Programmieren in Python

Begleitscript zur Vorlesung im Wintersemester 2020/21

Stand 2. März 2021

Universität Regensburg
Fakultät Physik

Vorwort

Zu Beginn dieses Kurses wollen wir uns eine bedeutende und tiefgreifende Wahrheit erkennen und in unserem Innersten verankern:

Computer sind strunzdumm.

Im Wesentlichen ist das auch ganz gut so. Wäre dem nicht so, könnten wir Ihnen nicht die Aufgaben überlassen, die uns zu lästig sind. Wie aber jeder weiß, der an einen weniger bemittelten Zeitgenossen Teile seiner Arbeit deligieren wollte, müssen die Anweisungen dann auch auf den Punkt exakt formuliert sein. Für Computer gilt das nur umso mehr. In den Übungsstunden der Programmierpraktika höre ich von KursteilnehmerInnen immer wieder so etwas wie *Ich dachte, der Computer würde verstehen dass...*, worauf ich gerne antworte: *Du denkst – der Computer denkt nicht.*

Die Programmiersprache *Python* wurde entwickelt, um gerade diese Schwierigkeit am Programmieren *etwas* abzumildern. Der Interpreter (also das Programm, das den Programmcode in Elektronik-Anweisungen übersetzt) erkennt viele implizite Angaben, die in anderen Sprachen mühevoll ergänzt werden müssten. Dies macht die Sprache gerade für AnwenderInnen attraktiv, die sich nicht primär als ProgrammiererInnen verstehen. Anstatt sich mit Speicherstrukturen und Datenflüssen zu beschäftigen, kann bei der Entwicklung in Python das Hauptaugenmerk auf dem übergeordneten Problem bleiben. Viele regelmäßig wiederkehrede Aufgaben sind in Python bereits gelöst und können als Standardbaublöcke in eigene Codes eingebaut werden. Dieses Sprachdesign veranlasste Randall Munroe[1] zur Veröffentlichung dieses Strips:

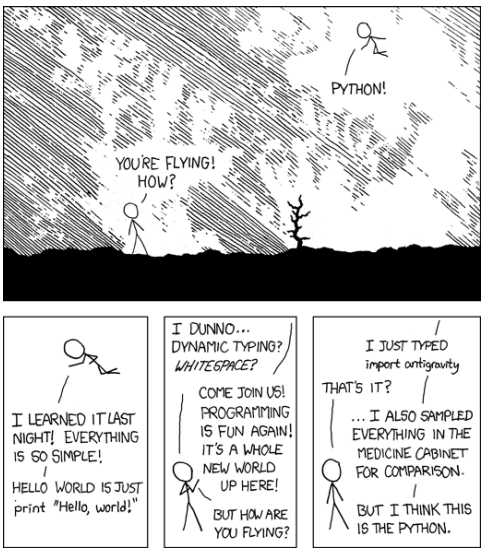


Abbildung 0.1.: Antigravity in Python. Quelle: `https://xkcd.com/353/`

[1] Autor des legendären Webcomics *xkcd* und der Bücher *What If*, *Thing Explainer* und *How To*. Falls Sie diese noch nicht kennen, füllen Sie diese Wissenslücke schnell auf – es lohnt sich!

Diese Euphorie soll nicht darüber hinweg täuschen, dass wir uns der schwierigen Aufgabe stellen, einem Computer unseren Willen abzuverlangen. Aber ähnlich wie Randall Munroe und ich werden Sie beim Erlernen dieser Fähigkeit mit Python schnell Erfolgserlebnisse haben und sicher Spaß daran finden.

Dieses Script soll Sie in der Vorlesung *Programmieren in Python* im Wintersemester 2020/21 begleiten. Alle wichtigen Kursinhalte werden hier behandelt und anhand von Beispielen verdeutlicht. Es versteht sich von selbst, dass ein optimaler Lernerfolg nur bei Besuch der Vorlesung gegeben ist. Vorkenntnisse in anderen Programmiersprachen sind zur Arbeit mit diesem Script nicht notwendig.

Hier wird von einer Linux-Arbeitsumgebung ausgegangen und minimale Grundkenntnisse dieser Arbeitsumgebung vorausgesetzt (Starten einer Kommandozeilen-Umgebung, Wechsel des Arbeitsverzeichnisses, Aufrufen von Programmen aus der Kommandozeile, Übergabe von Parametern an Programme). Die DozentInnen des Kurses können bei Bedarf erklären, wie die Arbeitsumgebung bedient wird.

