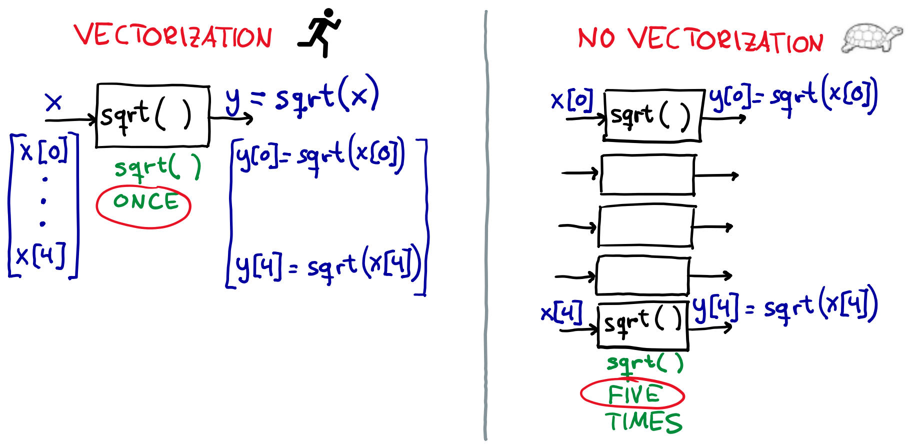


Figure 3.12: Illustration of vectorization vs. non-vectorization. The sqrt() function in the numpy package is used as an example function. (To save space in the drawing I have written sqrt() instead of np.sqrt().)

The example below illustrates vectorization with the np.sqrt() function. We are going to calculate the square root of 5 number values together in a vector (array).

```
>>> import numpy as np
>>> x = np.array([0, 10, 20, 30, 40])
>>> y = np.sqrt(x)
>>> y
array([0. , 3.16227766, 4.47213595, 5.47722558, 6.32455532])
```

How can we realize these 5 square root calculations without vectorization? Below is a solution, which is based on manually repeated calculations:

```
>>> import numpy as np
>>> x = np.array([0, 10, 20, 30, 40])
>>> y = np.zeros(5) # Preallokering
>>> y[0] = np.sqrt(x[0])
>>> y[1] = np.sqrt(x[1])
>>> y[2] = np.sqrt(x[2])
>>> y[3] = np.sqrt(x[3])
>>> y[4] = np.sqrt(x[4])
>>> y
array([0. , 3.16227766, 4.47213595, 5.47722558, 6.32455532])
```

The answer is the same as with vectorization, but achieved much more cumbersome.

Comment on the preallocation of y in the program above: Code line y = np.zeros (5) realizes so-called preallocation of the array y, i.e., it is set aside for the array y in the PC's memory before the elements in y get their correct values. We know that y becomes an array of 5 elements, so we can preallocate with e.g. the array np.zeros (5), which is [0, 0, 0, 0, 0]. In general, we can save a lot of execution time for our programs if we use preallocation of arrays that we know the length of. If we do not use preallocation, we must use the append() method to extend the array y with ever-new values – and the use of append() should be avoided if we can instead use preallocation. We will get closer to preallocation and time saving with preallocation in chap. 8.2.

Vectorization is not the only way to streamline repeated calculations. We can create a so-called for loop (python: for-loop) that runs through the loop 5 times, and for each time the square root of the relevant element in the vector (array) x is calculated. Although for loops are described in Chap. 8.2, let's use the for loop here (in the example below). The following two code lines are for the loop:

```
for k in range (0, 5)
    y[k] = np.sqrt(x[k])
```

For each k-value in the interval (0, 5), but – note – 5 is not included, y [k] = np.sqrt (x [k]) is calculated. The k-values are thus 0, 1, 2, 3 and 4. The for loop performs the square root calculation for each element in x, a total of 5 times.

```
>>> import numpy as np
>>> x = np.array([0, 10, 20, 30, 40])
>>> y = np.zeros(5)
>>> for k in range(0, 5):
        y[k] = np.sqrt(x[k])
>>> y
array([0. , 3.16227766, 4.47213595, 5.47722558, 6.32455532])
```

Vectorization used in ordinary mathematical expressions

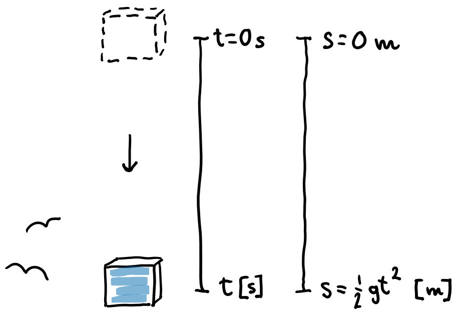


Figure 3.13: Body in free fall

Figure 3.13 shows a body in free fall, without air resistance, that was released at time 0 s. From physics, we know, that, at time t, the body has fallen the distance s [m] given by:

$$s = \frac{1}{2}gt^2 \tag{3.1}$$

$g = 9.81$ m/s^2 is the acceleration of gravity. Suppose we calculate s for each of the times fom. 0 tom. 5 s with time steps 1.0 s. The program below realizes this.[10]

[10]All Python prompts (>>>) indicate that the code is executed from a command line, but you can alternatively write all the code lines in a script and run the script.

```
>>> import numpy as np
>>> t_start = 0 # [s]
>>> t_stop = 5.0 # [s]
>>> Ts = 1.0 # [s]
>>> n = int((t_stop - t_start)/Ts + 1)
>>> g = 9.81 # [m/s2]
>>> t = np.linspace(t_start, t_stop, n)
>>> s = (1/2)*g*t*t
>>> t
array([0., 1., 2., 3., 4., 5.])
>>> s
array([ 0.  , 4.905, 19.62 , 44.145, 78.48 , 122.625])
```

Note this code line:

```
s = (1/2)*g*t*t
```

In this expression, s is calculated by the vectorized multiplication t * t where t is a vector (1D array) of times:

```
array([0, 1, 2, 3, 4, 5])
```

The expression t * t is calculated by Python like this (I drop for simplicity “np.array” here, and I also drop the decimal points):

```
t*t
= [0, 1, 2, 3, 4, 5]*[0, 1, 2, 3, 4, 5]
=> [0*0, 1*1, 2*2, 3*3, 4*4, 5*5]
= [0, 1, 4, 9, 16, 25]
```

Thus, vectorized multiplication. We could also have said element-wise multiplication of two vectors (arrays).

Something to think about

Vectorized calculation, as shown above, is elegant and efficient. But also a little scary. If a mathematician had flipped through these pages of the book without knowing the context (which is vectorized calculations in Python), I would think she/he had interpreted t * t as *scalar* with 55 as the result:

$$t \cdot t = [0,1,2,3,4,5]\cdot[0,1,2,3,4,5] = [0\cdot 0+1\cdot 1+2\cdot 2+3\cdot 3+4\cdot 4+5\cdot 5] = 55$$

But the Python term t * t does represent scalar product here. If you really need to calculate the scalar product $t \cdot t$, you can write the code t@t which will be ok since t is a 1D array. Alternatively, you can define t as a row vector in the form of a 2D array, and calculate the scalar product with t@t.T, cf. Section 3.12.5.3. Another alternative for calculating the scalar product, is the vdot function: np.vdot (t, t).

Summary

There are alternative ways of realizing repeated function calculations. They can be ranked this way, by both ease of use and efficiency:

1. Vectorization
2. For Loops
3. Manually repeated calculations

Chapter 4

Presenting data in charts and diagrams

4.1 Introduction

Probably the most commonly used Python package for presenting data graphically in charts and diagrams is matplotlib, which has website on

https://matplotlib.org

Matplotlib allows for a variety of different types of graphical presentations. We will in this chapter take a look at some of the most common types, namely:

- line plots
- bar charts
- pie charts
- histograms

4.2 Line plot

As a starting point to learn more about plotting, we will use the program for plotting temperatures in Skien from Chap. 1.2. The program is reproduced below, but now - for simplicity - almost without comments in the code. I commented the program in Chap. 1.2. I will now provide even more comments, and then show some extensions of the program that I assume are of particular interest.

4.2.1 Basic plot functions

Running the program below generates the line plot shown in Figure 4.1.

Program name: prog_plot_temp_skien.py

```
import numpy as np
import matplotlib.pyplot as plt

mnd = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]) # month
no.
temp = np.array([-3, -2, 2, 7, 11, 15, 17, 16, 12, 6, 2, -3]) # deg C
mean_temp = np.mean(temp) # Mean values

# Plotting av temperaturverdiene:

plt.close('all')
plt.figure('Monthly mean temp i Skien')
plt.plot(mnd, temp, 'o-')
plt.xlim(1, 12)
plt.ylim(-5, 20)
plt.title('Monthly mean temp in Skien year 2005-2015')
plt.xlabel(Month no.')
plt.ylabel(Deg C')
plt.grid()
plt.show()

plt.savefig('temp_skien.pdf')
```

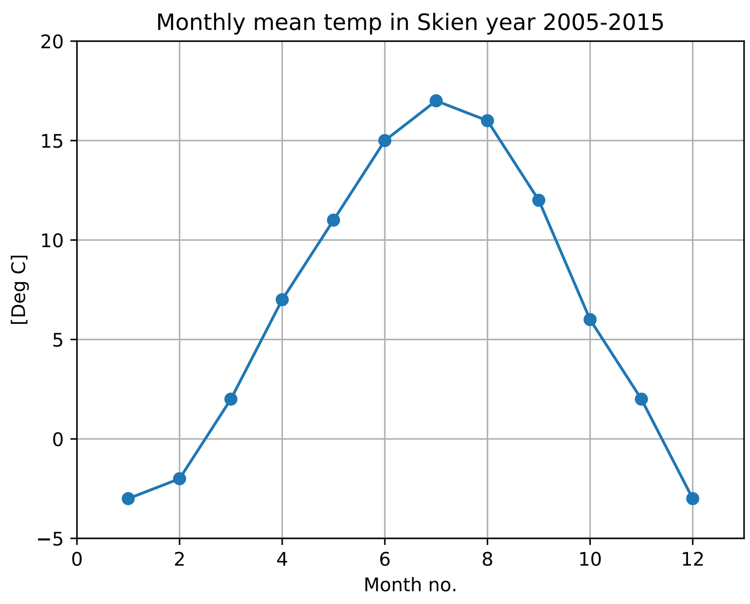


Figure 4.1: Line plot of montly mean temperatures in Skien

Comments on the part of the program concerning plotting (here is a little repetition from chap. 1.2):

- The code import matplotlib.pyplot as plt imports the pyplot module which is part of the matplotlib package and makes the module available to us through the name plt. This rename is Python tradition. the pyplot module allows you to set e.g. line type (e.g., dotted), line color (e.g., blue), grid, descriptive text, etc.
- The code plt.close ('all') closes all existing figure windows, which may be fine to clean up and reset before new plotting.
- The code plt.figure ('Monthly mean temp in Skien') opens a figure window which gets a "number" similar to the text 'Monthly mean temp in Skien', and prepares the figure for plotting with subsequent plot commands. You can also use numbers such as figure numbers, e.g. plt.figure (1).
- The code plt.plot (month, temp, 'o-') plots the temp (y values) array against the mnd (x values) array. Basically, the points (month [0], temp [0]), (month [1], temp [1]), etc. are plotted. Of course, these arrays must have the same number of elements. In this example, data in arrays is plotted, but lists can also be plotted.

- The text string 'o-' in the code plt.plot (month, temp, 'o-') determines the appearance of the curve: 'o' means that each point is plotted with a filled circle. '-' indicates the straight line between the points. You can also set color. The codes can be combined fairly freely, as 'o-' is an example of. The most relevant codes are:
  - Dot mark: 'o' for open circle. '*' for star. '.' (dot) for dot. Default (ie the dot mark that Python selects, if no code is entered) is dot.
  - Line (interpolation) between the points: '-' for the solid line. '-' for dotted line. '-.' for line + point. Default is the solid line.
  - Color: 'b' for blue. 'r' for red. 'y' for yellow. 'g' for green. 'k' for black. Default is blue.
- The code plt.xlim (1, 12) and the code plt.ylim (-5, 20) indicate the smallest and largest value along the x-axis and the y-axis, respectively.
- The code plt.xlim (1, 12) and the code plt.ylim (-5, 20) indicate the smallest and largest value along the x-axis and the y-axis, respectively.
- The codes plt.xlabel ('Month no.') and plt.ylabel ('Deg C') is written along the x-axis and the y-axis, respectively. The arguments are text strings.
- The code plt.grid() creates a grid.
- The code plt.show() ensures that the figure is displayed. In Spyder, the figure is shown even though plt.show() is omitted, but in other programming environments, eg. Visual Studio Code, it may be necessary to include plt.show().
- The code plt.savefig ('temp_skien.pdf') creates a pdf file of the plot. If you want a png file, replace 'temp_skien.pdf' with 'temp_skien.png'. The same applies to jpg files. The file is stored in the same directory where the script is stored.

Plot in multiple figure windows?

So far we have plotted in a figure window. If you want to have multiple plots in their own figure window, open new figure windows with respective plt.figure (figure name) expressions, where figure names can be text or integer, e.g.

- plt.figure('Monthly temp in Porsgrunn') or

- plt.figure(2).

After the plt.figure() expression, type the appropriate plot commands, which will then apply to the last opened figure window.

4.2.2 Viewing Plots in the Spyder Console or External Window?

You can decide whether a plot figure (plot Figure) should appear in the Spyder console or in an external window. The two options are discussed in more detail below.[1]

4.2.2.1 Plot figures to be shown in the Spyder console

Figure 1.6 shows a Figure of a plot shown in the Spyder console. An advantage of such a view is that then the figure will be displayed together with the result of calculations, in the same order as in the program code. One disadvantage is that the figure becomes quite small since it must fit in the console.

