# Kapitel 3

# Programmieren in Python I

## 3.1 Wahl der Programmiersprache

Es existieren dutzende bekannte Programmiersprachen [1]. Da die meisten Programmiersprachen jedoch viele Konzepte teilen, und auch oft eine starke Verwandtschaft aufweisen, wird dem Kenner von mindestens einer Sprache, das Hinzulernen einer weiteren Sprache deutlich leichter fallen. Deshalb genügt es meist, nur eine oder wenige Sprachen gut zu kennen. Mit diesen Vorkenntnissen können dann weitere Sprachen, bei Bedarf, recht schnell dazugelernt werden.
Wir haben uns entschieden, in diesem Kurs mit der Programmiersprache 'Python' zu arbeiten. Die Wahl fiel uns nicht schwer, da viele Gründe für sie sprechen: Python ist modern, enorm stark verbreitet, universell einsetzbar und höchst einsteigerfreundlich.

Zu diesem Thema lassen sich Video-Erklärungen auf unserem YouTube-Kanal
www.youtube.com/channel/UCxIIMOe4oVImeJQbsphVjug/playlists
finden.
Viele der Beispiele, die wir in diesem Kapitel besprechen werden, sind stark motiviert durch die Beispiele, welche im exzellenten Python-Tutorial der *Python Software Foundation*:
https://docs.python.org/3/tutorial/
gefunden werden können.

[1] Liste von verbreiteten Programmiersprachen:
https://en.wikipedia.org/wiki/List_of_programming_languages

## 3.2 Python Befehle im Terminal und erstes Programm

An den Computern der Schule werden wir mit einer bereits vorinstallierten Python-Distribution arbeiten. Für das Arbeiten in Python an privaten Rechnern, empfehlen wir die Python-Distribution *Anaconda* zu installieren. Genaue Anleitungen und Informationen zu Anaconda findet Ihr im Anhang A oder auf unserem YouTube-Kanal.
Alle folgenden Betrachtungen sind allgemeingültig und unabhängig vom persönlichen Python-Setup.

### 3.2.1 Python Befehle im Terminal

Um Python-Code ausführen zu können, müssen wir das Terminal unter macOS starten. Dazu klickt man oben rechts auf *Spotlight* (Lupen-Symbol) und sucht im Suchfeld nach 'terminal', um das Terminal zu öffnen. Im Terminal gibt man das Wort `python` ein und drückt anschliessend die *ENTER*-Taste. Dadurch gelangt man in das Python-Environment. Als kleinen Test kann man den Befehl `import this` eingeben und sollte dann folgenden Output im Terminal erhalten:

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Listing 1: The Zen of Python

Die Auflistung 1 ist eine Sammlung von Software-Prinzipien, welche das Design der Python-Programmiersprache beeinflussen. Beim Command `import this` handelt es sich um ein sogenanntes *easter egg*[2].
Im Folgenden wollen wir viele weiter Python-Commands kennenlernen.
Als erstes wollen wir sehen, wie uns das Python-Terminal als gewöhnlicher Taschenrechner dienen kann:

```
>>> 4+7*3 # Punkt vor Strich
25
>>> 8/5 # gewöhnliche Division
1.6
>>> 8//5 # ganzzahlige Division (floor division)
1
>>> 12%7 # modulo Operation (Rest der ganzzahligen Division)
5
>>> (80-4*5)/2 # Klammersetzung
30.0
>>> 5**2 # 5 hoch 2
25
```

Kommentare haben wir bereits in LaTeX in Kapitel 2 kennengelernt. In LaTeX beginnen Kommentare mit dem Prozent-Symbol (%), in Python hingegen werde Kommentare mit dem Hashtag-Symbol (#) eingeleitet.

