Computerpraktikum

Einführung in Python
-
Überblicksskript

Hannes Kroke, Florian Hirsch

September 2021

Inhaltsverzeichnis

1 Einführung

Wir heißen euch herzlich willkommen zu diesem Python-Skript. Wir werden uns hier gemeinsam die Grundlagen von Python ansehen. Python ist eine weit verbreitete und einfach zu erlernende Programmiersprache. Im Vergleich zu anderen Programmiersprachen liegt der Fokus hier deutlich mehr auf *Einfachheit* und *Übersichtlichkeit*: simple Programme sollen auch simpel aussehen und simpel zu schreiben sein.

Besonders in den Naturwissenschaften wird Python häufig verwendet, was mehrere Gründe hat. Erstens ist Python sehr anwenderfreundlich. Zweitens gibt es sehr viele Erweiterungen zu Python (die sogenannten *Packages*). Über viele konzeptuelle Problemstellungen wurde bereits vor uns nachgedacht, und die optimierten Lösungen dafür sind in Packages zu finden. Drittens ist Python-Code insgesamt sehr übersichtlich. Wo in anderen Programmiersprachen viele Semikolons und undurchsichtige Schlüsselbegriffe zu sehen sind, ist bei Python oft schon aus dem Namen von Funktionen klar, was sie tun werden. Der letzte gute Grund ist die Möglichkeit des schnellen Ausprobierens: Da wir vergleichsweise wenig Code schreiben müssen und dabei auf professionell entwickelten Code zurückgreifen können, uns gleichzeitig aber wenig Sorgen um formale Aspekte machen müssen, können kleine Probleme und Simulationen schnell und unkompliziert umgesetzt werden.

Python eignet sich also sehr gut für ProgrammieranfängerInnen, trotzdem ist das Lernen der Grundlagen dieser Programmiersprache ein großer Schritt. In diesem Skript wollen wir uns deshalb nur über die grundlegendsten Konzepte einen Überblick verschaffen. Es soll nicht darum gehen, danach Python zu beherrschen: Es soll darum gehen, dass wir in gegebenem Python-Code die Struktur erkennen können, und einfache Programme nachvollziehen können. Nachvollziehen bedeutet in diesem Fall, dass wir anhand unseres Wissens plausible Vermutungen anstellen können, wofür ein bestimmter Codeteil zuständig sein könnte.

Falls die Lektüre dieses Skripts Lust auf mehr macht, empfehlen wir das ausführlichere Python-Skript. Dort werden die Themen dieses Skripts in etwas mehr Detail und mit mehr Codebeispielen vermittelt.

Wir werden jetzt damit beginnen, Python näher kennenzulernen. Und los geht's.

2 Installation

Die für das Nachvollziehen dieses Skripts erforderliche Installation von Python findet sich in einer Schritt-für-Schritt-Anleitung zu Beginn des ausführlichen Python-Skripts erklärt.

3 Variablen und Datentypen

Variablen In der Programmierung mit Python haben *Variablen* eine ganz einfache Aufgabe: Sie dienen als Platzhalter. Wir müssen also nicht jedes Mal den kompletten Wert der Variable schreiben, sondern arbeiten nur mit dem Variablennamen weiter. Dabei gibt es einen wichtigen Unterschied zwischen Variablen in Python und dem, was wir als Variablen seit Jahren kennen und benutzen: Beim Programmieren müssen wir Variablen immer zuerst *definieren*, bevor wir sie benutzen können. Wenn wir also x^2 für $x = 5$ berechnen wollen, müssen wir zuerst die Variable x als 5 definieren und können sie dann erst quadrieren.
Wir wollen nun also Variablen definieren, und sie danach ausgeben lassen. Mit der Funktion `print()` gibt Python den Wert der Variable in den Klammern aus (und zwar unter der ausgeführten Zelle in JupyterLab). Dies ist unsere Möglichkeit, die Werte von berechneten Variablen abzurufen. Es folgt ein Prgrammierbeispiel, das die bisher erwähnten Strukturen im Code zeigt:

```
x = 5                               # eine Zahl (Integer)
obstsorte1 = "Apfel"                # ein Text (String)
obstliste = ["Birne","Banane"]      # eine Liste (List)
y=2*x                               # y ist das Quadrat von x
print(y)                            # drucke den Wert von y aus
print(obstsorte1,obstliste)         # drucke Text und Liste aus
```

```
10
Apfel ['Birne', 'Banane']
```