How do you get Spyder to show characters in the console? One way is to execute the command

```
%matplotlib inline
```

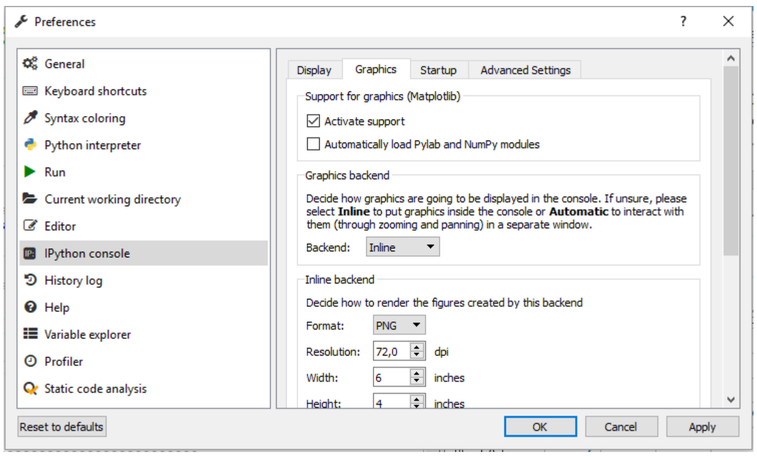
on the console command line. % matplotlib inline is an example of a so-called magic command, which belongs to IPython (Interactive Python), which is the user interface to Python implemented in e.g. Spyder.

An alternative to the magic command mentioned above is the following menu options in Spyder:

Tools / Preferences / IPython console / Graphics / Graphics backend / Backend

where you select Inline in the menu, see Figure 4.2.

[1] I myself tend to use external window.



Support for graphics (Matplotlib)

☑ Activate support

☐ Automatically load Pylab and NumPy modules

Graphics backend

Decide how graphics are going to be displayed in the console. If unsure, please select **Inline** to put graphics inside the console or **Automatic** to interact with them (through zooming and panning) in a separate window.

Backend: Inline

Inline backend

Decide how to render the figures created by this backend

Format: PNG

Resolution: 72,0 dpi

Width: 6 inches

Height: 4 inches

Figure 4.2: How to set Graphics backend to Inline via the Tools / Preferences menu in Spyder

By right-clicking on the figure in the console, you get the following two options:

- Copy Image, which makes a copy of thefFigure on the PC's clipboard. From there you can paste the figure into a word processor (eg MS Word) or a drawing program (eg with the shortcut ctrl + V).
- Save Image As, which allows you to save the image as a png file.

The choice to display plot figures inline, ie in the console, remains until you choose to display the plot figures in an external window. Te choice is remembered between each time you run programs, and it is also remembered if you exit Spyder and enter again.

Plot figures to be shown in an external window

Plot figures can be displayed in an external window (outside the Spyder window), see Figure 4.3.

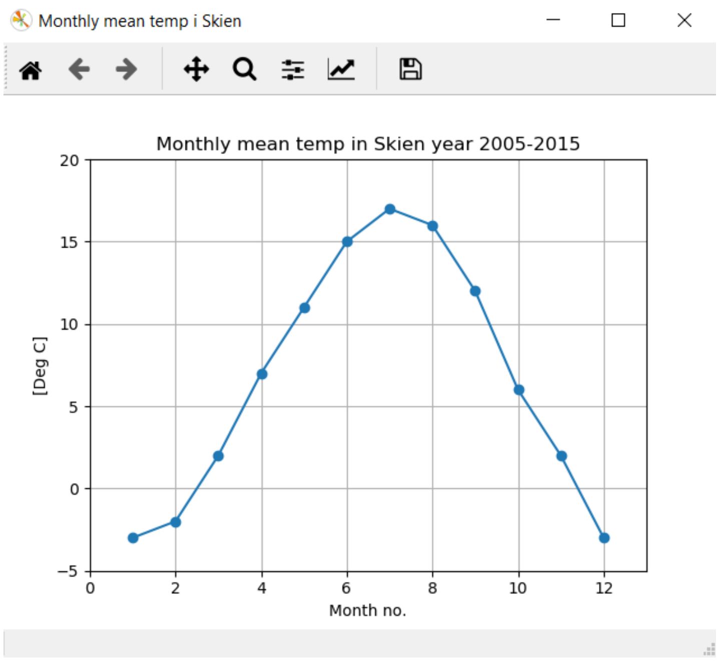

Monthly mean temp i Skien
Monthly mean temp in Skien year 2005-2015
20
15
10
5
0
−5
[Deg C]
0
2
4
6
8
10
12
Month no.

Figure 4.3: A plot figure shown in an external window (outside the Spyder window)

An advantage of displaying the plot the figure in an external window is that the figure becomes larger than it appears in the console. Another advantage is that the figure window provides a number of possibilities for figure processing, including choose the appearance of curvature lines. color, line type, etc., zoom, saving the figure to a file, including pdf, png and jpg.[2]

I do not go through all the menu choices in the figure window (they are easy to find out).

How do you get Spyder to display characters in an external window? One way is to execute the magic command

```
%matplotlib auto
```

on the console command line.

An alternative to this magic command is the following menu options in

[2]pdf should be used if the text editor allows, since pdf is based on vector graphics, which provides maximum resolution, ie best image quality. In this book I use pdf graphics as much as possible.

Spyder:

Tools / Preferences / IPython console / Graphics / Graphics backend / Backend

where you select Automatic in the menu, see Figure 4.2.

4.2.3 How to plot multiple curves at the same time

4.2.3.1 Multiple curves in one and the same diagram

The program code below shows how several - here two - curves can be plotted in one and the same diagram. The two curves are the monthly temperature values (temp) and the mean (mean_temp) of these temperature values.

The code also shows how each curve can be identified with a curve legend realized with the plt.legend() function. Figure 4.4 shows the plot.

Program name: prog_plot_temp_skien_several_curves.py

```
import numpy as np
import matplotlib.pyplot as plt



month = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
temp = np.array([-3, -2, 2, 7, 11, 15, 17, 16, 12, 6, 2, -3])

mean_temp = np.mean(temp)

mean_temp_array = np.zeros(len(temp)) + mean_temp

# Plotting av temperaturverdiene:
plt.figure(1)
plt.plot(month, temp, 'o-b', month, mean_temp_array, 'g')
plt.title('Monthly mean temp in Skien 2005-2015')
plt.xlabel(Month no.')
plt.ylabel('Deg C')
plt.legend(labels=('Temperature', 'Mean temp'),
           loc='upper right',
           handlelength=2,
           fontsize=8)
plt.grid()
plt.show()
```

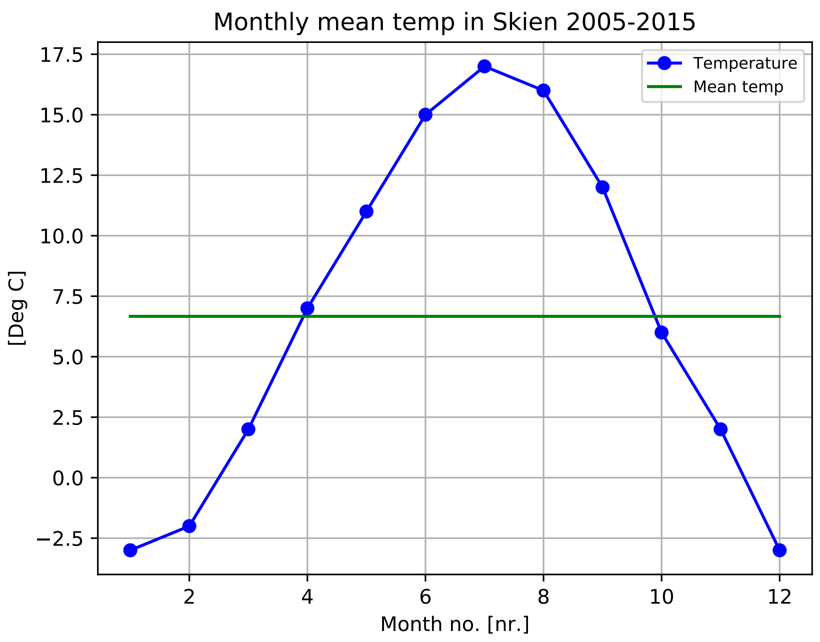


Figure 4.4: Multiple curves plotted in a single diagram identified by curve description generated by the plt.legend() function

Note how the plt.plot() function is used to plot multiple curves in one chart:

```
plt.plot(mnd, temp, 'o-b', mnd, mean_temp_array, 'g')
```

where the text string 'o-b' means circles connected by straight lines of blue color and the text string 'g' means green color.

The general syntax for plotting more than one curve (here two) is

```
plt.plot(x1_array, y1_array, 'formatting1', x2_array, y2_array, 'formatting2')
```

I've included here "array" in the variable names just to emphasize that the data type is array, but you are of course free to choose the variable name.

Generally in such composite plots, x1_array and x2_array may have different lengths (number of elements), but x1_array and y1_array must have equal lengths, and x2_array and y2_array must have equal lengths. In our example, all the arrays have equal lengths.

For the plot of the mean temperature, we need an array of length equal to the temp array and with mean_temp as the value for all the elements. In the program, this array is created with this code:

```
mean_temp_array = np.zeros(len(temp)) + mean_temp
```

Alternatively, I could have used this slightly shorter code:

```
mean_temp_array = temp*0 + mean_temp
```

Curve description with plt.legend()

In the upper right of the plot figure in Figure 4.4 it is a curve description (legend) realized with the plot.legend() function. Comments:

- The labels parameter specifies the text for each curve.
- The location parameter is set to 'upper right'. The possibilities are:
  - best (default)
  - upper right
  - upper left
  - lower left
  - lower right
  - right
  - center left
  - center right
  - lower center
  - upper center
  - center
- The handle length parameter specifies the length of the relatively short selection curve in the curve description measured by the font size as a unit.
- The font size parameter specifies the font size.

Note: If the curve description applies to only one curve, let's say only the Temperature curve, you must include a comma and blank:

```
plt.legend(labels=('Temperature', )
```

4.2.3.2 Multiple plots in a figure using subplot

You can have multiple plots in a figure window. These plants are called subplots. Subplot is organized with the function

plt.subplot(m,n,no)

where m is the number of rows, n is the number of columns and n is the number of the plot, see Figure 4.5. no is 1 for the subplot in the upper left and is counted from there to the right and then down as shown in Figure 4.5. We can designate all the subplots together as a (n, m) plot.

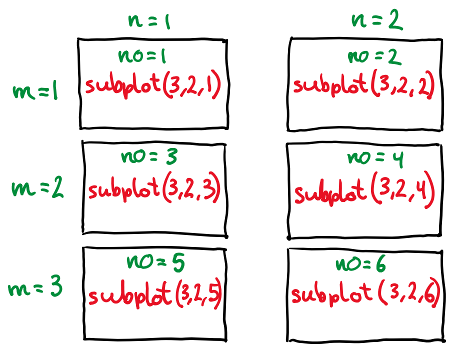


Figure 4.5: Organization of the subplots in a (n, m, no) plot

Below is an example of a (2x1) subplot.

Example 4.1 *Subplot*

```
import numpy as np
import matplotlib.pyplot as plt


x1_array = np.array([0, 1, 2])
y1_array = x1_array * 10
x2_array = np.array([0, 1, 2])
y2_array = x1_array * (-10)
plt.figure(1)
plt.subplot(2,1,1)
plt.plot(x1_array, y1_array)
plt.xlabel('x1')
plt.ylabel('y1')
plt.grid()
plt.subplot(2,1,2)
plt.plot(x2_array, y2_array)
plt.xlabel('x2')
plt.ylabel('y2')
plt.grid()
plt.show()
```

Figure 4.6 shows the diagram with the two subplots.

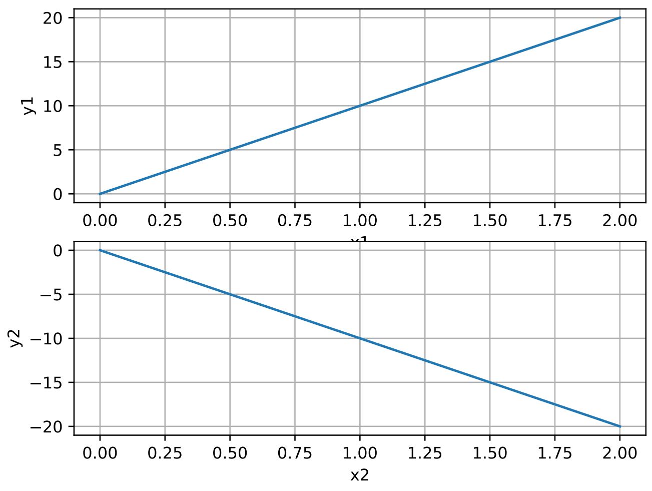


Figure 4.6: Subplott

[End of Example 4.1]

4.2.4 Mathematical symbols in chart title

You can include beautiful mathematical symbols in plt.title(), plt.xlabel() and plt.ylabel(). For this use codes from Latex, which is a text formatting system for books, scientific articles, etc. Example 4.2 demonstrates this.

Example 4.2 *Latex-code in plt.title()*

Some random data are plotted in the program shown below. Take a close look at the relationship between the code in the plt.title() argument and the plot shown in Figure 4.7. I guess you see how Latex code can be used (I won't explain this in detail).

Note:

- The letter r is just before the text strings with Latex code, e.g. r '$ x_ {a_1} $'. r stands for "raw text". Without r, the Latex codes will not work.
- The '\ n' text line (newline) is used to create more lines than one.