Dieses Dokument ist keine vollständige Referenz der Sprache Python. Hier sollen nur die Grundlagen des Programmierens in Python vermittelt werden. Als Standard-Nachschlagewerk empfehle ich die offizielle Dokumentation `https://docs.python.org/3/` (englisch). Zum einen finden Sie dort ausführliche und vollständige Erklärungen zu allen Themen, die Ihnen bei der Arbeit mit Python begegnen können. Zum anderen haben Sie sicherlich Freude an den vielen Anspielungen auf die Sketche der Britischen Anarcho-Komiker *Monty Python*[2], nach denen die Sprache benannt ist.

Dieses Script wurde nach bestem Wissen und Gewissen zusammengestellt; Code-Beispiele wurden auf wenigstens einer Maschine getestet. Dennoch können menschliche Fehler nicht ausgeschlossen werden. Wem Unstimmigkeiten auffallen oder wer Vorschläge und Anregungen zu diesem Text einbringen will, möge mir dies sehr gerne mitteilen. Ich bin erreichbar unter meiner Email-Adresse:
`stefan.hartinger@stud.uni-regensburg.de`.

Stefan Hartinger, März 2020

[2] *Das Leben des Brian* und *Die Ritter der Kokosnuß*[3] sollten sie unbedingt sehen!
[3] Ja, die Filme sind so alt, dass man damals *Kokosnuß* noch mit scharfem ß schrieb.

Inhaltsverzeichnis

1. Die ersten Schritte

> A little girl goes into a pet show and asks for a wabbit. The shop keeper looks down at her, smiles and says:
> „Would you like a lovely fluffy little white rabbit, or a cutesy wootesly little brown rabbit?“
> „Actually“, says the little girl, „I don't think my python would notice.“
>
> Nick Leaton

In diesem Abschnitt wollen wir unsere Arbeitsumgebung kennen lernen. Zu diesem Zweck werden wir ein sogenanntes *Hello World* schreiben, d. h. ein Programm, das lediglich den Text `Hello World!` auf dem Bildschirm ausgibt. Im Weiteren werden wir Python dazu benutzen, einfache Rechnungen umzusetzen.

1.1. Der Kommandozeilen-Interpreter

Wenn Sie eine *Kommandozeilen-Umgebung* starten, können Sie Textkommandos an das Betriebssystem senden. Überwiegend handelt es sich dabei um die Anweisung, andere Programme auszuführen. Die Anweisung `python3`[1] ist ein solcher Befehl. Tippen Sie dies ein und drücken Sie `[ENTER]` um den Python-Interpreter zu starten.

Sie werden nun einen kurzen Versionstext sehen und hinter drei Pfeilen (`>>>`) einen blinkenden Cursor. Obwohl Sie noch immer dasselbe Fenster angezeigt bekommen, sind sie nun in der *Interpreter-Umgebung*. Die Befehle, die Sie hier eingeben können, gehören zum Sprachumfang von Python!

1.1.1. Hello World

Ein solcher Befehl ist `print`. Er dient dazu, Informationen auf dem Bildschirm auszudrucken – also genau das, was wir für unser Hello-World-Programm brauchen. Natürlich muss dazu auch angegeben werden, *was* gedruckt werden soll. Wir müssen also einen Text *als Argument übergeben*. Dies tun wir, indem wir den Text in Klammern () und Anführungszeichen `""` einrahmen. Die Klammern dienen dazu, klar zu machen, was Argument ist, und was zum restlichen Code gehört. Die Anführungszeichen brauchen wir, um Text, der buchstäblich zu behandeln ist, von anderem Code abzutrennen. (Stellen Sie sich vor, sie wollten den Text `print` auf dem Bildschirm ausgeben. Der Computer versteht ohne Anführungszeichen den Unterschied zwischen dem Text `print` und dem Befehl `print` nicht. Vielleicht verstehen Sie nun den Anfang des Vorworts.) Geben Sie also ein:

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

[1] Die Programmiersprache Python wurde seit ihrer Veröffentlichung im Jahre 1991 beständig weiterentwickelt. Manche Konzepte mussten komplett überarbeitet werden, so dass die einzelnen Versionen der Sprache nicht zwingend kompatibel miteinander sind. Wir arbeiten in der derzeit aktuellen Version 3.8. Auf einem Rechner können mehrere Python-Interpreter nebeneinander installiert sein. Daher müssen wir beim Aufruf die Versionsnummer 3 mit nennen.