Der Text hinter den `#` ist ein *Kommentar*. Wir sollten Kommentare möglichst häufig schreiben, um die Verständlichkeit des Codes zu verbessern. Beim Ausführen des Programms beachtet Python Kommentare nicht.
Die Definition einer Variable überschreibt dabei alle Informationen, die vorher in der Variable gespeichert waren. Auch der Typ der Variable, den wir uns gleich etwas genauer ansehen werden, kann sich dabei ändern. Der Datentyp sagt dabei nicht allein zu Informationszwecken, ob eine Variable eine Zahl oder ein Text ist, sondern er legt auch fest, was wir mit der Variable machen können. Wie wir jetzt sehen werden, können wir mit unterschiedlichen Datentypen Unterschiedliches machen.

Boolean Auf unserer Kurzreise durch die verschiedenen Datentypen beginnen wir bei der einfachst möglichen Struktur: dem *Boolean*. Eine Variable des Datentyps Boolean kann nur zwei mögliche Werte annehmen: `True` (wahr) oder `False` (falsch).
Gerade für Abfragen, ob bestimmte Aussagen wahr oder falsch sind (zum Beispiel: Ist die Zahl x größer als die Zahl y?) sind Booleans wichtig. Wir kommen im nächsten Kapitel auf diese sogenannten *Fallunterscheidungen* zurück.

Strings Unser nächster Datentyp ist der *String*: die *Zeichenkette*. Um einen String zu erstellen, benutzen wir meist doppelte Anführungsstriche. Strings dienen also zur Speicherung von Text, präzise ausgedrückt: Sie sind eine sortierte Aneinanderreihung von Zeichen.
Die Anzahl dieser Zeichen ist die *Länge* des Strings, die wir über die Funktion `len()` ausgeben lassen können.

```
obst = "kirsche"    # obst ist ein String, was man an den " erkennt
laenge = len(obst)  # die Länge des Strings ist die Anzahl seiner Zeichen
print(laenge)       # schließlich geben wir die Länge aus
```

```
7
```

Nun können wir schon das erste spannende Konzept kennenlernen, das uns auch noch bei den Listen helfen wird: Die Zeichen sind nummeriert (auch *indiziert* genannt) und wir können einzeln auf sie zugreifen. Dazu schreiben wir den gewünschten Index in eckige Klammern hinter den Namen des Strings. Achtung: Nummerierungen beginnen *immer* bei 0.
Wir können auch auf mehrere Zeichen gleichzeitig zugreifen. Dies tun wir mit dem *Slicing*. Dazu schreiben wir in die eckigen Klammern einen Doppelpunkt, und davor den Beginn der gewünschten Zeichenkette, danach das Ende (nicht inklusive, siehe Codebeispiel). Schreiben wir nichts vor den Doppelpunkt, beginnt der Ausschnitt am Anfang, schreiben wir nichts hinter den Doppelpunkt, endet der Ausschnitt am Ende des Strings. Außerdem können wir auch negative Indizes benutzen. Dabei geht Python von hinten nach vorn. Das alles sieht dann so aus:

```
obstname = "kirsche"      # ein String
#  k  i  r  s  c  h  e
#  0  1  2  3  4  5  6    # zugeordnete Indizes
# -7 -6 -5 -4 -3 -2 -1    # negative Indizes in umgekehrte Reihenfolge
print(obstname[1])        # gib den zweiten Buchstaben aus
print(obstname[1:5])      # gib den zweiten bis fünften Buchstaben aus
print(obstname[-3])       # gib den vorvorletzten Buchstaben aus
```

```
i
irsc
c
```

Es gibt auch noch viele weitere Funktionen für die Arbeit mit Strings, die bisher vorgestellten sollen uns aber genügen.