### 3.2.2 Erstes Python-Programm

Das Python-Terminal ist nützlich um einzelne Befehle auszuführen. Ganze Python-Programme lassen sich aber komfortabler in einem Texteditor schreiben. Dazu öffnen wir, in einem beliebigen Texteditor, ein neues Dokument und schreiben Folgendes in das (noch) leere Dokument:

```
print("Hello, World!")
```

Nun geben wir dem Dokument den Namen `hello_world.py` und speichern es auf dem `Desktop` ab. Dies ist unser erstes Python-Programm. Um das Programm auszuführen, öffnen wir ein neues Terminal (da wir dafür nicht im Python-Environment sein wollen) und führen den Befehl:

```
cd Desktop/
```

[2] Gute Erklärung des Begriffs "easter egg": https://en.wikipedia.org/wiki/Easter_egg_(media)

aus.
Der Befehl `cd` steht für *change directory* und lässt uns in das Verzeichnis *Desktop* wechseln, wo unser `hello_world.py` File gespeichert ist. Das Programm `hello_world.py` kann nun durch folgenden Befehl (im Terminal) ausgeführt werden

```
python hello_world.py
```

und sollte den Output

```
Hello, World!
```

liefern.
Typischerweise, schreibt man (mehrzeilige) Programme in einem Texteditor. Das Python-Terminal ist aber zum Testen einzelner Commands hervorragend geeignet.

## 3.3 Grundoperationen

### 3.3.1 Variablen deklarieren, Typ einer Variablen

Eine der häufigsten Tätigkeiten beim Programmieren in Python,- ist das Deklarieren von Variablen. Das Gleichheitszeichen (=) wird verwendet um Variablen einen Wert zuzuweisen (assignment operator):

```
>>> a = 15 # a soll 15 sein
>>> b = 5 # b soll 5 sein
>>> c = a + b # c soll die Summe von a und b sein
>>> print(c) # gib die Variable an den Output aus
20
>>> x = "wundervoll" # x ist die Zeichenfolge (string) wundervoll
>>> print("Python ist " + x) # füge zwei Strings zusammen
Python ist wundervoll
>>> word = 'Python'
>>> word[0] # 1. Zeichen des Wortes
'P'
# (in Python hat das erste Element die Postion 0 und nicht 1)
>>> word[1] # 2. Zeichen des Wortes
'y'
>>> word[-1] # letztes Zeichen des Wortes
'n'
# Zeichen von Position 0 (enthalten) bis 2 (nicht enthalten):
```

```
>>> word[0:2]
'Py'
# Zeichen von Position 3 (enthalten) bis zum Ende
>>> word[3:]
'hon'
>>> x = 1 # x ist eine ganze Zahl (integer)
print(type(x)) # der Typ von x ist integer (int)
<class 'int'>
>>> y = 1.0 # y ist eine Gleitkommazahl (floating point number)
>>> z = 1.1 # z ist auch eine floating point number (float)
>>> print(type(y)) # der Typ von y ist float
<class 'float'>
>>> print(type(z))
<class 'float'>
>>> print(type(word)) # word ist ein String (str)
<class 'str'>
# die Funktion len() sagt uns die Länge eines Strings
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

### 3.3.2 Input, Typecasting

Öffne ein neues Textfile und speichere dieses als `my_python.py` auf dem Desktop. Schreibe folgenden Code in das File und speichere erneut.

```
print("Enter your name:")
x = input()
print("Hello, " + x)
```

Wie bereits bei unserem `Hello, World` Programm (siehe Unterabschnitt 3.2.2), kann das Programm mit dem Command `python my_python.py` im Terminal ausgeführt werden.
Der Command `x = input()` liest den Input aus dem standard-Inputstream (üblicherweise der Input von der Tastatur) und speichert ihn als String in der Variable `x`.
Was ist aber, wenn wir möchten, dass der User eine (ganze) Zahl und nicht ein String eingeben kann? Würde man nämlich im obigen Programm anstelle eines Namens, eine Zahl mit der Tastatur eingeben (z.B. `42`), dann würde diese Folge von Zahlen nicht als der Integer (Ganzzahl) `42` gespeichert werden, sondern als der String `'42'`. Dieses Problem führt uns zum Begriff

des *Typecastings*. Typecasting ist die Umwandlung eines Datentyps in einen anderen:

```
a = 1.8 # a ist vom Typ float
# a wird explizit in einen Integer umgewandelt
# dies geschieht durch das Weglassen des dezimalen Teils
a = int(1.8) # a hat nur den ganzzahligen Wert 1
# dieses Programm ermöglicht dem User die Eingabe eines Integers
print("Gib eine ganze Zahl ein:")
# der Tastatur Input (String) wird explizit zu einem
# Integer konvertiert (Typecasting)
x = int(input()) # int(...) = explizites Casting nach int
print("Du hast folgende Zahl eingegeben:", x)
# implizites Typecasting findet hier statt:
b = 10/5.0 # b wird den float Wert 2.0 (und nicht 2) haben
# der Integer 10 wurde implizit zu einem float konvertiert
```

## 3.4 Control Flow

Alle unsere bisherigen Python-Programme wurden von oben nach unten ausgeführt (linear control flow). Diese Art von Control Flow ist sehr restriktiv, da jeder Command in unserem Programm maximal einmal ausgeführt wird. Nehmen wir an, wir möchten ein Programm schreiben, welches eine Million Schritte auf irgend einem Input durchführt. Mit linearem Control Flow hätte dieses Programm mindestens eine Million Zeilen. Offensichtlich ist dies nicht wünschenswert. Wir benötigen deshalb mächtigere Tools, um mit solchen Aufgaben sinnvoll umgehen zu können.
Zu den wichtigsten Konzepten, die einen nicht linearen Control Flow ermöglichen, gehören sogenannte *control statements* wie
`for`-loops, `while`-loops, `if`-statements und `jump`-statements.
Diese wollen wir nun anhand von Beispielen kennenlernen.

### 3.4.1 for-loops

Loops sind, in praktisch jeder Programmiersprache, ungeheuer wichtig. Betrachten wir dazu einige Beispiele:

```
# for-loop, welcher die Zahlen 0, 1, 2, ..., 8, 9 ausgibt:
for i in range(10):
    print(i) # Output: 0, 1, 2, ..., 8, 9

```

```
for i in range(3,8):
   print(i) # Output: 3, 4, 5, 6, 7

# Berechne die Summe der Ersten
# 100'000 Zahlen (sum = 1+2+3+...+100000)
sum = 0 # initialisiere die Summe mit 0
for k in range(1,100001):
   sum = sum + k # erhöhe die Summe um den nächsten Summanden
print(sum) # Output: 5000050000

# erstelle eine Liste von Strings
# eine Liste hat die Form [Element 1, Element2, ...]
languages = ['C++', 'Python', 'Fortran', 'Go', 'Assembly', 'C']
# sortiere die Liste alphabetisch
# die Bedeutung des Punktes "." in "list.sort" wird noch
↪  besprochen
list.sort(languages)
# \n bewirkt einen Zeilenumbruch:
print('Bekannte Programmiersprachen:\n')
# der Index 'j' geht durch die Liste "languages" durch
for j in languages:
    print(j) # Output: C++, Python, Fortran, Go, Assembly, C
```

Man beachte die Einrückung (indentation) nach Beginn des for-loops:

```
for i in range(10):
    print(i)
```

Alles, was unter dem for-loop eingerückt ist, gehört zum Körper (body) des Loops. Blöcke von Code werden in Python nicht, wie in sehr vielen anderen Programmiersprachen (wie `R`, `C++`, `C`), durch geschweifte Klammern strukturiert, sondern durch Einrücken. Dadurch soll der Code leserlicher werden. Obiger Loop würde in `R` z.B. so aussehen:

```
for(i in 0:9) {
  print(i)
}
```

Dabei ist die Einrückung der `print()`-Funktion in `R` reine Kosmetik und könnte auch weggelassen werden:

```
for(i in 0:9) {
print(i)
}
```

Ebenfalls obligatorisch in Python ist der Doppelpunkt am Ende der Deklaration des Loops: `for i in range(10):`

### 3.4.2 while-loops

Der while-loop ist sehr ähnlich aufgebaut wie der for-loop. Es lässt sich beweisen, dass jeder while-loop als for-loop geschrieben werden kann und umgekehrt. So gesehen, sind die beiden Konstrukte äquivalent.
Das Konstrukt der while-loops wird typischerweise dann eingesetzt, wenn nicht von Anfang an (a priori) klar ist, wie viele Schritte ausgeführt werden müssen. Als Beispiel dazu, betrachten wir die Fibonacci-Folge. Die Fibonacci-Folge $f_1, f_2, f_3, ...$ ist für $n > 2$ rekursiv definiert durch

$$f_n = f_{n-1} + f_{n-2}, \tag{3.1}$$

mit den Anfangswerten $f_1 = f_2 = 1$.
Nun wollen wir alle Folgeglieder der Fibonacci-Folge ausgeben, die kleiner sind als z.B. 500. Wie viele solche Folgeglieder dies sind, wissen wir (noch) nicht:

```
1  # Fibonacci-Folge mit while-loop
2  a, b = 0, 1 # Anfangswerte: a = 0, b = 1
3  # führe den while-loop aus, solange a kleiner ist als 500
4  while a < 500:
5      print(a)
6      a, b = b, a+b
7  # Output:
8  # 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377
```

### 3.4.3 jump-statements

Die jump-statements *continue* und *break* sind gelegentlich sehr nützlich. Ihre Bedeutung sollte aus einem Beispiel klar werden:

```
1  # errate die richtige Kombination eines 4-Stelligen
2  # Zahlenschlosses
3  kombination = 3110 # richtige Kombination
4
5  # der while-loop wird wiederholt, bis die richtige
6  # Kombination eingegeben wird
7  while True: # ist immer true
```

```
8       print('errate die 4-Stellige Kombination')
9       x = int(input())
10      if len(str(x)) != len(str(kombination)):
11          print('die Eingabe hat die falsche Laenge')
12          continue # der Rest des Loops kann übersprungen
             ↪ werden
13      if x == kombination:
14          print('richtig geraten!')
15          break # der while-loop kann nun abgebrochen werden
16      else:
17          print('leider falsch')
```

Nach Ausführung des **continue**-statements, wird der Rest der aktuellen Iteration (Schritt) übersprungen und mit dem nächsten Loop-Durchgang weiter gemacht. Das **break**-statement bricht den Loop ab.

### 3.4.4 if-statements

Die sogenannten if-statements werden beim Programmieren ständig gebraucht. Ihre Funktionsweise wollen wir wieder anhand einfacher Beispiele aufzeigen:

```
1  a = 50
2  b = 312.3
3  if a > b:
4     # kein Output, da a nicht grösser ist als b
5      print('a ist groesser als b')
6
7  x = 10.0
8  y = 0.0
9  if y != 0: # != steht für "nicht gleich"
10     z = x/y # die Division wäre hier erlaubt, da y nicht Null
        ↪ ist
11 else:
12    # in diesem Fall würden wir durch Null teilen
13     print('ACHTUNG! DIVISION MIT NULL!')
14
15 # gib alle nicht-negativen Zahlen < 100 aus,
16 # die durch 17 teilbar sind:
17 for i in range(100):
18     # % ist der modulo-Operator
19     # == ist der Vergleichsoperator
20     if i%17 == 0:
```

```
21          print(i)
22
23  x = -23.21329
24  if x > 0:
25      print('x ist eine positive Zahl')
26  elif x == 0: # else if
27      print('x ist genau Null')
28  else:
29      print('x ist negativ')
```

## 3.5 Funktionen in Python

Eines der wichtigsten Konzepte in Python ist das der Funktionen. Wir haben in diesem Kapitel schon einige Funktionen angetroffen, sind jedoch nicht näher auf diese eingegangen.
Zum Beispiel sind wir der Funktion `print(...)` häufig begegnet und haben (in Unterabschnitt 3.3.1) die Funktion `len(s)` benutzt, welche uns die Länge des Strings `s` genannt hat. Man beachte die grosse Ähnlichkeit dieser Python-Funktionen zu dem Funktionsbegriff, den wir aus der Mathematik kennen:
Genau wie in der Mathematik, hat eine Python-Funktion ein Name wie z.B. `len` (in der Mathematik wird als Funktionsname häufig $f$ gewählt) und akzeptiert eines oder mehrere Argumente `len(x)` $\leftrightarrow f(x)$. Um diese Ähnlichkeit noch besser zu verstehen, schauen wir uns an, wie in Python eine lineare Funktion

$$f(x) = ax + b, \tag{3.2}$$

mit $a, b \in \mathbb{R}$ und $a \neq 0$ (falls $a = 0$ gilt, ist die Funktion konstant und nicht linear). In Python lässt sich die lineare Funktion 3.2 wie folgt implementieren:

```
1  a = 2
2  b = 3
3  def lineare_funktion(x):
4      y = a*x + b
5      return(y)
6
7  # Aufruf der linearen Funktion mit dem Argument x == 5
8  t = lineare_funktion(5)
9  # t hat den Wert 2*5 + 3 = 13
10 print(t) # Output: 13
```

Eine Python-Funktion beginnt immer mit dem Keyboard **def**. Danach schreibt man, wie die Funktion heissen soll und (in Klammern) welche Funktionsargumente sie benötigt.
Was ist nun aber, wenn wir eine allgemeine lineare Funktion, mit beliebigen Werten der Parameter $a$ (Steigung) und $b$ ($y$-Achsenabschnitt) haben möchten? Ganz einfach, wir übergeben $a$ und $b$ ebenfalls als Funktionsargumente:

```
def lineare_funktion(x,a,b):
    y = a*x + b
    return(y) # gibt den Wert y zurück (return value)

# Aufruf der linearen Funktion mit dem Argument x == 5, a ==
 ↪  2 und b == 3
t1 = lineare_funktion(5,2,3)
# t1 hat den Wert 2*5 + 3 = 13
# um sicher zu sein, dass wir die Funktionsargumente nicht
 ↪  vertauschen
# können wir sie auch explizit beim Funktionsaufruf angeben
t2 = lineare_funktion(b=3,x=5,a=2) # t2 == t1
```

Um die Nützlichkeit von Python-Funktionen noch mehr zu betonen, schauen wir uns nochmals das Beispiel der Fibonacci-Folge aus Unterabschnitt 3.4.2 an. In diesem Beispiel wollten wir alle Glieder der Fibonacci-Folge die kleiner als 500, sind ausgeben. Wenn wir unsere Fibonacci-Folge als Python-Funktion $fibonacci(n)$ umformulieren, dann wird diese in der Lage sein, alle Fibonacci-Zahlen, die kleiner als $n$ sind auszugeben. Dies ist natürlich viel nützlicher und universeller, als sich auf eine fixe Zahl wie z.B. $n = 500$ zu beschränken:

```
# Fibonacci-Folge als Funktion
def fibonacci(n):
    # gibt alle Glieder der Fibonacci-Folge < n aus
    a, b = 0, 1 # Anfangswerte der Folge
    while a < n:
        print(a)
        a, b = b, a+b