Der Interpreter reagiert prompt, und auf dem Bildschirm finden Sie exakt das, was Sie erwarten: Die Zeile `Hello World!`.

Nach den letzten Schritten sollten Sie also folgendes auf dem Bildschirm sehen:

Starten des Python-Interpreters und Hello-World

```
blue-chameleon@blue-chameleon:~$ python3
Python 3.8.2 (default, Apr 27 2020, 15:53:34)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("hello world!")
hello world!
```

Sollten Sie sich vertippen, wird ihnen dies in Form einer (anfangs etwas kryptischen) Fehlermeldung mitgeteilt:

Eingabe mit Fehlern

```
>>> pirnt("hello world!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pirnt' is not defined
```

Die letzte Zeile dieser Fehlermeldung teilt Ihnen mit, dass der Befehl `pirnt` nicht existiert. Sie werden diese Fehlermeldung sehr häufig bei Tippfehlern sehen, wie dies in diesem Beispiel der Fall war. Da Programme zum Teil sehr lang werden können, unterstützt Sie der Interpreter beim Debuggen, indem in der vorletzten Zeile der Ausgabe die Stelle genannt wird, an der die fehlerhafte Eingabe stattfand: `line 1` sagt, dass die erste Zeile dieses Codes fehlerhaft war.

Vergessen wir beispielsweise, die Anführungszeichen abzuschließen, erhalten wir eine ähnliche Fehlermeldung:

Eingabe mit Fehlern

```
>>> print("hello world)
  File "<stdin>", line 1
    print("hello world)
                       ^
SyntaxError: EOL while scanning string literal
```

`EOL` steht für *end of line*: Bevor der *string literal* (also unser Text) durch ein abschließendes Anführungszeichen beendet wurde, endete die Zeile.

Ein interessantes Verhalten erzielen Sie, wenn Sie die abschließende Klammer vergessen: Argumente in Python dürfen sich über mehrere Zeilen erstrecken. Anstatt eine Fehlermeldung zu zeigen, wird der Interpreter Sie über drei Punkte (`...`) auffordern, die Zeile fortzusetzen. Wenn Sie hier die Klammer nachträglich schließen, erhalten Sie die erwartete Ausgabe:

Eingabe über mehrere Zeilen

```
>>> print("hello world"
... )
hello world
```

Mit dem Befehl

```
quit()
```

beenden Sie den Kommandozeilen-Interpreter wieder; sie sind nun wieder in der Umgebung Ihres Betriebssystems, wo Sie also *keine* Python-Befehle mehr eingeben können.

1.2. Script-Dateien

Mit dem letzten Abschnitt haben Sie den ersten Python-Befehl `print` kennen gelernt! In den Kommandozeilen-Interpreter eingegeben bewirkt er eine direkte Ausgabe auf dem Bildschirm. Sie werden sehr bald weitaus komplexere Programme schreiben, für die es umständlich wäre, sie Zeile für Zeile für jede Ausführung neu einzugeben.

Stattdessen können Sie ein *Textdokument* anlegen, in dem Sie alle Anweisungen nacheinander eingeben und speichern, und den Interpreter anschließend dazu auffordern, diese Datei zu lesen und den Code darin auszuführen. Wichtig hierbei ist, dass es sich wirklich um eine *reine Textdatei* handelt, dass also keine Formatierungen oder sonstigen Inhalte enthält, die über den reinen Code hinaus gehen. Verwenden Sie also zum Schreiben von Code *nicht* Programme wie Word, LibreOffice o. ä., sondern Code/Text-Editoren. Ich empfehle:

- für Linux
  - kate (KDE-Editor): auf die Arbeit mit vielen Programmiersprachen ausgelegt, sehr lightweight
  - gedit: oft vorinstalliert, minimale Features aber alles notwendige gegeben
  - geany: auf größere Projekte ausgelegt, aber immer noch hinreichend Ressourcen schonend
- für Windows
  - Notepad++: Bietet alle Funktionalitäten, die das Programmieren angenehm machen, ohne dabei zu viele Systemressourcen zu verbrauchen.
    Siehe `https://notepad-plus-plus.org/`
  - Notepad: Immer vorinstalliert. Die Arbeit mit diesem Programm ist oft mühselig, da Features wie Syntax Highlighting oder Automatische Einrückung nicht gegeben sind; dafür muss nichts installiert werden

In diesen (und etwa einer Million weiteren) Editoren können Sie Code verfassen und als `*.py`-Datei abspeichern. Aus der Kommandozeile können Sie diesen Code an den Interpreter weitergeben, indem Sie eingeben:

```
python3 [myCode].py
```

Wobei `[myCode]` selbstverständlich durch den von Ihnen vergebenen Dateinamen ersetzt werden muss.