Integer und Floats Beide Datentypen sind Zahlen. *Integer* sind ganze Zahlen, *Floats* Kommazahlen. Wir werden diese Namen in diesem Abschnitt synonym behandeln. Unterscheiden können wir sie aber klar voneinander: Floats haben immer einen Punkt (wir würden ein Komma schreiben, aber es wird als Punkt notiert), selbst wenn hinter dem Komma eigentlich nichts steht.
Mit Zahlen können wir rechnen, und zwar genau so, wie wir es kennen. Die wichtigsten Rechenoperationen seien hier exemplarisch vorgeführt:

```
a = 3                     # Variable a soll gleich 3 sein und ein Integer
b = 4.                    # Variable b soll gleich 4 sein und ein Float
print("a + b =",a + b)    # Summe
print("a * b =",a * b)    # Produkt
print("a - b =",a - b)    # Differenz
print("a / b =",a / b)    # Quotient
print("a^b =",a**b)       # Potenz a^b
```

```
a + b = 7.0
a * b = 12.0
a - b = -1.0
a / b = 0.75
a^b = 81.0
```

Sollten wir doch einmal explizit einen der beiden Datentypen brauchen, so können wir sie leicht ineinander umwandeln. Dazu setzen wir die Variablen einfach in die Klammern von int() bzw. float() und erhalten dann einen Integer bzw. einen Float.
Floats können wir runden, was oft praktisch ist. Dazu verwenden wir die Funktion round(), die als zweites Argument (hinter einem Komma) die Anzahl an Stellen bekommt, auf die gerundet werden soll. Wir sehen es im folgenden Codebeispiel:

```python
a = 5.746253      # ein Float (Kommazahl)
b = int(a)        # mache einen (abgeschnittenen) Integer daraus
print(b)          # drucke den Integer aus
c = round(a,2)    # runde den Float auf 2 Nachkommastellen
print(c)          # gib den gerundeten Float aus
```

```
5
5.75
```

Listen und Tupel Nun kommen wir zu einem Konzept, das wir besonders häufig brauchen werden: *Listen*. Wie eine tatsächliche Liste bestehen auch Variablen des Datentyps `list` aus mehreren Einträgen in einer bestimmten Reihenfolge.
Eine Liste zeichnet sich durch die eckigen Klammern aus. Zwischen den Listeneinträgen stehen Kommata, und das ist auch in der Ausgabe der Fall. Eine Liste kann eine beliebige Anzahl von Einträgen haben. Einträge können dabei Variablen eines beliebigen Datentyps sein, sogar Listen als Einträge von Listen sind möglich.
Auch Listen sind indiziert, das heißt jedes Element hat eine Nummer, unter der es aufgerufen werden kann. Die Indizierung läuft genau so wie bei den Strings. Wenn wir allerdings Listen als Elemente von Listen haben, müssen wir, um einzelne Elemente zu erreichen, mehrere Indizes verwenden.

```python
obstliste = ["apfel",5,"banane",True,["weintraube","kiwi"]]     # eine ganz normale Liste
print(obstliste[2])         # drittes Element der Liste
print(obstliste[4])         # fünftes Element der Liste (die Unterliste)
print(obstliste[4][1])      # zweites Element der Unterliste
```

```
banane
['weintraube', 'kiwi']
kiwi
```

Natürlich können wir Listen auch verändern. Wir können neue Werte *anhängen* mit `append()`, und bestehende Werte *löschen* mit `remove()`. Listen haben wie Strings eine Länge, die wir mit `len()` abrufen.

```python
obstliste = ["apfel",5,"banane",True,["weintraube","kiwi"]]     # unsere Liste von vorhin
obstliste.remove("apfel")          # entferne das Element "apfel" aus der Liste
print(obstliste,len(obstliste))    # drucke veränderte Liste und ihre Länge
obstliste.append(5.746253)         # füge den Float 5.746253 hinzu
print(obstliste,len(obstliste))    # drucke veränderte Liste und ihre Länge
```

```
[5, 'banane', True, ['weintraube', 'kiwi']] 4
[5, 'banane', True, ['weintraube', 'kiwi'], 5.746253] 5
```

Schließlich wollen wir uns noch ansehen, dass wir Listen *entpacken* können. Entpacken bedeutet, dass wir die Listenelemente einzelnen Variablen zuordnen können. Wir erkennen die Funktionsweise am folgenden Codebeispiel sehr gut:

```python
gemueseliste = ["gurke","paprika","kohlrabi"]
gemuese1,gemuese2,gemuese3 = gemueseliste
print(gemuese1,gemuese3)
```

```
gurke kohlrabi
```