fibonacci(4000) # Aufruf der Funktion für n == 4000
```

Man beachte, dass die Funktion `fibonacci(n)`, im Gegensatz zu der linearen Funktion, keinen Rückgabe-Wert (return value) hat.

# Literaturverzeichnis

[1] Volker Claus und Andreas Schwill. *Duden Informatik A-Z: Fachlexikon für Studium, Ausbildung und Beruf*, volume 4. Bibliographisches Institut & F. A. Brockhaus, 2006. 3

# Anhang A

# Installation von Python

Selbstverständlich ist jeder Schüler in der Wahl seiner Python-Distribution und des Texteditors völlig frei.
Wir empfehlen die *Anaconda-Distribution*[1] zu verwenden. Diese lässt sich auf *Microsoft Windows*, *macOS* sowie auf den meisten gängigen *GNU/Linux Distributionen* auf einfache Weise installieren.
Anaconda ist eine äusserst moderne und umfangreiche Python-Distribution, die alle Werkzeuge zur Verfügung stellt, welche wir benötigen werden (und noch sehr viele mehr). Im Umfang der Anaconda-Distribution sind unter anderem zahlreiche nützliche Software-Pakete (ein paar davon werden wir benötigen), die sehr übersichtliche integrierte Entwicklungsumgebung (integrated development environment, Abkürzung: IDE) *spyder*, sowie das *Jupyter Notebook* enthalten.
Im Folgenden wird beschrieben, wie sich Anaconda auf den entsprechenden Betriebssystemen installieren und starten lässt.
Sollte man sich für eine andere Distribution als Anaconda entschieden haben, ist man für deren Installation und Verwaltung selbst verantwortlich. Im Internet lassen sich häufig nützliche Installationsanleitungen finden.

## Installation von Anaconda für Windows OS

Öffne `https://www.anaconda.com/download/#windows` im Browser und lade die neueste Version von Python herunter (z.B. Python 3.6 oder höher). Die heruntergeladene Datei wird vermutlich von der Form `Anaconda3-VERSIONSNUMMER-Windows-x86_64.exe` sein. Dabei steht 'VERSIONSNUMMER' stellvertretend für diejenige Versionsnummer, welche zum Zeitpunkt des Downloads aktuell ist. Diese .exe-Datei kann durch einen

[1] offizielle Webseite: `https://www.anaconda.com/`

Doppelklick aufgerufen werden. Die Installation ist nun selbsterklärend (wähle überall die empfohlenen Einstellungen). Nach abgeschlossener Installation kann man über die Windows-Suchfunktion (auf dem Desktop unten links über 'Start' z.B.) nach 'Anaconda-Navigator' suchen und diesen aufrufen. Falls der Anaconda-Navigator erfolgreich gestartet ist, kann man von dort aus beispielsweise *spyder* öffnen.
Alternativ kann man auch den Instruktionen unter
https://docs.anaconda.com/anaconda/install/windows
folgen oder sich das YouTube-Video
https://www.youtube.com/watch?v=YB2zOi3Hd0I
anschauen.

## Installation von Anaconda für macOS

Öffne https://www.anaconda.com/download/#macos im Browser und lade die neueste Version von Python herunter (z.B. Python 3.6 oder höher). Die heruntergeladene Datei wird vermutlich von der Form
`Anaconda3-VERSIONSNUMMER-MacOSX-x86_64.bin` sein. Dabei steht 'VERSIONSNUMMER' stellvertretend für diejenige Versionsnummer, welche zum Zeitpunkt des Downloads aktuell ist. Diese .bin-Datei kann durch einen Doppelklick aufgerufen werden. Die Installation ist nun selbsterklärend (wähle überall die empfohlenen Einstellungen). Nach abgeschlossener Installation kann man über die macOS-Suchfunktion (*Spotlight*) nach 'Anaconda-Navigator' suchen und diesen aufrufen. Falls der Anaconda-Navigator erfolgreich gestartet ist, kann man von dort aus beispielsweise *spyder* öffnen.
Alternativ kann man auch den Instruktionen unter
https://docs.anaconda.com/anaconda/install/mac-os
folgen oder sich das YouTube-Video
https://www.youtube.com/watch?v=EFksZiYva5k
anschauen.

## Installation von Anaconda für GNU/Linux

Öffne https://www.anaconda.com/download/#linux im Browser und lade die neueste Version von Python herunter (z.B. Python 3.6 oder höher). Die heruntergeladene Datei wird vermutlich von der Form
`Anaconda3-VERSIONSNUMMER-Linux-x86_64.sh` sein. Dabei steht 'VERSIONSNUMMER' stellvertretend für diejenige Versionsnummer, welche zum Zeitpunkt des Downloads aktuell ist. Navigiere nun im Terminal (Shell), mit

Hilfe des Befehls `cd` (change directory), in den Ordner, in welchem sich die heruntergeladene Datei befindet. Führe dort im Terminal den Befehl

```
bash Anaconda3-VERSIONSNUMMER-Linux-x86_64.sh
```

aus, wobei natürlich 'VERSIONSNUMMER' durch die tatsächliche Versionsnummer der Datei ersetzt werden muss. Der komplette Befehl könnte also z.B. so lauten: `bash Anaconda3-5.1.0-Linux-x86_64.sh`. Dies startet den Installationsvorgang. Die Installation selbst ist nun selbsterklärend (drücke die `ENTER`-Taste um nach unten zu scrollen, wähle überall die empfohlenen Einstellungen und tippe, falls erforderlich, stets 'yes').
Führe nun den Befehl

```
source /.bashrc
```

im Terminal aus. Von jetzt an kann der Anaconda-Navigator stets ganz bequem über den Terminal-Befehl

```
anaconda-navigator
```

gestartet werden (die Vorherigen Schritte müssen natürlich nicht nochmals wiederholt werden). Falls der Anaconda-Navigator erfolgreich gestartet ist, kann man von dort aus beispielsweise *spyder* öffnen. Alternativ kann man auch den Instruktionen unter
https://docs.anaconda.com/anaconda/install/linux
folgen oder sich das YouTube-Video
https://www.youtube.com/watch?v=mjtkTbtakkM
anschauen.