1.2.1. Beispiel

Schreiben Sie den folgenden Code in eine Textdatei, und speichern Sie diese als `HelloWorld.py` ab. (Achten Sie auch auf Groß/Kleinschreibung).

Datei `HelloWorld.py`

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

Starten Sie eine Kommandozeilen-Umgebung, und wechseln in dieser in das Verzeichnis, unter dem Sie die Datei abgespeichert haben[2]. In diesem Fall sei der Code unter `~/Codes` abgelegt. Zum Ausführen dieses Codes geben Sie also ein:

Starten des Python-Interpreters und Hello-World

```
blue-chameleon@blue-chameleon:~$ cd Codes/
blue-chameleon@blue-chameleon:~/Codes$ python3 HelloWorld.py
Hello World!
```

1.3. IDEs

Neben der Arbeit mit Texteditoren, die vom Interpreter abgegrenzt stehen, existieren auch IDEs, also *Integrated Development Environments*. Es handelt sich dabei um Programme, die eine direkte Anbindung an den Interpreter haben und somit Code-Schreiben und Ausführen im selben Fenster erlauben.

Viele empfinden es als bequemer, mit IDEs zu arbeiten. Diese Programme sind oft aber auch etwas aufwändiger gebaut, und brauchen länger, bis sie geladen sind. Experimentieren Sie hier selbst, welcher Modus Ihnen am besten zusagt; für den Kurs sind beide Wege – Text-Editor und IDE – gangbare Wege.

Ich empfehle die IDE spyder3. Linux-User können diese einfach aus dem Paketverwaltungssystem heraus installieren; Windows-User mögen von `https://www.spyder-ide.org/` die Installationspakete herunterladen.

In Abbildung 1.1 sehen Sie die Arbeitsumgebung des Programms Spyder. Insbesondere finden Sie links einen größeren Bereich, in dem sie komplexere Codes schreiben können, wie schon in Abschnitt 1.2 angedeutet. In der rechten Fensterhälfte sehen Sie den Interpreter-Bereich, in den Sie direkt Python-Kommandos eingeben können. Genauso, wie in Abschnitt 1.1 gezeigt, werden die Befehle, die Sie hier eintippen, sofort ausgeführt.

1.4. Rechnen

Wie erwähnt können wir Python dazu benutzen, einfache Berechnungen ausführen zu lassen. Dazu tippen wir diese einfach direkt in die Interpreter-Umgebung ein:

Python als „Taschenrechner“

```
>>> 1 + 2
3
```

[2] Stichwort `cd`. Falls Sie nicht wissen, wie das geht, sprechen Sie bitte Ihre Dozenten an.