4 Fallunterscheidungen

Wie bereits angedeutet, ist Python auch dazu imstande, bestimmte Codeteile nur auszuführen, wenn gewisse Bedingungen erfüllt sind. Wir reden dann von *Fallunterscheidungen*.
Es gibt drei wichtige Schlüsselbegriffe für Fallunterscheidungen: if, elif und else. Die grundlegende Struktur ist dann folgende: Zuerst schreiben wir das Schlüsselwort, dann die Aussage, die auf ihren Wahrheitswert (Boolean) überprüft werden soll, und dann kommt ein Doppelpunkt. Der im Falle einer erfüllten Bedingung auszuführende Codebereich muss um einen Tab eingerückt werden (dazu drücken wir auf die Tabulatortaste).

```
1 x=4          # der Wert von x ist 4
2 if x<6:      # Schlüsselwort "if", dann die wahre/falsche Aussage
3     print("Die Aussage hat gestimmt.")   # wird nur ausgeführt, wenn x<6, also 4<6 wahr ist
```

```
Die Aussage hat gestimmt.
```

if ist das einfachste Schlüsselwort. *Falls* die danach folgende Aussage wahr ist, wird der eingerückte Code ausgeführt, sonst nicht.
elif funktioniert wie if, wird aber nur beachtet, falls keine der vorherigen Aussagen wahr war. Demzufolge ergibt es nur in Kombination mit anderen Fallunterscheidungen Sinn. Es steht immer hinter if oder anderen elif. Ist bei mehreren elif hintereinander also eine Aussage davon wahr, so wird alles danach nicht mehr ausgeführt.
else sammelt alle Fälle auf, die in den vorherigen Bedingungen nie wahr waren. Steht nur ein if davor, so wird der Codeteil nach else dann und nur dann ausgeführt, wenn die Bedingung nach if falsch war. Stehen elif dazwischen, so wird der Codeteil nach else dann und nur dann ausgeführt, wenn *alle* Bedingungen nach den if und elif falsch waren.
Wir sehen diese Strukturen im folgenden Codebeispiel umgesetzt. Für x können wir verschiedene Zahlen einsetzen, und werden je nach Größe von x immer die entsprechende Aussage dazu bekommen.

```
1 x = 2
2 if x > 4:
3     print("x ist größer als 4.")
4 elif x >= 1:       # >= steht hier für ≥, <= stünde für ≤
5     print("x ist nicht größer als 4, aber größer oder gleich 1.")
6 elif x == 0:       # ob Zahlen gleich sind, prüfen wir mit einem doppelten =
7     print("x ist genau 0.")
8 else:              # falls alles bisher falsch war
9     print("Keine der Aussagen war wahr, x muss also kleiner als 0 sein.")
```

```
x ist nicht größer als 4, aber größer oder gleich 1.
```

Weitere Ergebnisse könnten sein (mit den 3 Punkten stellvertretend für die Zeilen 2 bis 9):

```
1 x = 200
2 ...
```

```
x ist größer als 4.
```

```
1 x = -1.5
2 ...
```

```
Keine der Aussagen war wahr, x muss also kleiner als 0 sein.
```

5 Schleifen

In Programmen kommt es oft vor, dass ein Codeabschnitt mehrfach ausgeführt werden soll. Es wäre zum Beispiel wirklich praktisch, das Codebeispiel von eben automatisch für mehrere Werte von x durchlaufen lassen zu können. Dazu brauchen wir *Schleifen*. Wir unterscheiden Schleifen in for- und while-Schleifen, und wir beginnen mit den häufiger eingesetzten for-Schleifen.

for-Schleifen Die Struktur ist relativ ähnlich zu der von Fallunterscheidungen. Wir beginnen mit dem Schlüsselwort for. Dann kommt der Name der sogenannten *Laufvariable*, also der Variable, die sich mit jedem Durchlauf der Schleife ändert. Es folgt das Schlüsselwort in und dann eine Liste, in der die Werte der Laufvariable stehen, die durchlaufen werden sollen. Schließlich kommt ein Doppelpunkt, und der Codeteil, der zur Schleife gehört, wird eingerückt. Wir machen dazu schon ein kleines Beispiel:

```
liste = [1,3,5]
for i in liste:  # Laufvariable heißt i, durchläuft die Werte 1, 3 und 5
    print(i)     # drucke die Laufvariable aus
```

```
1
3
5
```