Programmets navn: prog_plot_latex.py

```
import numpy as np
import matplotlib.pyplot as plt


x_array = np.array([0, 1, 2])
y_array = x_array * 10
plt.figure(1)
plt.plot(x_array, y_array)
plt.title('Beautiful:' + '\n'
          + r'$x_{a_1}$'
          + ' and ' + r'$\sqrt{\alpha\beta\gamma\Omega\pi\theta\tau}$'
          + ' and ' + r'$\int_0^t z d\tau$'
          + ' and finally ' + r'$\sum_{k=0}^{N-1} y(k)$')
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.show()
```

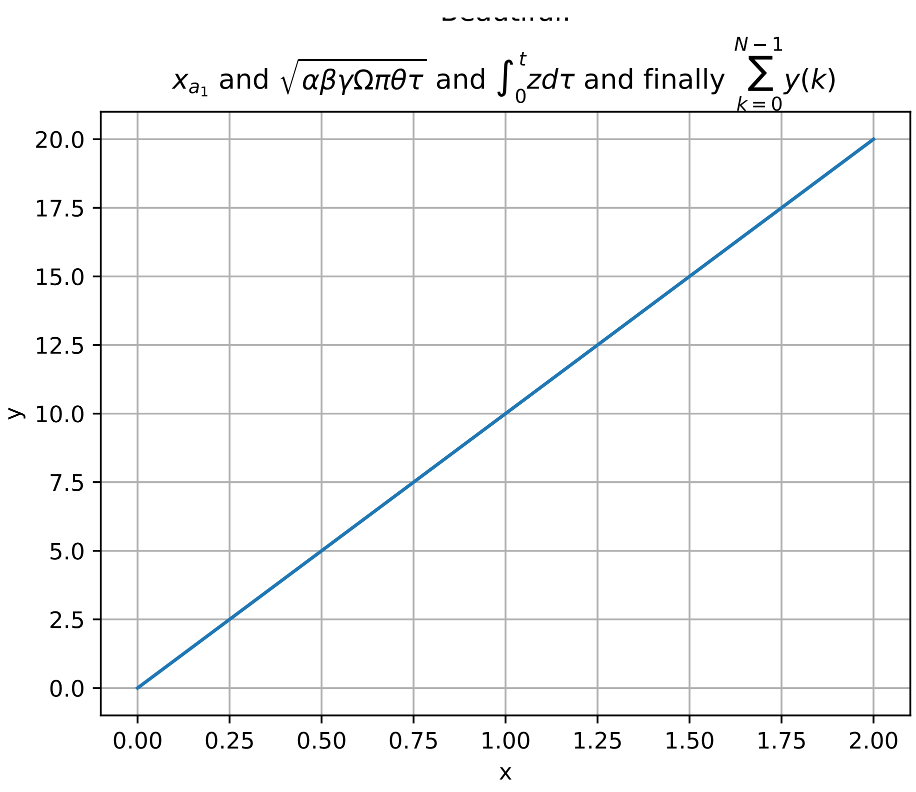


Figure 4.7: Latex code is used in plt.title().

[End of Example 4.2]

4.2.5 How to set the size of the plot figure

You can set the size of a plot figure using. the argument figsize in the plt.figure() function:

plt.figure(num=1, figsize=(fig_width_inch, fig_height_inch))

where figsize is a tuple of two values, namely fig_width_inch which is the figure's width in inches, and fig_height_inch which is the figure's height in inches. The width and height in inches can of course be calculated from the width and height indicated in e.g. centimeter.

Example 4.3 *Latex code in plt.title()*

The program in the example below plots x_array vs. y_array (which consists of customs data) in a figure window with width 24 cm and height 18 cm. (The figure itself does not appear here as it contains no interesting information.)

```
import numpy as np
import matplotlib.pyplot as plt

x_array = np.array([0, 1, 2])
y_array = x_array * 10

fig_width_cm = 24
fig_height_cm = 18
cm_per_inch = 2.54
fig_width_inch = fig_width_cm/cm_per_inch
fig_height_inch = fig_height_cm/cm_per_inch

plt.figure(num=1, figsize=(fig_width_inch, fig_height_inch))
plt.plot(x_array, y_array)
plt.title('Tittel y vs. x')
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.show()
```

[End of Example 4.3]

4.3 Bar charts

Example 4.4 demonstrates presenting data in a bar graph.

Example 4.4 *Bar graph*

The program below generates the bar graph shown in Figure 4.8.

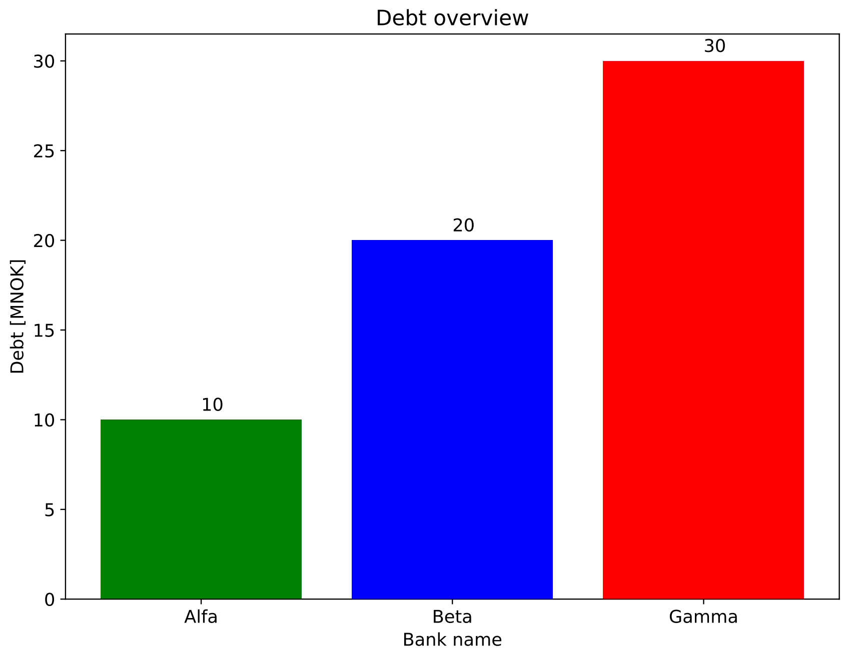


Figure 4.8: Bar graph

Program name: prog_bar_graph.py.

```
import matplotlib.pyplot as plt
import numpy as np

# Data to be plotted:
x = ['Alfa', 'Beta', 'Gamma']
y = np.array([10, 20, 30])

# Size of figure window:
fig_width_cm = 24
fig_height_cm = 18
plt.figure(num=1, figsize=(fig_width_cm/2.54, fig_height_cm/2.54))

# Plots a bar graph:
plt.bar(x, y, width=0.8, color=('green', 'blue', 'red'))

# Plots text on top of each bar using plt.txt():
offset_y = 0.5
for k in range(0, len(x)):
    plt.text(x[k], y[k]+offset_y, str(y[k]))

plt.title('Debt overview')
plt.xlabel('Bank name')
plt.ylabel('Debt [MNOK]')

#plt.show()
```

[End of Example 4.4]

4.4 Pie charts

Example 4.5 demonstrates plotting data in a pie chart.

Example 4.5 *Pie chart*

The program below generates the pie chart shown in Figure 4.9.

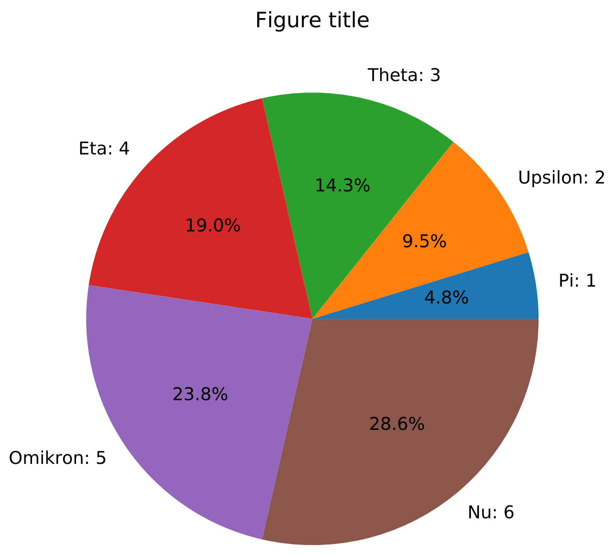


Figure 4.9: Pie chart

Program name: prog_pie_chart.py.

```
import matplotlib.pyplot as plt
import numpy as np

# Data and text to be displayed in the pie diagram:
data = np.array([1, 2, 3, 4, 5, 6])
data_normalized = data/sum(data)
my_labels = ['Pi: ' + str(data[0]),
             'Upsilon: ' + str(data[1]),
             'Theta: ' + str(data[2]),
             'Eta: ' + str(data[3]),
             'Omikron: ' + str(data[4]),
             'Nu: ' + str(data[5])]

# Size of figure window:
fig_width_cm = 24
fig_height_cm = 18
plt.figure(num=1, figsize=(fig_width_cm/2.54, fig_height_cm/2.54))

# Pie diagram:
plt.rc('font', size=12) # rc is abbrev for 'run configuration'
plt.pie(normalized, labels=my_labels, autopct='%.1f%%')
plt.title('Figure title')

#plt.show()
```

[End of Example 4.5]

4.5 Histogram

Example 4.6 demonstrates plotting data in a histogram.

Example 4.6 *Histogram*

The program shown below generates the histogram shown in Figure 4.10.

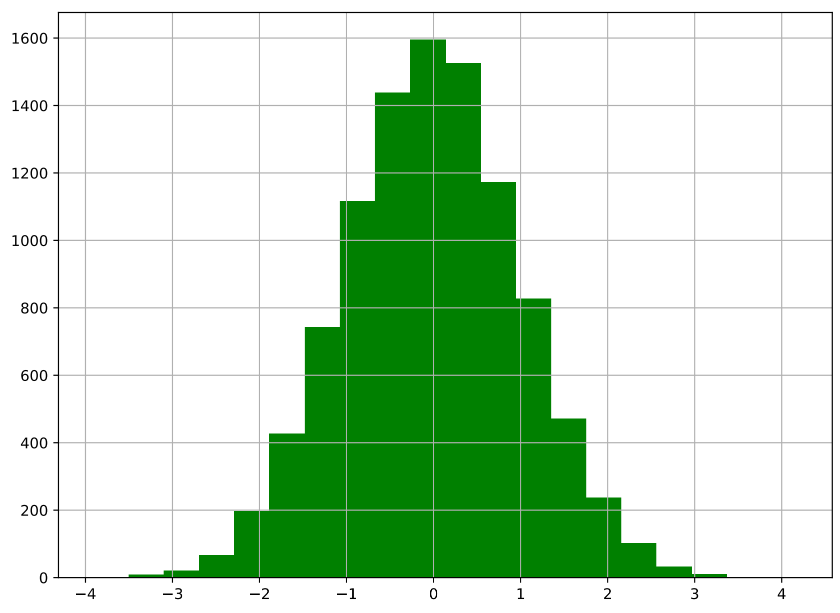


Figure 4.10: Histogram

Program name: prog_histogram.py

```
import numpy as np
import matplotlib.pyplot as plt

# Data to be plotted:
n = 10000 # Number of random values
x = np.random.randn(n) # The random values
num_bins = 20 # Number of bins

# Size of figure window:
fig_width_cm = 24
fig_height_cm = 18
plt.figure(num=1, figsize=(fig_width_cm/2.54, fig_height_cm/2.54))

# Plotting the histogram:
plt.hist(x, num_bins, color='green')
plt.grid()

#plt.show()
```

[End of Example 4.6]

4.6 Interactive plots

Interactive plots are plots that are updated after the user has manipulated the user interface widgets in the figure window itself. You can create interactive plots in Matplotlib.

Typical user interface elements – often called GUI-elementes (GUI = Graphical User Interface) – are:

- Text in textboxes, which are interpreted as numbers (pytonsk: textboxes)
- Buttons
- Radio buttons
- Sliders

These GUI elements are objects that have methods (functions) that respond to – or are fired by – events such as value change, button click and press the Enter key. And these methods can be used to call any function that we have created ourselves, e.g. a function to update a function value and plot the new value. Here is a small taste of Example 4.7:

textbox_a.on_submit(fun_submit_a)

there:

- **textbox_a** is a textbox object that represents a textbox that is embedded in the character window itself.
- **on_submit** is a method associated with the textbox object. The method "fires off" or is activated by an "on_submit" event, which is that the user has pressed the Enter key after a new value is entered in the text box. When firing, the method calls a user-defined function which here is assumed to have the name fun_submit_a, which does one or the other, e.g. performs a calculation or updates a plot.
- **fun_submit_a** is the name of the user-defined function.

We shall now study an example where a curve with two parameters, a and b, is plotted. When we change a or b via the respective text boxes in the figure window, the plot changes *immediately.*

Example 4.7 *Interactive plot with adjustable numbers in textboxes*

Program name: prog_interactive_plot_textbox.py

The curve to be plotted is given by the following function:

$$y = ax + b \tag{4.1}$$

where x is in the range [0, 2] of solution 0.001. The default values are $a = 1$ og $b = 0$. Figure 4.11 shows a plot of (4.1) with these default values, and Figure 4.6 shows a plot after that a and b has been changed to $a = -1$ respectively $b = 1$ via the textboxes in the figure window.