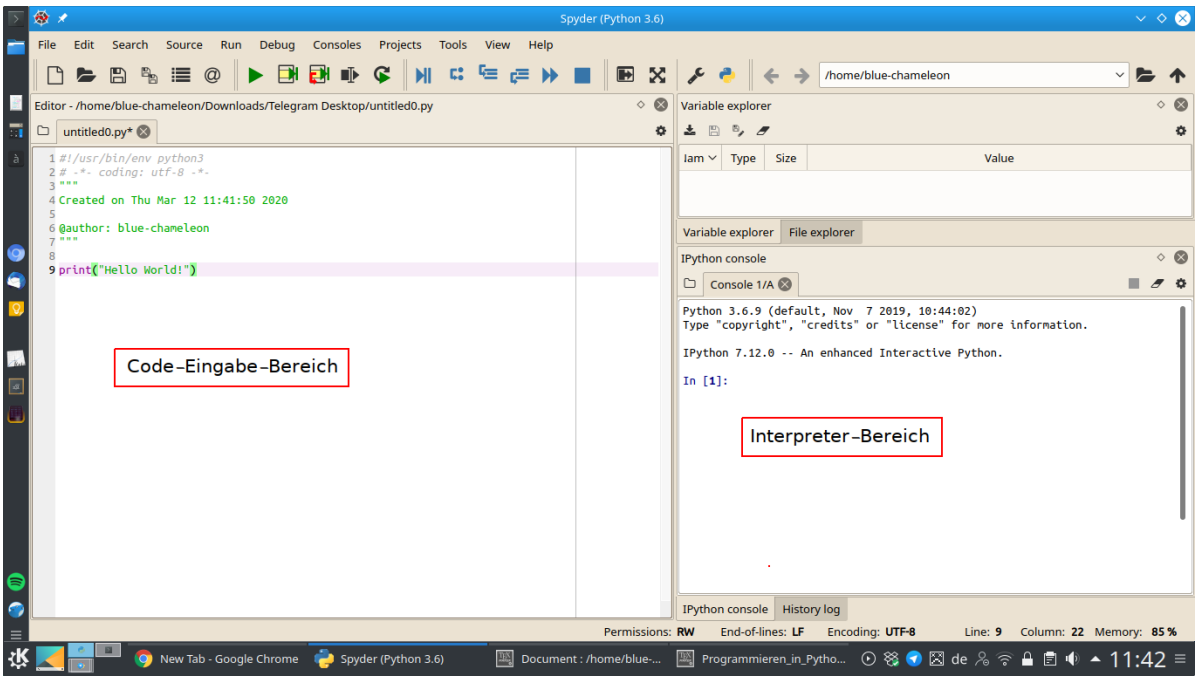


Abbildung 1.1.: Die IDE Spyder

Diese Rechnungen dürfen aus beliebig vielen Operationen bestehen, halten sich an die Regel „Punkt vor Strich“ und können auch Klammern enthalten:

Python als „Taschenrechner“

```
>>> ((1 - 5) * 3) / (4 + 1) ** 2
-0.48
```

Dabei werden folgende Zeichen als Operatoren verstanden:

Zeichen	Funktion	Beispiel
+	Addition	1 + 2 = 3
-	Subtraktion	5 - 7 = -2
*	Multiplikation	2 * 4 = 8
/	Division	7 / 5 = 1.4
//	Ganzzahl-Division	7 // 5 = 1
%	Modulo (Rest der Division)	7 % 5 = 2
**	Potenzierung	3 ** 2 = 9

Tabelle 1.1.: Rechenoperatoren in python3

Auch das Rechnen mit *komplexen Zahlen* ist möglich. Als imaginäre Einheit wird das Zeichen `j` verwendet:

Komplexe Zahlen

```
>>> (1j)**2
(-1+0j)
```

Wie Sie sehen, wird das Ergebnis von Rechnungen mit komplexen Zahlen als komplexe Zahl ausgegeben, selbst wenn die Zahl rein reell ist. Mehr dazu im Abschnitt 1.4.2.

1.4.1. Variablen

Die Ergebnisse einer Rechnung können in *Variablen* gespeichert werden. Es handelt sich hierbei um Speicherstellen, denen Sie einen mehr oder minder beliebigen Namen geben können:

Variable für Zwischenergebnisse

```
>>> x = 3 + 7
>>> x ** 2
100
```

Variablen können einzelne Buchstaben sein, dürfen aber auch ganze Worte zum Namen haben. Im Variablennamen dürfen auch der Unterstrich (_) und die Ziffern 0-9 vorkommen; jedoch muss das erste Zeichen ein Buchstabe sein. Python ist *case sensitive*, d. h. zwischen Groß- und Kleinschreibung wird unterschieden.

Name	Erlaubt	Begründung
`x`	ja	
`counter`	ja	
`CounTer`	ja	
`number_of_elements`	ja	
`list4`	ja	
`5th_list`	nein	Ziffer als erstes Zeichen
`list 4`	nein	Leerzeichen
`Best_List_Ever!`	nein	Rufezeichen
`print`	Problematisch	überschreibt den Befehl print

Tabelle 1.2.: Beispiele für Variablen in python3

Sprechende Variablennamen

Ihre Programme werden sehr bald einige Komplexität annehmen. Sie sollten daher Variablen so benennen, dass auf den ersten Blick erkennbar wird, welche Art Information gespeichert wird. Der Name `ListLength` ist in jedem Fall dem Namen `l` vorzuziehen.

Schlüsselworte

Die Liste oben nennt das Symbol `print` als erlaubten, aber problematischen Namen. Tatsächlich können Sie in Python die Sprachelemente umdefinieren, und so etwa `print` als Variable benutzen oder eine andere Routine unter diesem Namen aufrufen. Im Sinne von Lesbarkeit und Kompatiblität mit anderen Programmen sollten Sie hiervon aber Abstand nehmen! Wenn Sie `print` überschreiben, können Sie zunächst nichts mehr auf dem Bildschirm ausgeben.