Strukturell ist das schon alles zur for-Schleife. Wir wollen aber noch eine Funktion vorstellen, die uns Arbeit abnimmt. Häufig wollen wir einen Codeteil einfach beispielsweise 15 Mal durchlaufen lassen, und uns ist der Wert der Laufvariable egal, oder wir wollen einfach, dass die Zählung bei 0 beginnt und in Einerschritten hochzählt. Dafür gibt es die Funktion range(). Wir können ihr ein Argument übergeben, das ist dann der letzte Wert der Laufvariablen (nicht inklusive). So haben range(5) und [0,1,2,3,4] sowie range(len([5,"heidelbeere"])) und [0,1] den gleichen Effekt.

while-Schleifen while-Schleifen sind ebenfalls Schleifen, arbeiten aber nach einem grundlegend anderen Konzept. Es gibt keine Laufvariable, sondern nur eine Wahrheitsaussage (genau so wie bei Fallunterscheidungen). Zu Beginn jedes Durchlaufs der Schleife wird die Bedingung hinter while überprüft, und wenn sie wahr ist, wird die Schleife ein weiteres Mal ausgeführt. Ist sie nicht wahr, geht es mit dem Code hinter der Schleife weiter. Demzufolge wird einfach die gesamte Schleife übersprungen, wenn die Bedingung zu Beginn schon falsch ist. Wir machen dazu ein Beispiel, das Elemente aus einer Liste löscht, bis die Liste leer ist.

```
liste = [27,18,"beere",True]
while len(liste) != 0:       # Schleife so lange ausgeführen, bis Länge der Liste 0 ist
    liste.remove(liste[0])   # entferne immer erstes Element der Liste aus der Liste
    print(liste)             # aktuelle Liste
```

```
[18, 'beere', True]
['beere', True]
[True]
[]
```

Wir sehen: In dem Moment, in dem die Länge der Liste 0 ist, die Liste also leer ist, wird die Schleife *nicht* noch einmal durchlaufen.

6 Funktionen

Motivation Den Begriff *Funktion* haben wir nun schon öfter gehört. Doch was sind Funktionen eigentlich?
Funktionen sind im Grunde zusammengefasste Codeblöcke. Wenn wir die Funktion aufrufen, wird der Codeteil in der Funktion ausgeführt. Doch nicht nur das: Wir können Funktionen auch sogenannte *Argumente* übergeben (das sind ganz normale Variablen), und die Funktion kennt dann diese Werte und kann sie verwenden. Außerdem können Funktionen Werte zurückgeben. Damit können wir also einer Funktion Argumente übergeben, die Funktion wertet diese aus (führt also den Code in ihr aus) und gibt uns dann das Ergebnis zurück.

```
1  def plusrechnen(a,b):          # Definition der Funktion
2      summe = a + b              # Berechnung der Summe
3      return summe               # Rückgabe des Ergebnisses
4
5  ergebnis = plusrechnen(4,5)  # Ausführen der Funktion und "Auffangen" des Ergebnisses
6  print(ergebnis)              # Ergebnis ausdrucken
```

```
9
```

Diesmal drehen wir es um: Wir haben jetzt schon das Codebeispiel, und werden die Struktur anhand dessen erklären.
Funktionen müssen definiert werden. Dies erfolgt über das Schlüsselwort `def`. Danach folgt der Name der Funktion, in diesem Fall `plusrechnen`. Hinter den Funktionsnamen schreiben wir in Klammern die Argumente der Funktion. Unsere Funktion `plusrechnen` möchte also gern zwei Argumente (nicht weniger, nicht mehr) übergeben bekommen. Innerhalb der Funktion werden diese beiden Elemente dann `a` und `b` genannt, und zwar in dieser Reihenfolge (also das erste `a`, das zweite `b`). Es folgt der übliche Doppelpunkt, und der Code innerhalb der Funktion ist eingerückt. Am Ende kann das Ergebnis zurückübergeben werden. Dazu schreiben wir die zu übergebende Variable hinter das Schlüsselwort `return`. Dies sorgt dafür, dass `ergebnis` nun die Ausgabe der Funktion, also die Summe ist. Wir sehen das, indem wir es uns ausdrucken lassen.
Funktionen neigen dazu, sehr komplex auszusehen, aber die Struktur ist immer gleich. Nichtsdestotrotz sehen Funktionen nicht immer exakt so wie in unserem Beispiel aus. Es kann sehr viele Argumente geben, aber auch keine sind möglich. Es darf auch mehr als eine Variable zurückgegeben werden, dies geschieht dann als eine Art Liste, und die Variablen können so aufgefangen werden, wie wir es beim Entpacken von Listen gelernt haben. Es dürfen sogar mehrere `return` in einer Funktion stehen. Die Funktion wird aber beim Erreichen des ersten `return` beendet und die Variable hinter diesem Schlüsselwort zurückgegeben.