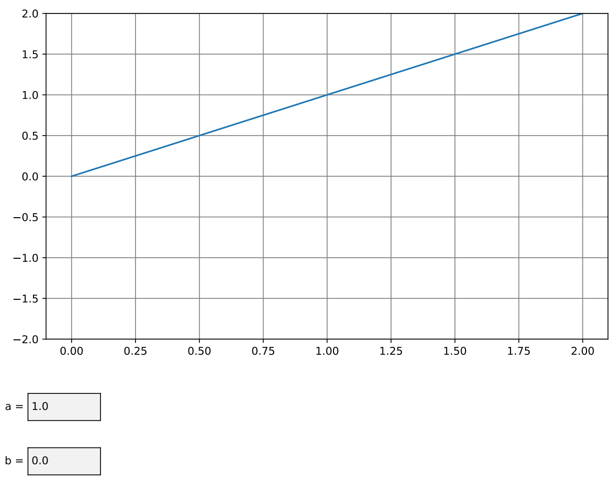


Figure 4.11: Plot of (4.1) with $a = 1$ and $b = 0$

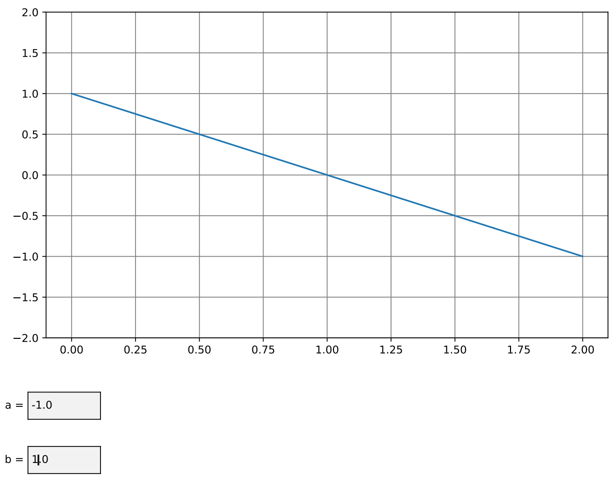


Figure 4.12: Plot of (4.1) with $a = -1$ and $b = 1$

The program that realizes this is shown below.[3]

[3]The program is based on the program https://matplotlib.org/gallery/widgets/textbox.html#sphx-glr-gallery-widgets-textbox-py, which I have modified so that there are two text boxes instead of one.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import TextBox

# Function for updating plot with new value of param a:
def fun_submit_a(dummy_arg):
    a = eval(text_box_a.text)
    b = eval(text_box_b.text)
    y_array = a*x_array + b
    line_1.set_ydata(y_array)
    # ax.set_ylim(np.min(y_array), np.max(y_array))
    plt.draw()

# Function for updating plot with new value of param b:
def fun_submit_b(dummy_arg):
    a = eval(text_box_a.text)
    b = eval(text_box_b.text)
    y_array = a*x_array + b
    line_1.set_ydata(y_array)
    # ax.set_ylim(np.min(y_array), np.max(y_array))
    plt.draw()

# Opening new Figure with one positioned plot
plt.close('all')
fig_width_inch = 24/2.54
fig_height_inch = 18/2.54
(fig, ax) = plt.subplots(num='Interactive plot',
                         figsize=(fig_width_inch, fig_height_inch))
left=0.125; bottom=0.3; right=0.9; top=0.9
plt.subplots_adjust(left, bottom, right, top)
plt.ylim(-2, 2)

# To be continued:
```

```
# Continued:

# Generating and plotting y_array using initial values of a and b:
x_array = np.arange(0.0, 2.0, 0.001)
a = 1
b = 0
y_array = a*x_array + b
(line_1, ) = plt.plot(x_array, y_array)
plt.grid(which='both', color='grey')

# Generating textboxes for a and b:
left_a=0.1; bottom_a=0.15; width_a=0.1; height_a=0.05
left_b=0.1; bottom_b=0.05; width_b=0.1; height_b=0.05
axbox_a = plt.axes([left_a, bottom_a, width_a, height_a])
axbox_b = plt.axes([left_b, bottom_b, width_b, height_b])
caption_a = 'a = '
caption_b = 'b = '
initial_text_a = "1.0"
initial_text_b = "0.0"
text_box_a = TextBox(axbox_a, caption_a, initial_text_a)
text_box_b = TextBox(axbox_b, caption_b, initial_text_b)

# Invoking the functions named fun_submit_a() or fun_submit_b()
# on click on Enter button of a or b:
text_box_a.on_submit(fun_submit_a)
text_box_b.on_submit(fun_submit_b)

plt.show()
```

Comments on the program (but only comments regarding the realization of interactive plot with textbox):

- Code line from matplotlib.widgets import TextBox imports the textbox object.
- The code line def fun_submit_a (dummy_arg): is the beginning of the definition of the function fun_submit_a(). This function uses the updated values of a and b to calculate the updated value of y_array and plot y_array vs. x_array. The fun_submit_a() function is called only when the user has entered a new value in the text box for parameter a (not b). dummy_arg is an argument that is not used, but Python returns an error message if that argument is dropped.
- The code line a = eval (text_box_a.text) interprets the text

text_box_a.text, which is the text in the text box element for parameter a, as a number.

- The code line b = eval (text_box_b.text) is like the code line above, but for parameter b.
- The code line line_1.set_ydata (y_array) updates the plot identified with the object called line_1. This identification occurs in the code line (line_1,) = plt.plot (x_array, y_array) in the main part of the program.
- The code line # ax.set_ylim (np.min (y_array), np.max (y_array)) is actually "commented away", but I've included it to show how you can adjust the value range along the y axis to the updated value of y_array.
- The plt.draw() code line ensures that the plot is updated.
- The code line def fun_submit_b (dummy_arg): is the beginning of the definition of the function fun_submit_b(). The comments here are as for the fun_submit_a() function.
- The code line under # Opening new Figure with one positioned plot opens a figure window with a plot figure of specified size and location:
  - The size is indicated by the figsize argument.
  - The location is specified with the code plt.subplots_adjust (left, bottom, right, top) where the unit for the four parameters is the height and width of the figure window. You can count to get the desired size and position, but a little trial and error will probably work well.
- The code under # Generating and plotting y_array based on initial values of a and b: generates an initial plot. The plot is updated if the user sets new values for parameters a and b in the respective text boxes.
- The code line line_1, = plt.plot (x_array, y_array) creates an object called line_1, which is used as a reference to the plot Figure in the functions fun_submit_a and fun_submit_b.
- The code under # Generating textboxes for a and b: generates the text boxes for parameters a and b. The unit for left_a, bottom_a, etc. is the height and width of the figure window. Trial and error to get usable size and placement of text boxes works well.
- Code line text_box_a.on_submit (fun_submit_a):

    - text_box_a is the object that represents the text box for parameter a.
    - on_submit is a method of the text_box_a object. When the user presses the Enter key in the text box, the function is called fun_submit_a, which the user has defined (see comment on this function further up in this comment list). Actually, the text in the text box is transferred to this function, but I have chosen not to use this text in the fun_submit_a() function. Instead, I make sure to read the value of the text attribute of the text_box_a object, as well as the text attribute of the text_box_b object, in the fun_submit_a() function. Thus, the fun_submit_a() function will use the updated values of both a and b whenever the y_array is updated.

- Code line text_box_b.on_submit (fun_submit_b): Here the comments are exactly the same as for the code line text_box_a.on_submit (fun_submit_a) above.

[End of Example 4.7]

Chapter 5

Programming of functions

5.1 Introduction

A little rehearsal from chap. 3.4 about functions: A function "does something" with the value of the function's input or input argument, and produces a result. When the result is a value, e.g. the square root of the input argument, is called the result output or output argument, see Figure 5.1. But not all functions produce values. Some “do a job”, e.g. plots data or writes data to a file.

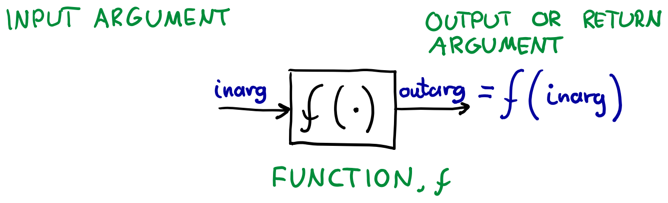


Figure 5.1: Function with input argument and output argument

At the front of the book we have met many pre-programmed functions. Some examples:

- int(), float(), len(), print() which is in the library of inbuilt functions in Python, see Figure 2.26. We can also say that this library is Python's standard suite of functions.,
- np.array(), np.linspace(), np.mean(), etc. which is included in the numpy package (here represented by “np” in front of the function

name itself).

- plt.plot(), plt.xlabel(), plt.grid() which is included in the pyplot module in the matplotlib package (pyplot is represented here with "plt" in front of the function name).

Why are these functions made? Some important reasons are:

- **Support for reuse**: The functions perform a well-defined task and can be used by all programmers and in many different applications.
- **Good program structure**: The functions give the programs a simpler and more transparent structure by solving sub-tasks using. a single function call. For example, the np.mean() function calculates the mean of a data series with one function call, and you do not have to write program code for this from scratch.
- **Less chance of error coding**: Programming a program part with a given functionality from scratch every time you need the functionality increases the chance of error coding compared to creating a function (which hopefully is flawless) and reusing it.

It is likely that in our programs we will need functionality that is not fully covered by any of the completed functions. Python allows us to create functions ourselves, and the reasons for doing so are exactly the same as mentioned above.

5.2 How to program functions

5.2.1 Basic function definition

As an example, let's create a function that calculates the value of y according to. the mathematical formula (function)

$$y = ax + b \tag{5.1}$$

There a and b are parameters (constants). x is input variable or independent variable. y is the output variable or dependent variable. The program below contains *definition* of the function and code that *uses* the function. The part of the program that is outside the function definition is often called the main program.

Example 5.1 *Programming of function*

Program name: prog_fun_intro.py

```
def fun_lin(x, a, b):
    y = a*x + b
    return y


p0 = 2.5
p1 = 3.0
inn = 10.0
out = fun_lin(in, p0, p1)
print('Result = ', out)
```

Running the program:

```
Result = 28.0
```

Comments:

- I have named the function fun_lin. In general, you are quite free to choose file names, but cf. 3.3.3.
- The definition of the function starts with the command def followed by the file name and the function's arguments, which are the three arguments x, a and b, separated by a comma in parentheses.
- It is necessary that the definition is before the function is used (in the program). I have – as usual – written the definition of the function at the beginning of the program, after the comments about the program itself indicated at the very top of the program.[1]
- The three input arguments x, a and b are used as variables in the function's program code, which is the program code that expresses what the function does. These three variables are local to the function – or belong to the function namespace, which means that they (local variables) do not exist for use outside the function. Which namespaces that variables belong to, we look at in more detail in Chap. 5.3. The local variables are also called the function's so-called formal parameters.

[1]If you know MATLAB programming, you know that functions must be defined there *in the end* of the program.

- function program code or functional body consists of two lines of code:
  - Code line y = a * x + b, which calculates the value of y as a function of x, a and b.
  - The code line return y, which expresses that the function returns (to the main program) the calculated value of y. We say that y is the return argument or output argument of the function.
- The functional body stands *indented* 4 spaces in relation to the def command, which is according to. Python Enhancement Proposal PEP 8 rules at python.org/dev/peps/pep-0008/ for good Python programming: "Use 4 spaces per indentation level". In other program languages, brackets, such as {...}, are used to mark the function body.
- Two blank lines are inserted between the end of the function definition and the following program code in the main program, cf. the PEP 8 rules.
- The program code
  p0 = 2.5
  p1 = 3.0
  inn = 10.0
  (in the main program) defines three variables with respective values.
- Program code out = fun_lin (in, p0, p1) in the main program *calls* the function with the call arguments in, p0 and p1, ie the main program asks the fun_lin() function to return the value of the function calculated from the actual values of the call arguments, and this return value becomes 28.0. The function is called with the three input arguments, p0 and p1, which constitute the function's so-called actual parameters ifm. function call. Note that the actual parameters may have different names than the formal parameters, but they must be in the same order. (It is actually possible to circumvent the requirement for the same order, but we will not go into that.)
- The code print ('Result =', out) shows the text string 'Result =' and the value of the output variable in the console.

[End of Example 5.1]

5.2.2 How to return more than one value

In the example above, the function fun_lin() has a return argument which has one value, namely the floating point y from the formula or function $y = ax + b$. But in general, it is curant to create functions that return multiple values: Just make sure that the function returns e.g. a tuple or list or array, which are data types that can consist of multiple elements.

Below is an example of a program where the function returns two values together in one tuple. The program is a modified version of the program fun_intro.py, see page 5.2.1.

Example 5.2 *Function that returns multiple values with tuple*

Program name: prog_fun_tuppel_retur.py.

```
def fun_value_and_slope(x, a, b):
    y = a*x + b
    dy = a
    return (y, dy)


p0 = 2.5
p1 = 3.0
x_value = 10.0
(value, slope) = fun_value_and_slope(x_value, p0, p1)
print(Function value = ', value)
print(Slope = ', slope)
```

Running the program gives the following results:

```
Function value = 28.0
Slope = 2.5
```

Comments:

- The function fun_value_and_slope now returns both the function value $y = ax + b$ and the slope of the function $dy = a$ (agree, the latter operation is pretty trivial), collected in the tuple (y, dy).
- The tuple (value, slope) gets a value equal to the returned tuple value.

[End of Example 5.2]

5.2.3 Default value function arguments

A function input argument can be defined with a default value. If for any reason the user has chosen to *not* enter this argument in the function call, the function uses the argument's default value.

Below is an example of a function with a default value argument. The program is a modified version of the program prog_fun_intro.py, see page 5.2.1.

Example 5.3 *Function with an argument with default value*

Program name: prog_fun_default_arg.py.