7 Mathematische Berechnungen mit Numpy

Wir werden uns nun mit einem sogenannten *Package* auseinandersetzen. Das Package Numpy zeichnet sich besonders dadurch aus, dass es sehr effizient und schnell läuft, und die Handhabung bei mathematischen und numerischen Rechnungen angenehmer gestaltet.

Numpy Arrays Ein wichtiger Aspekt in Numpy sind die sogenannten *Numpy Arrays*. Es sind im Grunde Listen, nur dass es ein paar andere Funktionen gibt. Zu diesen Funktionen gehört beispielsweise `np.linspace(start,stop,num)`. Dieser Funktion übergeben wir drei Argumente: den ersten Wert einer Liste, den letzten Wert einer Liste und die Anzahl an Listenelementen. Dann generiert Numpy aus diesen Informationen einen Array der Länge `num`, dessen kleinster Wert `start` und größter Wert `stop` ist, und dessen Werte alle den gleichen Abstand zueinander haben. Das ist ziemlich praktisch.

```
1  import numpy as np              # zunächst müssen wir Numpy importieren
2                                  # wir kürzen numpy mit np ab, um Schreibaufwand zu sparen
3  array = np.linspace(-2,2,11)   # der Array geht in 11 Schritten von -2 bis 2
4  print(array)                   # drucke den Array aus
```

```
[-2.  -1.6 -1.2 -0.8 -0.4  0.   0.4  0.8  1.2  1.6  2. ]
```

Rechnen mit Arrays Bisher kennen wir noch keine wirklichen Unterschiede zwischen Arrays und Listen (außer, dass wir mit `np.linspace()` Arrays erstellen, nicht Listen). Ein großer Unterschied zeigt sich im Rechnen mit Arrays. Wenn wir eine Rechenoperation auf einen Array anwenden, dann wird diese Rechenoperation auf jedes einzelne Element des Arrays angewendet (dies ist bei Listen nicht der Fall). Außerdem stellt Numpy Rechenoperationen zur Verfügung, die für uns an der Tagesordnung, in Python aber nicht verfügbar sind. Welche das sind, zeigt die nachfolgende Tabelle exemplarisch. Danach machen wir uns diesen Unterschied zwischen Arrays und Listen anhand eines Codebeispiels klar.

Funktion	Bedeutung	Funktion	Bedeutung
np.pi	Kreiszahl π	np.sqrt()	Wurzel
np.abs()	Betrag	np.sin()	Sinus
np.exp()	Exponentialfunktion (Basis e)	np.log()	Natürlicher Logarithmus

Tabelle 1: Wichtigste Funktionen und Konstanten aus Numpy

```
5  liste = [1,1.5,2,2.5,3]
6  print("Liste:",liste)
7  array = np.linspace(1,3,5)              # der Array geht in 5 Schritten von 1 bis 3
8  print("Array:",array)
9  print("3 x Liste:",3*liste)             # drucke die verdreifachte Liste aus
10 print("3 x Array:",3*array)             # drucke den verdreifachten Array aus
11 print("quadrierter Array:",array**2)    # drucke den quadrierten Array aus
```

```
Liste: [1, 1.5, 2, 2.5, 3]
Array: [1.  1.5 2.  2.5 3. ]
3 x Liste: [1, 1.5, 2, 2.5, 3, 1, 1.5, 2, 2.5, 3, 1, 1.5, 2, 2.5, 3]
3 x Array: [3.  4.5 6.  7.5 9. ]
quadrierter Array: [1.   2.25 4.   6.25 9.  ]
```

8 Daten lesen

Wenn wir Daten auswerten wollen, dann müssen wir sie erst einmal in das Programm einlesen. Es wäre allerdings ziemlich aufwändig und nicht zielführend, wenn wir sie direkt in den Code hineinschreiben würden, also importieren wir sie aus einer Datei. Eine einfache Möglichkeit, die Daten aus einer Datei einzulesen, ist, sie aus einer Textdatei (*.txt*) einzulesen. Deshalb werden die folgenden Abschnitte darauf abgestimmt sein.

Zuerst müssen wir die Datei finden. Am einfachsten funktioniert dies, wenn die Datei im selben Verzeichnis (also in unserer Ordnerstruktur auf dem Computer am selben Ort) wie die Datei, in der wir unseren Code schreiben, ist (siehe erste Zeile). Falls dem nicht so ist, müssen wir den korrekten Pfad angeben (siehe zweite Zeile). Im Folgenden wollen wir die Datei also zunächst einmal öffnen:

```
1  datei = open("data.txt",mode="r",encoding="utf-8")
2  datei = open(r"C:\Users\Username\Desktop\data.txt",mode="r",encoding="utf-8")
```

Wir öffnen eine Datei mit `open("Dateiname/Pfad","Modus")` (und beachten das kleine *r vor* dem Pfad, das dafür sorgt, dass Python die \ nicht als Funktionen interpretiert). In welchem Modus wir eine Datei öffnen wollen, hängt ganz davon ab, was wir mit der Datei machen wollen. Wir wollen nur lesen, also heißt unser Modus "r" für Lesen. Das Argument `encoding` verhindert einige Komplikationen und macht das Einlesen angenehmer.
Tatsächlich lesen wir die Datei erst mit `read()` wirklich ein, denn uns interessiert nur der Text, der in der Datei steht. `read()` gibt uns den gesamten Dateiinhalt als String.

```
1  datei = open("data.txt","r",encoding="utf-8")  # nur die Datei, noch nicht der Text
2  text = datei.read()                            # read() anwenden
3  print(text)                                    # Inhalt der Datei als String
4  datei.close()                                  # schließe Datei nach dem Auslesen wieder
```

```
Obst Farbe Größe
Apfel grün mittel
Erdbeere rot klein
Heidelbeere blau winzig
```

Mit der Funktion `split()` sind wir dazu imstande, den Text so zu zerlegen, dass wir ihn verwenden können. Dazu verwandeln wir den Fließtext in eine Liste von Zeilen. Indem wir `split()` das Argument "\n" (ein Zeilenumbruch) übergeben, wird der Text nach Zeilen aufgetrennt. Danach trennen wir jede Zeile nochmal auf, diesmal ist das Trennzeichen (das Zeichen zwischen zwei Elementen) ein Leerzeichen.

```
5  gesamtliste=[]                 # erstelle schon mal Zielliste
6  zeilen = text.split("\n")      # trenne Text nach Zeilen
7  for zeile in zeilen:           # für jede Zeile im Text...
8      zeile = zeile.split(" ")   # ... trenne die Zeile nach Leerzeichen
9      gesamtliste.append(zeile)  # füge der Zielliste die gewonnene Zeile hinzu
10 print(gesamtliste)             # drucke fertige Liste aus
```

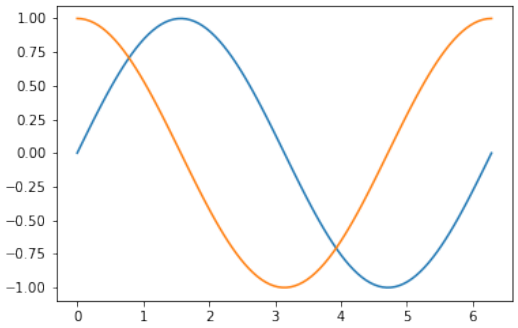
```
[['Obst', 'Farbe', 'Größe'], ['Apfel', 'grün', 'mittel'], ['Erdbeere', 'rot', 'klein'],
['Heidelbeere', 'blau', 'winzig']]
```

9 Diagramme zeichnen mit Pyplot

Wir haben eine Menge gelernt bis hierhin, und wollen uns nun endlich mal ein Bild von dem machen, was wir können. Wir gehen also über zum Zeichnen von *Diagrammen*. Hier ist die benötigte Struktur relativ einfach, es gibt aber sehr viele Einstellungsoptionen.