```
def fun_lin(x, a, b=5.0):
    y = a*x + b
    return y

x_value = 10.0
p0 = 2.5
p1 = 3.0

out = fun_lin(x_value, p0)
print('Resultat = ', out)
```

Running the program gives the following results:

```
Resultat = 30.0
```

Comments:

- In the function definition, ie in the expression def fun_lin (x, a, b = 5.0), the argument b is defined with a default value 5.0.
- In the function shell, ie in the expression out = fun_lin (in, p0), values are specified for the first two arguments. Since the value of the third argument (b) is omitted in the function call, Python uses the default value of b.
- The default value arguments must follow the non-standard value arguments in the function definition. For example, the function definition def fun_lin (x, a = 4.0, b) is incorrect because b stands *after* a = 4.0.

[End of Example 5.3]

5.2.4 Function call using keyword argument

You can call a function with explicitly naming of the names of the formal function arguments. This is called function calling with keyword arguments as actual arguments.

In function calls with keyword arguments, the order of the arguments does not matter, which is natural since the arguments are crystal-clear in that the formal argument names are used.

We continue with the example above (the prog_fun_default_arg.py program), but now add two different ways to call the function - with and without keyword arguments.

Example 5.4 *Function with a keyword argument*

Program name: prog_fun_keyword_arg.py.

```
def fun_lin(x, a, b):
    y = a*x + b
    return y

x_in = 10.0
p0 = 2.5
p1 = 3.0

out1 = fun_lin(in, p0, p1)
out2 = fun_lin(a=p0, b=p1, x=x_in)

print('out1 = ', out1)
print('out2 = ', out2)
```

Running the program gives the following results:

```
out1 = 28.0
out2 = 28.0
```

Comments:

- Function calls *without* keyword arguments:

out1 = fun_lin(in, p0, p1)
that produces results 28.0.

- Function calls *with* keyword arguments:
  out2 = fun_lin(a=p0, b=p1, x=in)
  Here I deliberately changed the order of the actual arguments to become different from the order of the formal arguments. The result is 28.0, the same.

[End of Example 5.4]

5.2.5 * args and ** kwargs

*args and **kwargs may almost look like a curse, but stands here for Python's functional arguments of a slightly special kind.

Many pre-made functions have *args (abbreviation for arguments) and / or **kwargs (abbreviation for keyword arguments) among its formal arguments. The two arguments have in common that they represent one *any number* arguments, ie it is up to the user of the functions to determine the number of arguments.

* args arguments are, in principle, just a list of individual arguments that can be in the form of values – without names. You can consider this listing as a Python list.

** kwargs are arguments where each argument is specified by name and value.

Here we will only look at an example of * args.

Example 5.5 *Functions with *args*

Program name: prog_fun_args.py.

```
import numpy as np

def fun_arg_test(k, *args):
    y = np.sum(args) * k
    return y

y = fun_arg_test(1000, 1, 2, 3)
print('y = ', y)
```

Running the program gives the following results:

```
y = 6000
```

because the sum of 1, 2 and 3 is 6, which is multiplied by 1000, giving 6000.

Comments:

- The function definition is:
  def fun_arg_test(k, *args)
  k is a “normal” argument, while * args represents all arguments (often considered a list) following the argument named factor.
- Code line:
  y = np.sum(args) * k
  Inside the function, the sum of the elements in the list, is multiplied,with k.
- Function call:
  y = fun_arg_test(1000, 1, 2, 3)
  1000 is a current argument assigned to the formal argument named k, while the sequence 1, 2, 3 is a actual argument that can be considered as a list assigned to the formal argument * args. The elements in this list are summed in the code np.sum (args).

[End of Example 5.5]

5.2.6 Documentation text (docstring)

You can enter docstring in functions that you create. Documentation text is text where you can describe the function’s structure, its input and output arguments and their data types, and what the function does. The documentation text can be displayed, among other things, using the code help (the function name), as demonstrated below.

A distinction is often made between so-called single-line and multi-line documentation.[2] Both types are demonstrated below.

One-line documentation text

In the program below I have entered a one-line documentation text within the function fun_lin.

[2]Guidelines for designing documentation text can be found in the document PEP 257på https://www.python.org/dev/peps/pep-0257.

Example 5.6 *One-line documentation text*

Program name: prog_fun_with_docstring.py

```
def fun_lin(x, a, b):
    """Calculation of the value of a linear function."""
    y = a*x + b
    return y


p0 = 2.5
p1 = 3.0
x_in = 10.0
out = fun_lin(x_in, p0, p1)
print('Result = ', out)
```

Running the program:

```
>>> %run prog_fun_with_docstring.py (or F5 or Run-button i Spyder)
Result = 28.0
>>> help(fun_lin)
Help on function fun_lin in module __main__:
fun_lin(x, a, b)
    Calculation of the value of a linear function.
```

We see that the document text is displayed in the console.

[End of Example 5.6]

Multi-line documentation text

The example below shows how multi-line documentation text can / should be written. Mark the indents and the blank line after the first line, which is assumed to be a heading for the following description.

Example 5.7 *Multi-line documentation text*

Program name: prog_fun_with_multiple_docstrings.py

```
def fun_lin(x, a, b):
    """Calculation of the value of a linear function.

    x (float) is an independent variable or input variable.
    a and b (float) are parameters.
    y (float) is a dependent variable or output variable.
    """
    y = a*x + b
    return y


p0 = 2.5
p1 = 3.0
x_in = 10.0
out = fun_lin(x_in, p0, p1)
print('Result = ', out)
```

The result of running the above program is not shown here.

[End of Example 5.7]

5.3 Namespace

When we create functions, it is important to know the term namespace. All variables belong to or exist in a namespace, but not outside the namespace. The relationship between variables and namespaces can be one of the following:

- The variable is both readable and writable (writable means the value can be changed) in the namespace.
- The variable is readable only (ie the value can be read but not changed) in the namespace.
- The variable is neither readable nor writable, ie it does not exist in the namespace.

Functions define namespaces, see Figure 5.2. (The variable names globalvar1 and localvar1 are here “random, ie they can generally have different names.)

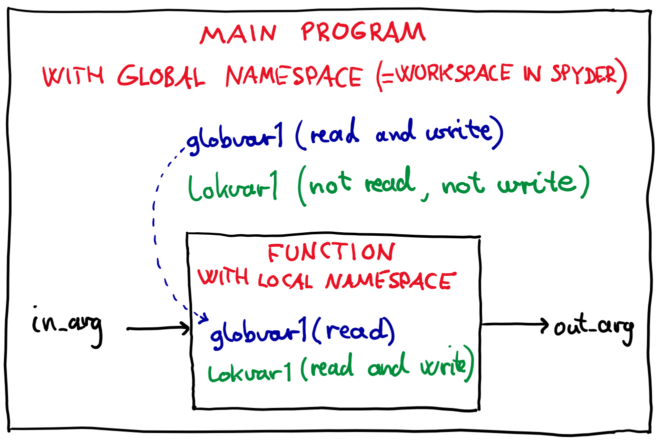

MAIN PROGRAM
WITH GLOBAL NAMESPACE (=WORKSPACE IN SPYDER)
globvar1 (read and write)
Lokvar1 (not read, not write)
FUNCTION
WITH LOCAL NAMESPACE
in_arg
globvar1 (read)
Lokvar1 (read and write)
out_arg

Figure 5.2: Navnerom

Note:

- locvar1 is readable and writable only within the function (its namespace), ie not outside. If you try to use localvar1 in a calculation in your main program, outside of the function, Python will give an (incorrect) message that the variable does not exist.
- globalvar1 is readable, but not writable, within the function. Thus, you cannot change the value of globalvar1 with code that you entered in the function.

The program in the example below demonstrates the difference between different namespaces.

Example 5.8 *Navnerom*

The program is based on the program fun_intro.py shown on page 136. print() functions are used to check the existence of the two variables x and in that belong in namespace like:

- x is a local variable in the fun_intro function, which defines the namespace of x.
- inn is a global variable in the main program, which defines the namespace to inn.

Program name: prog_fun_name_space.py

```
def fun_lin(x, a, b):
    y = a*x + b
    print('x_locvar_fun_lin =', x)
    print('in_globvar_fun_lin =', in)
    return y


p0 = 2.5
p1 = 3.0
in = 10.0
out = fun_lin(in, p0, p1)
print('in_globvar_mainprog =', in)
print('x_locvar_mainprog =', x)
```

Below are comments on this program. Try to figure out the answers to the question at each point. The solution is at the end of this example.

- The code print ('in_globvar_fun_lin =', in) inside the fun_lin function tries to print the value of the global variable named in.
  Success or failure?
- The code print ('x_locvar_fun_lin =', x) inside the fun_lin function tries to print the value of the local variable x.
  Success or failure?
- The code print ('in_globvar_mainprog =', in) in the main program tries to print the value of the global variable in.
  Success or failure?
- The code print ('x_locvar_mainprog =', x) in the main program tries to print the value of the local variable x.
  Success or failure?

Running the program:

```
>>> %run fun_navnerom.py (or F5 or Run-button in Spyder)
in_globvar_fun_lin = 10.0
x_locvar_fun_lin = 10.0
in_globvar_mainprog = 10.0
⋮
print('x_locvar_mainprog =', x)
NameError: name 'x' is not defined
```

Conclusion: Success! Success! Success! Failure!

[End of Example 5.8]

We can conclude this subchapter as follows:

- Local variables defined in functions cannot be used outside the function. If their value is to be used outside the function (in the main program), they must be passed there as a return variable from the function.
- Global variables are directly available for reading (not writing) within functions. But it is considered rather poor program technique to read global variables inside functions, since the function's interface with the main program is then not well defined. The interface is well defined if all data exchange with the main program takes place via arguments in the function shell, so this is the recommended way to transfer data.

5.4 Programming of modules

One module is a script that contains functions and / or variables. You can use these functions and variables in a main program after importing the module into the program. Figure 5.3 illustrates the relationship between a module and main applications.

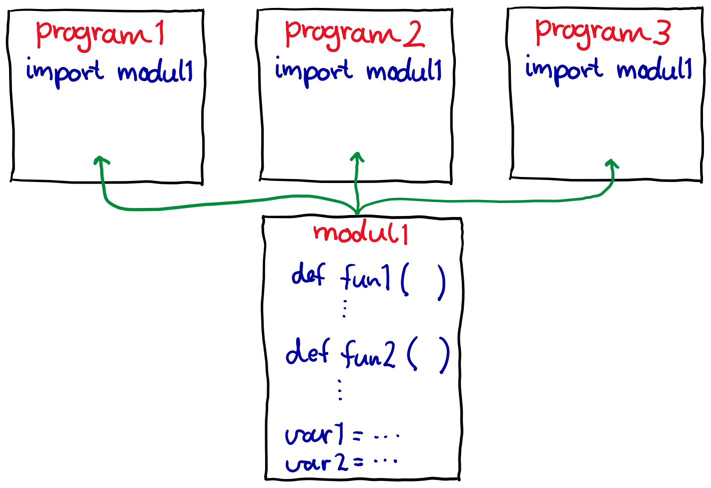


Figure 5.3: The relationship between a module with its functions and/or variables and main programs where the module is imported

What benefit can you have of modules?

Probably the most important use of modules is that you (in modules) can collect functions and variables that you use in several programs – ie "multi-use functions" and "multi-use variables" – thus avoiding having a copy of them in each of the programs. Therefore, you can have better control over your functions and/or variables if they are to be reused.

Import of modules to the (main) program

The import command is used to import a module.

It may be convenient to rename the module to a short name when importing the module, and using the short name in the program, just as when the numpy package is renamed to the short name np when importing numpy (as we have seen in many examples earlier in the book).

Example: The module my_module can be renamed to mm using the following code in the program:

```
import min_modul as mm
```

When using imported functions and variables in the program, the module name must be included in this way (the names function1 and var1 are random here):

mm.funksjon1

and

mm.var1

Example 5.9 *Import of our own module*

This example works the same way as the prog_fun_intro.py program, see page 136, but we will now define our own developed function fun_lin() in a module and not directly in the program. The variables p0 and p1 must also be defined in this module. The function and the two variables are made available in the program by importing the module.

The main program is shown in the box below.

Program name: prog_module.py.

```
import my_module as mm


x_in = 10.0
out = mm.fun_lin(x_in, mm.p0, mm.p1)
print('Result = ', out)
```

The module is shown in the box below.

Module name: my_module.py.

```
def fun_lin(x, a, b):
    y = a*x + b
    return y


p0 = 2.5
p1 = 3.0
```

The result of running the main program is:

```
Reloaded modules: my_modul
Result = 28.0
```

[End of Example 5.4]

5.5 Lambda functions

Lambda functions is a special type of functions in Python that can be used to solve simple calculations. Some important characteristics:

- They are not defined with the def command.
- They have no local variables.
- They are usually written on one line of code.

Lambda functions can be used as a kind of calculator based on a formula. Lambda functions are not assigned a name, which is why they are also called anonymous functions.

The syntax is shown in the following example where we create a lambda function to calculate

$$y = a*x + b$$

where x, a and b are arguments, and y is arguments. (The Lambda function in the example below does the same job as the regular function that we developed in Chap. 5.2.1.)

Example 5.10 *Lambda functions*

```
y = lambda x, a, b: a * x + b
out = y(10, 2.5, 3)
print('out = ', out)
```

We see that the Lambda function is defined in only one line:

$$y = \text{lambda } x, a, b:\ a * x + b$$

there:

- The input arguments are x, a and b, which are listed, separated by commas.
- The function (formula) is a * x + b that uses the arguments.

- The output argument is y, which is calculated by the function (formula).

The actual call of the lambda function is

out = y(10, 2.5, 3)

Running the program gives the result:

```
out = 28.0
```

[End of Example 5.10]

Chapter 6

Testing your own code

6.1 Introduction

Don't trust yourself completely. The programs you develop will probably have errors.

There are two main types of errors:

- *Syntax Error*, which are purely "technical" errors, e.g. that you have forgotten a parenthesis or entered a function name incorrectly or the like. The programming environment that you are using will indicate syntax errors in some way. For example, Spyder provides good information about syntax errors, cf. 2.2.4.
- *Functional errors*, which are errors in the program logic or algorithms. Functional errors can be very difficult to find!

In this chapter, we will concentrate on testing with the aim of finding any functional errors.

Who should test?

- You (the programmer)?
- A test user?
- The end user, who expects to get a program that works correctly?

The end user is excluded. Admittedly, it is good to hear about any bugs in the program, but the end user should not test, but instead rely on the program to function properly.

It is very good if you have a test user at your disposal. It is useful that others than yourself try the program you have developed. A potential end user can also act as a test user in a test phase.

Whether or not you get help from others for testing:

You must test the program yourself!

6.2 How to test for functional errors?

A program consists of a main program that uses one or more proprietary sub-programs in the form of functions or modules, see Figure 6.1.

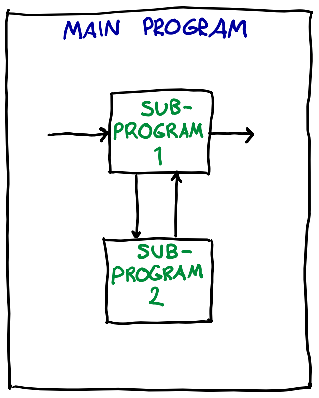


Figure 6.1: Main program with sub-programs with their inputs and outputs

You must test:

- Sub-program 1
- Sub-program 2
- Main program – after testing the sub-programs

How do you test the programs? Here's a recipe:

1. Assume a typical (normal) value of the sub-program's inputs (input arguments). Run the program with these values. Does the program's initial value correspond to an analytical value, which is a value you can calculate "on paper"?

2. Repeat step 1, but with a representative number of other input values.

3. Assume abnormal values of the input values, which may be values that you assume the user entered incorrectly, e.g. a negative number as an input to a function where you have actually assumed only positive input values. Notice how the program responds to these abnormal input values. Does the program stop? Does Python give an error message? Should you limit the range for possible input values? Should you program alerts for the user?

Testing must be documented, e.g. based on table 6.2. The table applies to a specific example, which is presented below.

What	**How**	**Who**	**When**	**Result**	**Comm.**
fun_losn_2grads_likn()	Uses different sets of coefficient values (a, b, c). Must yield the same result as analytical solution. The function must also handle negative discriminant.	FH	15.6 2019	Ok	-

Table 6.1: Documentation of program testing

In the example that follows, we will program a function for solving 2nd order equations. Note: There is something wrong with the program (that is what the purpose of the example is :-).

Example 6.1 *Testing of your own function*

We assume a general 2nd order equation:

$$ax^2 + bx + c = 0 \tag{6.1}$$

I think that the two solutions are[1]

$$x_1 = \frac{b + \sqrt{b^2 - 4ac}}{2a} \tag{6.2}$$

and

$$x_2 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{6.3}$$

and that any concurrent solution yields discriminant equal to zero:

$$D = b^2 - 4ac = 0 \tag{6.4}$$

The program is shown below.

Program name:

```
import numpy as np


def fun_losn_2grads_likn(a, b, c):
x1 = (b + np.sqrt(b*b - 4*a*c))/(2*a)
x2 = (-b + np.sqrt(b*b - 4*a*c))/(2*a)
return (x1, x2)


a = 1.0
b = 0.0
c = -4.0
(x1, x2) = fun_losn_2grads_likn(a, b, c)
print('x1 =', x1)
print('x2= ', x2)
```

The result is:

```
x1 = 2.0
x2= 2.0
```

The fact that the solutions are "something with 2" seems reasonable based on the given coefficient values, so it is tempting to conclude that the program gives the correct answer.

[1];-)

But, let's test the program:

Test 1

Let's put the solutions into the given equation and see if the right side value really becomes zero, which is specified. For such a calculation we can create a program that calculates the value of the right-hand side, but here it is easier to do the hand calculation. The two solutions are the same, so we set $x_1 = x_2 = 2.0$ in for x i (6.1) and get

$$1 \cdot 2.0^2 + 0 \cdot 2.0 - 4.0 = 0 \tag{6.5}$$

that's right! Thus, 2.0 is a solution. So far, nothing indicates that something is wrong. But you may remember that with two concurrent solutions, the discriminator equals zero.

Test 2

We calculate the discriminant, D:

$$D = b^2 - 4ac = 0^2 + 4 \cdot 1 \cdot (-4) = -16 \tag{6.6}$$

which is different from zero!

Since the program failed both tests, we must conclude that the program has one or more errors.

We therefore review the program code once again. The code is correct in that it implements the solution formulas (6.2) and (6.3). So then there must be something wrong with one or both of the solution formulas. There are actually errors in both formulas. The correct ones are

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{6.7}$$

and

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{6.8}$$

Once we have fixed the errors, we run the tests again. (The result of this is not shown here.)

Test 3

In the testing above, we used examples of coefficient values (a, b, c) which did not create problems in relation to the sign of the discriminant, D, ie D

was not negative. But let's assume that it is conceivable that the user of the program will enter values that give negative D. For negative D there are no real solutions of the 2nd degree equation; The solutions are complex numbers. How our program responds to the negative D? Let's try

$$a = 1,\ b = 0,\ c = 4$$

The result is Python for Python (nan stands for not-a-number):

```
x1 = nan
x2 = nan
C:/techteach.no/publications/python/scripts/prog_losn_2grads_likn.py:14:
RuntimeWarning: invalid value encountered in sqrt x1 = (-b + np.sqrt(b*b
- 4*a*c))/(2*a)
C:/techteach.no/publications/python/scripts/prog_losn_2grads_likn.py:15:
RuntimeWarning: invalid value encountered in sqrt x2 = (-b - np.sqrt(b*b
- 4*a*c))/(2*a)
```

It appears that the np.sqrt() function is unable to calculate complex solutions. That's a bit stubborn, I think, but we have to accept it, of course. Then we have two options:

1. Find a function, as an alternative to np.sqrt(), that is capable of calculating complex solutions.
2. Continue using np.sqrt() in the program, but notify the user that there are no real solutions to the specified coefficient values if the discriminator is negative.

I stop the example her, but you may try the above two options.

[End of Example 6.1]

6.3 How to run only part of the program?

Sometimes we want to run only part of the program code to see if it works. In Spyder, it can be done as follows:

1. Select the current program section.
2. Run the program section by making one of the following choices:

(a) The Run / Run selection or current line menu option

(b) Menu button Run / Run selection or current line

(c) F9 key on the keyboard

Of course, it is necessary that variables included in the program part to be run are already defined, ie are in the workspace.

Chapter 7

Conditional program execution with if-structures

7.1 if-else

With an if-program structure we can control the program execution, see Figure 7.1. If the if condition has a Boolean value of True, the program part is executed in the if branch. I have called this code section for CODE_IF. Otherwise, ie if the if condition has a Boolean value of False, the program part is executed in the else branch, ie the code in CODE_ELSE. Of course, it is up to us to enter Python code into these two alternative code sections.

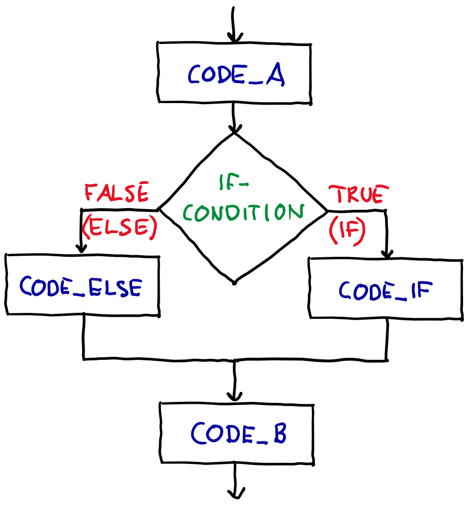


Figure 7.1: If-else structure

The program execution is therefore governed by the boolean (logical) value of the if condition. We construct the if condition with logical operators and comparison operators, cf. 3.8.

The program code below demonstrates programming with the if structure. The program has an if branch and an else branch. The comparison expression (x> y), where x and y are numbers, determines which branch is run, cf. the detailed comments below.

Program name: prog_if_else.py

```
x = 10
y = 0
if (x > y):
    print('(x > y) =', (x > y))
    print('The If branch is active.')
else:
    print('(x > y) =', (x > y))
    print('The Else branch is active.')

print('This is the first code line after the if structure.')
```

As demonstrated in the program above:

- The code lines according to the if condition must be indented. The same applies to the code lines according to the else condition. The

Python standard is indented with 4 spaces.

- There should (not must) be a blank line between the last code line in the if structure and subsequent code lines.

The box below shows the result of the run with the values x = 10 and y = 0. The comparison expression (x> y) then has a Boolean value True, which means that the if branch is active. The code that is then executed, ie CODE_IF in Figure 7.1, are the following two calls of the print() function:

- The code print ('(x> y) =', (x> y)) which prints the text '(x> y) =' and the boolean value of the comparison expression (x> y).
- The code print ('if branch is active.').

```
(x > y) = True
if branch is active.
This is the first code line after the if structure.
```

The box below shows the result of the run with the values x = –10 and y = 0. The expression (x> y) then has a Boolean value False, which means that the else branch is active. The code being executed, ie CODE_ELSE in Figure 7.1, are two calls of the print() function (not reproduced here).

```
(x > y) = False
else branch is active.
This is the first code line after the if structure.
```

7.2 Without else

The if structure doesn't really need to include anything else, see Figure 7.2. If the if condition has the value False, the program skips all code contained within the if structure.

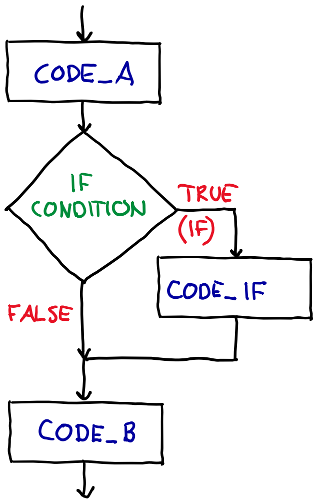


Figure 7.2: Program flowchart for an if structure without else

The program code below demonstrates programming an if structure without else.

Program name: prog_if.py.

```
x = 10
y = 0
if (x > y):
    print('(x > y) =', (x > y))
    print('if branch is active.')

print('This is the first code line after the if structure.')
```

The box below shows the result of the run with the values x = 10 and y = 0. The comparison expression (x> y) then has a Boolean value of True, which means that the if branch is active.

```
(x > y) = True
if branch is active.
This is the first code line after the if structure.
```

The box below shows the result of the run with the values x = –10 and y = 0. The comparison expression (x> y) then has a Boolean value False, which means that the if branch is inactive. And since there is no other branch here, the program goes straight to the code line after the if structure, which is print ('This is the first code line after the if structure.').

```
This is the first code line after the if structure.
```

7.3 elif

And so we have elif. Both elif and else are used to define code that is run if the if condition is False. But elif is narrower than else. You can specify your own elif condition (while you do not specify your own else condition). This is demonstrated in the program below.

Program name: prog_elif.py.

```
x = 0
y = 0
if (x > y):
    print('if branch is active.')
elif (x == y):
    print('elif branch is active.')
else:
    print('else branch is active.')

print('This is the first code line after the if structure.')
```

The box below shows the result of the run, where x = 0 and y = 0, which means that the elif condition (x == y) is True, which makes the elif branch active.