Wir benutzen zum Zeichnen von Diagrammen `pyplot` aus dem Package `matplotlib`. Das Plotten von Graphen funktioniert über die Funktion `plt.plot(x-Werte,y-Werte)`. Wir können auch mehrere Graphen in ein Diagramm plotten, dazu schreiben wir das `plot` mehrfach. Gezeigt wird das Diagramm dann mit der Funktion `plt.show()`.

```
1  import numpy as np                  # importiere Numpy
2  import matplotlib.pyplot as plt     # importiere Pyplot
3
4  xl = np.linspace(0,2*np.pi,100)     # 100 Datenpunkte zwischen 0 und 2π
5  yl1 = np.sin(xl)                    # Sinus-Funktion
6  yl2 = np.cos(xl)                    # Kosinus-Funktion
7
8  plt.plot(xl,yl1)                    # Plotten der ersten Funktion (blau im Bild)
9  plt.plot(xl,yl2)                    # Plotten der zweiten Funktion (orange im Bild)
10 plt.show()                          # und gleich zeigen
```



Wenn wir also mathematische Funktionen plotten, erstellen wir uns selbst x-Werte mit `np.linspace()`, und zwar möglichst so viele, dass der Graph glatt wird (und nicht eckig). Durch die elementweise Wirkung von Rechenoperationen auf Numpy Arrays können wir dann einfach die entsprechende Funktion auf die x-Werteliste anwenden, und bekommen so die y-Werteliste.

Für den Graphen und das Diagramm drumherum gibt es sehr viele Optionen. Wir wollen im folgenden Codebeispiel die wichtigsten beispielhaft zeigen. Vieles erschließt sich von selbst oder aus dem entsprechenden Kommentar dazu. Für uns NaturwissenschaftlerInnen, die wir selbst Experimente durchführen, sind Fehler sehr relevant. Deshalb wollen wir im folgenden Beispiel auch gleich `plt.errorbar()` vorstellen. Wir schreiben es statt `plt.plot()`. Auch diese Funktion nimmt die x- und die y-Werte als Argumente an, zusätzlich aber noch x- und/oder y-Fehler. Diese werden als Liste übergeben (die genau so lang wie die Listen der x- und y-Werte ist). Außerdem ist wichtig, dass die Fehler mit den Schlüsselworten `xerr` und `yerr` übergeben werden. Wir zeigen dies und einige grundlegende Einstellungen exemplarisch im folgenden Codebeispiel, das die lineare Regression einer Messreihe zeigt.

```
xwerte = [1.5,3,4.5,6,7.5,9]                          # x-Messwerte
ywerte = [8.4,11.8,15.1,20.4,24.4,27.9]               # y-Messwerte
xfehler = [0.2,0.2,0.1,0.3,0.2,0.2]                   # Fehler der x-Messwerte

x = np.linspace(1,10,50)                              # x-Punkte der Geraden
y = 2.6*x + 4                                         # Formel für die Gerade

plt.errorbar(xwerte,ywerte,                           # Messwerte
             xerr=xfehler,                            # übergebe die Fehler
             color="red",                             # Datenpunkte sind rot
             linestyle="",                            # keine Linie zwischen den Punkten
             marker="x",                              # Datenpunkte als Kreuze dargestellt
             capsize=3,                               # Größe der "Kappen" der Fehlerbalken
             capthick=2,                              # Dicke der "Kappen" der Fehlerbalken
             ecolor="blue",                           # Farbe der Fehlerbalken
             label="Messpunkte mit Fehlerbalken")     # Name des Graphen
plt.plot(x,y,                                         # Werte der Ausgleichsgeraden
         linestyle="--",color="green",                # gestrichelte grüne Linie
         label="Ausgleichsgerade")                    # Name des Graphen
plt.title("Beispielmessung")                          # Titel des Diagramms
plt.xlabel("x-Achse")                                 # x-Achsenbeschriftung
plt.ylabel("y-Achse")                                 # y-Achsenbeschriftung
plt.legend(loc="best")                                # Legende an der "besten" Position
plt.grid()                                            # Gitternetzlinien einschalten
plt.savefig("Beispieldiagramm.png")                   # Speichern des Diagramms als PNG-Bild
plt.show()                                            # Diagramm zeigen
```