```
elif branch is active.
This is the first code line after the if structure.
```

Chapter 8

Iterated program runs with for loops and while loops

8.1 Introduction

Loops are very useful program structures. With loops, you can iterate (repeat) program code as many times as you like – automatically!

Loops are used in a variety of contexts, including:

- *Mathematical operations* on arrays and lists, etc. where the loop iterates the operations on the elements
- *Solving equations* with an iterative method (until the solution is found with sufficient accuracy)
- *Optimization* (find the maximum or minimum of a function) with an iterative method (until the optimal value is found with sufficient accuracy)
- *Simulation* where the mathematical model equations are solved for each time in the simulation interval

The present version of this book does not cover the above applications. I assume that you are to work on such applications in other contexts (courses or projects). Here and now my aim is only to describe looping techniques in Python.

In this chapter we will take a closer look at the two main types of loops in Python:

- For loops, which are loops that iterate a predetermined number of times, e.g. as many times as there are elements in a given array on which the loop should operate. (Chap. 8.2.)
- While loops, which are loops that iterate as long as a continue condition is satisfied. In principle, the number of iterations is in while loops *not known* in advance, which is a significant difference from for-loops where the number of iterations is determined before starting the loop. (Chap. 8.3.)

Figure 8.1 illustrates for-loops and while-loops with program flow charts. The for loop runs a predetermined number (N) times.

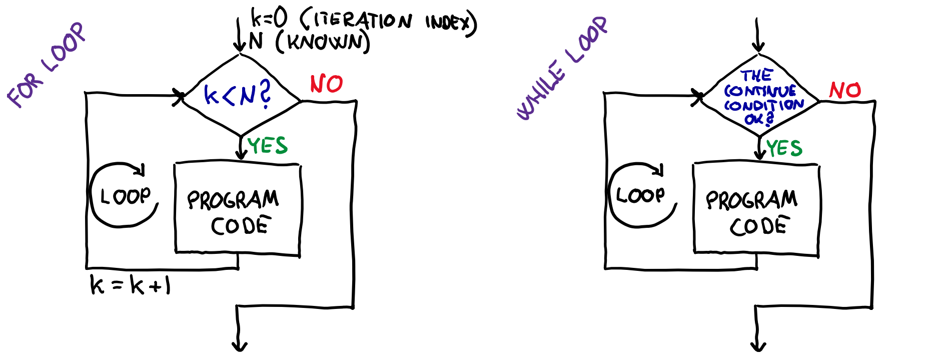


Figure 8.1: Program loop charts for for loop and while loop: Program loops are used to iterate program code execution.

8.2 For loops

8.2.1 Basic programming of for loops

The program code below demonstrates basic programming of for-loops.

Example 8.1 *For loops*

Symbol use in the program code:

- k is the iteration index of the loop. k is 0 at the first iteration of the loop.
- E represents the current element in the array.

Program name: prog_for_loop_read.py

```
import numpy as np



A = np.array([5, 10, 15])
k = 0

for E in A:
    print('k =', k)
    k = k + 1
    print('E =', E)
    print('-----------------')

print('This is the first code line executed after the loop has stopped.')
```

The result of the run:

```
k = 0
E = 5
-----------------
k = 1
E = 10
-----------------
k = 2
E = 15
-----------------
This is the first code line executed after the loop has stopped.
```

Comments:

- The for loop iterates as many times as there are elements in the array A, starting with the first element in the array.
- For each iteration of the loop, we can access the relevant element (E) in the array (A). In this example, I use the print() function to display the value of E in the console.
- At each iteration of the for loop, the loop index k is equal to the value of the number of times the for loop has iterated.

A technicality: I implemented the update of k at each iteration with the code: k = k + 1, which increases the previous value of k by 1. Alternative code is: k + = 1.

- The code inside the for loop must be indented (python: indented). It is a Python standard with 4 space indents.[1]
- The code line print ('-----------------') is for cosmetic reasons only.
- The very last line of code, ie the code line print ("This is the first code line executed after the loop has stopped."), is here displayed to demonstrate that there should (not must) be a blank line between the last line of code in the for loop and subsequent code lines.
- In this example, I have used array (A) as the so-called iteration element, but also other data types consisting of a sequence or series of data can be used, as lists and tuples.

[End of Example 8.1]

8.2.2 How to write to array elements in a for loop

In the program example above, the elements of an array are *read* as the for loop iterates. We can also *write* values to array elements, as shown in the example below, to write values to an array that has 10 elements.

Example 8.2 *For loop*

Note: In this example, the array to which values are to be written is created *in advance*, ie before the for-loop starts. This is so-called preallocation of the array. Preallocation is a good programming technique since it can make the program spend far less time running than without preallocation. Running times with and without preallocation are shown in the example 8.3.

Program name: prog_for_loop_write.py

[1]In other program languages, it is common to use parentheses to mark the start and end of the program code inside loops, but in Python, indentation is used. Indentation helps to make the program appear structured and clearly cosmetic (but the logic of the program may just as well be unclear).

```
import numpy as np

A = np.zeros(10)
N = len(A)
k = 0

for k in range(0, N):
    A[k] = 10*k

print('A =', A)
```

The result of the run is the following array, which has 10 elements:

```
A = [ 0. 10. 20. 30. 40. 50. 60. 70. 80. 90.]
```

Comments:

- The code A = np.zeros (10) creates an array A with 10 elements, each of which has a value of 0. A value is to be written to each of the elements in this array when the for loop iterates. That the array is created before the for-loop starts is called the preallocation of the array.
- The code N = len (A) gives N value equal to the number of elements in the array A.
- The code k = 0 sets the loop index to zero before the loop starts.
- The code for k in range (0, N) causes the loop to iterate N times. Note: k gets the values 0, 1, ..., 8, 9 as the loop iterates. Thus, the last value of k is N-1 = 9, and not N = 10.
- At iteration k, the value 10 * k is written to the array element A [k].
- The code k = k + 1 increases the loop index k by value 1. Alternatively, we could have written k + = 1.
- The code print ('A =', A) shows the resulting value of the array A in the console.

[End of Example 8.2]

8.2.3 Preallocation of arrays to save execution time

As mentioned, it can be a lot of time to save on preallocating arrays to write values to iteratively (in a loop). Figure 8.2 illustrates two principles for iterative data writing in an array:

- Continuous expansion of an array with the np.append() function
- Writing to a preallocated array.

Python spends relatively much time expanding an existing array of new elements. That is why we should have as few such operations as possible in our programs. We can save a lot of execution time by creating the array with the sufficient number of elements in advance, perhaps with a value of 0 in all the elements,and then write values to the elements in that array (in the loop). This is *preallocation* of the array.

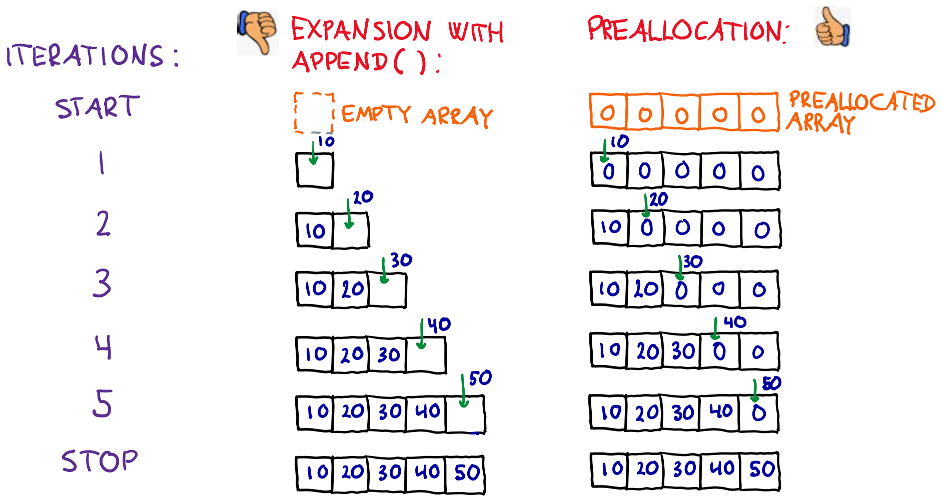


Figure 8.2: Illustration of two principles for iterative writing of data in array: (1) Continuous expansion of an array with np.append() function and (2) writing to the preallocated array.

Here is an example that demonstrates the time savings that preallocation can provide.

Example 8.3 *The importance of preallocation*

Some comments on the program shown below:

- The program writes values to two arrays with 100,000 elements. The value written to element # k is 10 * k.
- For one array, called dyn_array, values are written using. np.append() - function. dyn_array is not preallocated and is expanded with the append() function for each iteration of the for-loop.
- For the second array, called preallok_array, values are written at each iteration through accessing the current element of the array. preallok_array is preallocated, ie created before the for loop starts.
- The time to run the relevant program sections is measured using. function time() in the time packet, which is imported with the code import time at the top of the program.
- At the end of the program, the resulting running times without preallocation and with preallocation are compared.

Here is the program:

Program name: prog_time_preallocation_array.py

```
# ----------------------------------------------------------
# Import:

import numpy as np
import time

# ----------------------------------------------------------
# Initialisering:

N = 100000
dyn_array = np.array([])
preallok_array = np.zeros(N)
# ----------------------------------------------------------
# append:

tic = time.time()
for k in range(0, N):
dyn_array = np.append(dyn_array, 10*k)
toc = time.time()
t_elapsed_append = toc - tic
# ----------------------------------------------------------
# preallok:

tic = time.time()
for k in range(0, N):
preallok_array[k] = 10*k
toc = time.time()
t_elapsed_preallok = toc - tic
# ----------------------------------------------------------
# Sammenlikning:

ratio_append_preallok = t_elapsed_append/t_elapsed_preallok
print('-----------------------------------------------')
print('N =', N)
print('t_elapsed_append [s] = ',t_elapsed_append)
print('t_elapsed_preallok [s] =', t_elapsed_preallok)
print('t_elapsed_append/t_elapsed_preallok =', ratio_append_preallok)
print('-----------------------------------------------')
```

Result of a run (rounded numbers):

```
N = 100000
t_elapsed_append [s] = 4.982
t_elapsed_preallok [s] = 0.03000
t_elapsed_append/t_elapsed_preallok = 166.1
```

Comments:

- The preallocation program of the array to which data is written runs approximately. 166 times less time than without preallocation (with np.append() function). Actually, there may be a lot of *real time* to save: Suppose that in another case, with the ratio of run time as in our original example, it takes 1 minute to run the program with preallocation. It will then take 166 minutes, ie not far from 3 hours, to run the program without preallocation. Assume you need the result of the loop in some subsequent task... You can draw the conclusion about selecting between preallocation and no preallocation yourself.
- Of course, the ratio 166 applies only to this test, but I have seen significant time savings in other examples as well.

[End of Example 8.3]

8.3 While loops

While loops are loops that iterate as long as a given continue condition is satisfied. Figure 8.1 illustrates while loop with a program flowchart.

In principle, the number of iterations in while loops is not known in advance. In Chap. 8.2.3 we saw that the preallocation of arrays can greatly save the run time of a program where values are written to the array inside the for loop. Since the number of iterations in while loops is not known in advance, it may be more difficult to implement the preallocation of arrays in the context of while loops. But preallocation can still be realized if one knows an upper limit on the number of times the loop will be iterated. After all, it's only possible to preallocate an array of length equal to this greatest number of times.

The continue condition shown in Figure 8.1 is, in the python-technical sense, a logical expression that has either the value True or False:

- If the value is True, the while loop continues to iterate.

- If the value is False, the iteration stops and Python continues to run the program code after the while loop.

We construct the continue condition with logical operators and comparison operators, cf. 3.8.

The program code below demonstrates basic programming of while loops.

Program name: prog_while_loop.py

```
k = 0
while (k < 5):
    print(k)
    k = k + 1 #Alternatively: k += 1

print('This is the first code line executed after the loop has stopped.')
```

The result of the run:

```
0
1
2
3
4
This is the first code line executed after the loop has stopped.
```

Comments:

- The while loop is iterated as long as the continue condition (k <5) is satisfied.
- In each iteration, the following occurs:
  - The value of k is written to the console with the print() command.
  - The value of k is increased by 1 with the code k = k + 1, which can alternatively be written k + = 1.
- When k becomes 5, the continue condition is not satisfied and the loop stops. The program then goes to the program code that appears after the (below) while loop, ie the code: print('This is the first code line executed after the loop has stopped.').
- In the continue condition, the comparison operator < is used, cf.

Table 3.4. You can drop the parentheses around k <5.[2]

- The code inside the while loop must be indented (python: indented). It is a Python standard (the PEP 8 guidelines) with 4 space indents (as for the for loops).
- The very last line of code, ie the code line `print ("This is the first code line executed after the loop has stopped.")`, is here displayed to demonstrate that there should (not must) be a blank line between the last line of code in the while loop and subsequent code lines.

[2]I myself prefer to include this unnecessary parenthesis because the code is then easier to read.

Chapter 9

Write and read file data

9.1 Introduction

Suppose you have useful numerical data from a calculation or a simulation or an experiment. You may want to write the data to (save the data on) a file so that you can read it later for plotting or analysis or data manipulation, or send it to someone else who can use the data.

In this chapter we will see how we can write and read data files.

9.2 File Formats

The numeric data can be stored on file in two alternative formats:

- Textual data files
- Binary data files

Let's take a closer look at these two file formats.

9.2.1 Textual data files

Textual data files are files where the data is in the form of numbers which are strictly text and which you can therefore read yourself, eg. 1.2 and -18.57 with decimal point or 1.2 and -18.57 with decimal point. In this context, the numeric characters are text characters! You can open text files – even if they contain only numeric characters – with word processors such as Word and Notepad and spreadsheet programs such as Excel or similar OpenOffice programs, etc. and literally (!) see the data there.

Example 9.1 *Textual data file from an experiment on an air heater process*

Figure 9.1 shows a laboratory model of an air heater process[1].

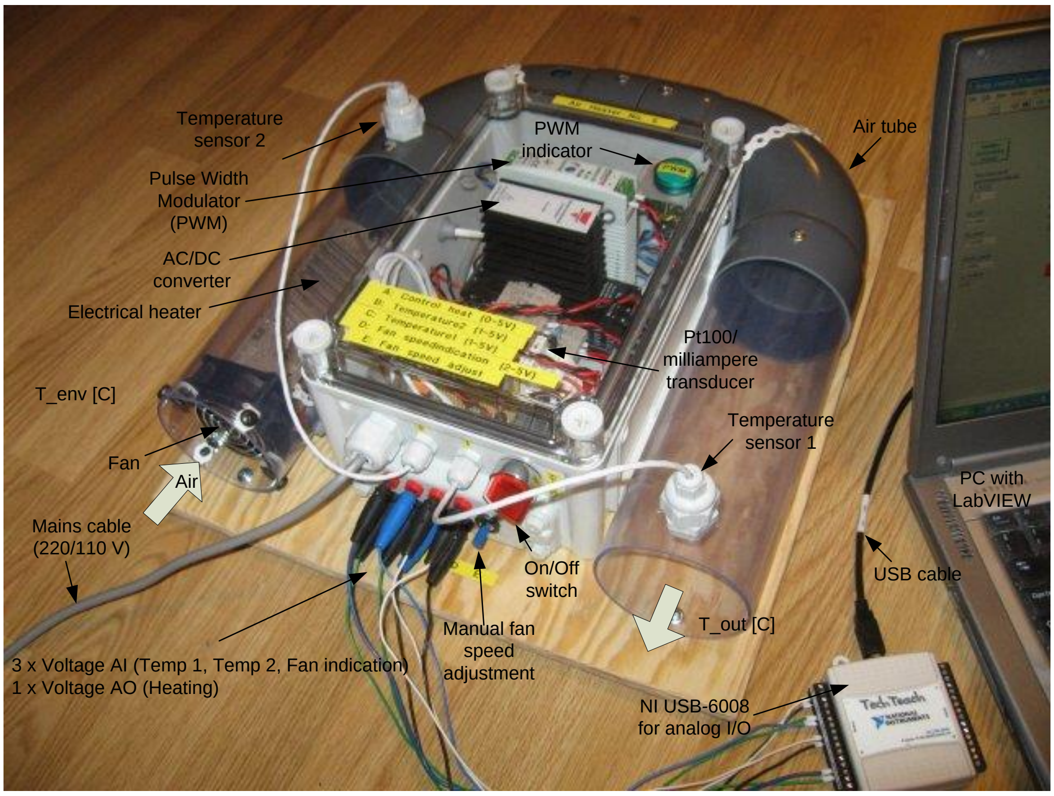


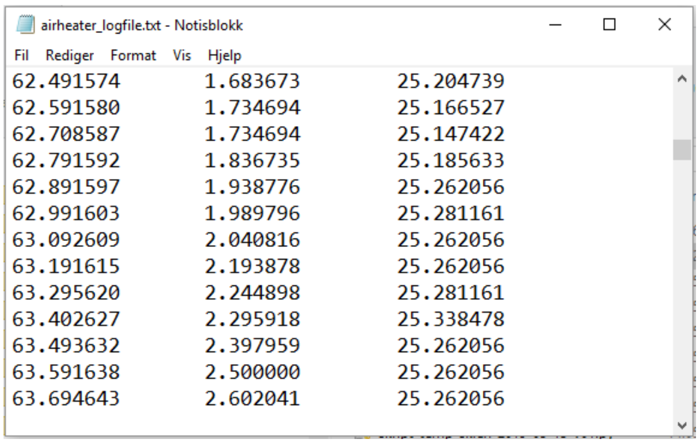
Figure 9.1: Air heater

Figure 9.2 shows a textuial data file, opened in Notepad, with data from an experiment on the air heater process. The data file has the following three columns of data (from left):

- Time stamps [s]

[1]http://home.usn.no/finnh/air_heater

- Control signal to the heating element ("Electrical heater" in Figure 9.2) [V]
- Temperature measurement (from "Temperature sensor 1" in Figure 9.2) [°C]


airheater_logfile.txt - Notisblokk
Fil Rediger Format Vis Hjelp

```
62.491574       1.683673        25.204739
62.591580       1.734694        25.166527
62.708587       1.734694        25.147422
62.791592       1.836735        25.185633
62.891597       1.938776        25.262056
62.991603       1.989796        25.281161
63.092609       2.040816        25.262056
63.191615       2.193878        25.262056
63.295620       2.244898        25.281161
63.402627       2.295918        25.338478
63.493632       2.397959        25.262056
63.591638       2.500000        25.262056
63.694643       2.602041        25.262056
```

Figure 9.2: Test-based data file opened in Notepad (Notepad)

[End of Example 9.1]

It is common to represent data files in the form of text files because:

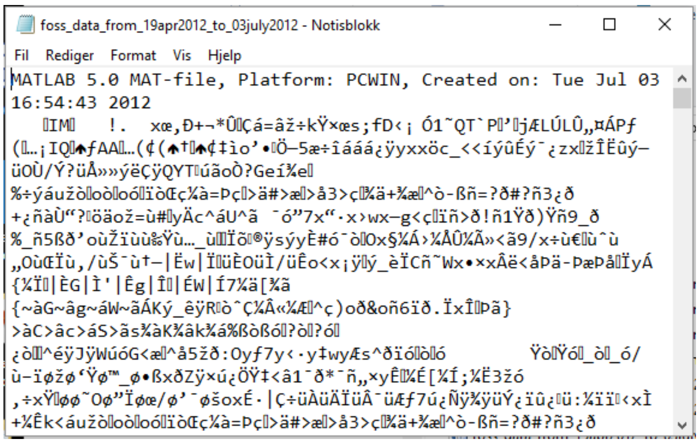
- You can see the numbers (as number signs), so you can check your data visually.
- You can assume that any special tools for analyzing and plotting and manipulating data can handle textuial data files. (They will also be able to handle data files that have some special format, which will then be a binary file, see below.) Examples of such special tools are MATLAB, LabVIEW, Octave and Excel. Text files are thus a flexible data format.

In this chapter we will take a closer look at how we can write data to and read data from text files, but let's first take a look at binary files.

9.2.2 Binary data files

Binary data files – or just binary files – can contain number data, as text (data) files do, but in binary files, the data is represented in a more compact form than in text files and in some code that people cannot read. Only special tools (programs) can read the information in binary files.

Figure 9.3 shows an example of a binary file, namely a mat file (generated in the MATLAB software) opened in Notepad. The data in the file are measured values from a biogas reactor: temperature, biogas flow, methane concentration, etc. Except for information about the file itself at the top of the file, the content is not readable to us. But the content is fully readable in MATLAB.



```
MATLAB 5.0 MAT-file, Platform: PCWIN, Created on: Tue Jul 03
16:54:43 2012
   IM    !.  xœ,Ð+¬*ÛÇá=âž÷kŸ×œs;ƒD‹¡ Ó1˜QT`PÌ’ĳÆLÚLÛ„¤ÁPƒ
(…¡IQ ƒAA…(¢(†¢‡ìo’•Ö–5æ÷îáâá¿ÿyxxöc_<<íýûÉý¯¿zxžÎÈûý–
üOÙ/Ý?üÅ»»ýëÇÿQYTúãoÒ?GeÍ‰e
%÷ýáužòòoóìòŒç¼à=Þç>ä#>æ>å3>ç‰ä+‰æ^ò-ßñ=?ð#?ñ3¿ð
+¿ñàÙ“?öäož=ù#lyÄc^áU^ã ¯ó”7x“·x›wx–g‹çìñ>ð!ñ1Ÿð)Ÿñ9_ð
%_ñ5ßð’oùŽïùù‰Ÿù…_ùÏõ®ÿsýyÈ#ó¯òOx§¼Á›¼ÂÛ¼Ã»<ã9/x÷ù€ù^ù
„OùŒÏù,/ùŠ¯ù†–|Ëw|ÏüÈOüÏ/üÊo<x¡ÿlý_èÏCñ˜Wx•×xÂë<åÞä-ÞæÞåÏyÁ
{¼Ï|ÈG|Ì'|Êg|Î|Éw|Í7¼ä[¼ã
{~àG~âg~áW~ãÁKý_êÿRò^Ç¼Â«¼Æ^ç)oð&oñ6ïð.ÏxÎÞã}
>àC>âc>áS>ãs¾àK¾âk¾á%ßòßó?ò?ó
¿ò^éÿJÿWúóG<æ^â5žð:Oyƒ7y‹·y‡wyÆs^ðïóòó        ŸòŸó_ò_ó/
ù–ïøžø‘Ÿø™_ø•ßxðZÿ×ú¿ÖŸ‡<â1¯ð*¯ñ„×yÊÊ¼É[¼Í;¼Ë3žó
,÷xŸøø˜Oø”Ïøœ/ø’¯øšoxÉ·|Ç÷üÀüÄÏüÀ¯üÆƒ7ú¿Ñÿ¾ÿüÝ¿ïü¿ü:¼ïï‹xÌ
+¼Êk‹áužòòoóìòŒç¼à=Þç>ä#>æ>å3>ç‰ä+‰æ^ò-ßñ=?ð#?ñ3¿ð
```

Figure 9.3: A binary file opened in Notepad

We can store data in the form of binary files in Python, but I will not go into this further since text data files are far more convenient, even though they are typically larger than binary files with the same information content.

9.3 Write data to file

The open() function available in Python's standard package, see Ch. 2.6.2, can be used to write data of different types to file. open() is thus a general function. This book focuses on using Python for calculations, and the data is then typical numbers, most often floating numbers. The Numpy library has the savetxt() function tailored to our purpose: "Save an array to a text file", cf. documentation on

https://docs.scipy.org/doc/numpy/reference/generated/numpy.savetxt.html

It seems easier – at least to me – using savetxt() rather than open() to write numeric data to text file. I therefore concentrate on savetxt() in this chapter.

When using savetxt() in a program, we need to type np.savetxt() if we have imported the numpy package with the command

import numpy as np

np.savetxt() has a number of arguments, but for many of the arguments the default values will be ok, so we do not need to give explicit values to these arguments. I guess the most current use of np.savetxt() is:

np.savetxt(fname, X, fmt='%.18e', delimiter=' ')

Here is a description of the individual arguments:

- fname (file name) is the name you give the file, quoted in quotation marks, e.g. 'Eksperiment1.txt'.
- X represents the name of the variable to be stored, e.g. sensordata1. The variable must have data type array, either 1D array or 2D array.
- fmt (format) determines the format by which the numbers are stored in the file. The default value is '% .18e', which means that the numbers are stored in exponential form by 18 digits after the decimal point, e.g. 1.123456789123456789e + 02 where e + 02 means 10 ˆ {2}. The number before the decimal point will be an integer in the range 1 - 9. It is possibly overkill to have 18 digits after the decimal point. A proposal is 3, which means that 4 digits determine the number in addition to the 10's exponent.
- delimiter = ” (note: there are spaces between quotes) indicates that spaces are used to separate the columns in the file. Space is thus the default value. You may use a comma, and then use delimiter = ','.

Here's an example where we're going to create a 2D array of columns equal to two given 1D arrays and then write the 2D array to file.

Example 9.2 *Writing 2D array to np.savetxt file()*

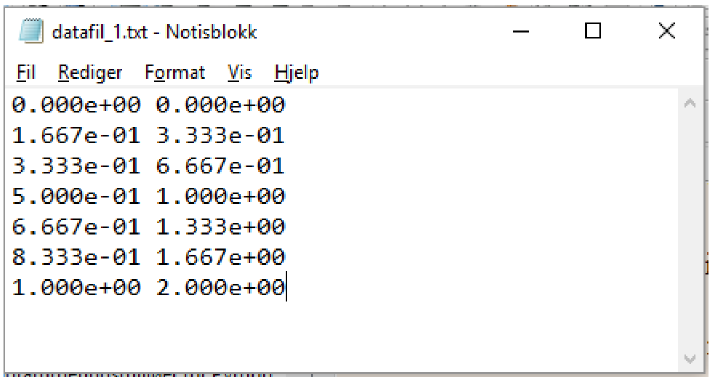
I hope the code in the box below is self-explanatory.

Program name: prog_savetxt.py

Note that the two 1D arrays turn into columns in the 2D array, which is realized with the transpose method (T) of the np.array array object ([A1, A2]), cf. Example 3.33 on page 96.

```
import numpy as np


A1 = np.linspace(0, 1, 7)
A2 = np.linspace(0, 2, 7)
M = np.array([A1, A2]).T
np.savetxt('datafil_1.txt', M, fmt='%.3e', delimiter=' ')
```

Figure 9.4 shows the file datafil_1.txt opened in the Notepad program.



```
0.000e+00 0.000e+00
1.667e-01 3.333e-01
3.333e-01 6.667e-01
5.000e-01 1.000e+00
6.667e-01 1.333e+00
8.333e-01 1.667e+00
1.000e+00 2.000e+00
```

Figure 9.4: Datafile datafil_1.txt opened in Notepad

[End of Example 9.2]

9.4 Read data from file

The function open() in Python's standard package, see chap. 2.6.2, can be used to read data from files, but for the same reason as for writing data to file, cf. Section 9.3, I recommend using instead the loadtxt() Numpy function, which reads the numeric data in a text file and save it as an array in the Python workspace.

Documentation of loadtxt() is available at https://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html

When we use loadtxt() in a program, we must type np.loadtxt() if we have imported the numpy package with the command

```
import numpy as np
```

np.loadtxt() has a number of arguments, but for many of the arguments, the default values will be ok, so we do not need to give explicit values to these arguments. I guess the most current use of np.loadtxt() is:

```
np.loadtxt(fname, delimiter=' ')
```

Here is a description of the individual arguments:

- fname (file name) is the name you give the file, quoted in quotation marks, e.g. 'Eksperiment1.txt'.
- delimiter = '' (note: there are spaces between quotes) indicates that spaces are used to separate the columns in the file. Space is thus the default value. If instead a comma is used to separate the columns in the file, use delimiter = ','.

Here is an example of using np.loadtxt().

Example 9.3 *Read 2D array from file with np.loadtxt()*

The program below reads the number data stored in datafil_1.txt, which we created in the example 9.2, and returns the data as a 2D array here called M. Two 1D arrays A1 and A2 are extracted from respective columns in M.

```
import numpy as np


M = np.loadtxt('datafil_1.txt', delimiter=' ')
A1 = M[:, 0]
A2 = M[:, 1]

print('M =', M)
print('A1 =', A1)
print('A2 =', A2)
```

The result of running the program is:

```
M = [[0.  0.  ]
 [0.1667 0.3333]
 [0.3333 0.6667]
 [0.5 1.  ]
 [0.6667 1.333 ]
 [0.8333 1.667 ]
 [1.  2.  ]]
A1 = [0.  0.1667 0.3333 0.5 0.6667 0.8333 1.  ]
A2 = [0.  0.3333 0.6667 1.   1.333   1.667   2.  ]
```

[End of Example 9.3]

Bibliography

Haugen, F. A. (2019), *Automatic Control*, 'home.usn.no/finnh/books'.

Langtangen, H. P. (2016), *A Primer on Scientific Programming with Python*, Springer.

Linge, S. & Langtangen, H. P. (2016), *Programming for Computations - Python*, Springer (Open Access).

Index