Da Sie gerade erst beginnen, die Sprache Python zu erlernen, können Sie natürlich noch nicht alle Symbole kennen, die bereits vergeben sind. Wenn Sie einen Editor mit Syntax-Highlighting verwenden, können Sie aber i. d. R. diese farblichen Markierungen zur Hilfe nehmen: Wenn der Editor ihr Symbol wie einen Befehl markiert, sollten Sie es umbenennen. Wird keine besondere Farbe zugewiesen, so ist der Name vermutlich noch frei. Einige Schlüsselworte sind besonders geschützt, und können nicht überschrieben werden. Sie erhalten die Fehlermeldung `SyntaxError: invalid syntax`, falls Sie versuchen, ein solches Schlüsselwort als Variablenname zu verwenden.

Unterstriche in Variablennamen

Variablennamen dürfen prinzipiell an jeder Stelle Unterstriche enthalten, auch als erstes Zeichen. Es ist aber Konvention, dies nur in bestimmten Situationen zu tun, auf die ich an gegebener Stelle erst eingehen werde. Vermeiden Sie daher vorerst Namen wie `_var`.

Non-ASCII-Variablennamen (Umlaute, Sonderzeichen, ...)

Python 3 erlaubt es prinzipiell, Variablennamen aus dem UTF-8-Zeichenvorrat zu wählen. Das bedeutet, dass neben den lateinischen Klein- und Großbuchstaben (a-z, A-Z) auch Umlaute, Zeichen mit Akzenten, griechische, japanische, ... Zeichen für Variablennamen erlaubt sind. Dies führt aber schnell zu Kompatibilitätsproblemen. Neben den offensichtlichen Problemen – Kollegen in anderen Ländern könnten die Schriftzeichen, die zur Bedienung Ihres Codes nötig sind, nicht eingeben können – ist manchmal schon das Versenden und Ausführen von Code auf einem anderen Rechner in derselben Arbeitsgruppe schwierig.

Während Python also Variablennamen wie `äußerstWichtig` durchaus erlaubt, sollten Sie also dennoch nur auf den englischen Zeichenvorrat zurückgreifen, und eine Variable beispielsweise `exceptionallyImportant` benennen.

Variablen können aktualisiert (d. h. überschrieben werden). Dies kann auch mit Bezug auf den alten Wert derselben Variable geschehen:

Variable für Zwischenergebnisse

```
>>> x = 1
>>> x
1
>>> x = 2
>>> x
2
>>> x = x + 1
>>> x
3
```

Python ist kein Gleichungslöser

Für mathematisch denkende KursteilnehmerInnen mag die Zeile `x = x + 1` unsinnig wirken. Offensichtlich gibt es keine Zahl `x`, die diese Gleichung erfüllt. In Python beschreiben wir auf diese Art aber auch keine Gleichung, sondern einen Arbeitsauftrag: Speichere in der Variable `x` den Wert der Summe des aktuellen Werts von `x` plus 1!

Denken Sie an das Vorwort bei der Arbeit mit Computern: Maschinen verstehen komplexe Aufgaben wie das Lösen eines Gleichungssystems nicht.

Shorthands

Das Aktualisieren eines Werts unter Bezug auf den alten Wert ist ein häufiger Arbeitsschritt beim Programmieren. Daher wurden Abkürzungen (Shorthands) eingeführt. So steht zum Beispiel der Ausdruck

```
x += 1
```

für den Code

```
x = x + 1
```

Ähnlich sind auch `x -= y`, `x *= y`, usw. erlaubt.

Die Werte von Variablen können in einem Schritt getauscht werden, indem wir ein Komma zur Hilfe nehmen:

Variable für Zwischenergebnisse

```
>>> x = 1
>>> y = 2
>>> x, y = y, x
>>> x
2
>>> y
1
```

Ein *Dreieckstausch* (d. h. die Zuhilfe-Nahme einer dritten Variable) ist nicht nötig. (Intern führt Python einen Dreieckstausch aus; dies wird aber automatisch für Sie erledigt, ohne weiteres zutun Ihrerseits).

1.4.2. Datentypen

In diesem Abschnitt arbeiten wir mit Zahlen. Für Sie als Mensch ist eine Zahl eindeutig durch ihren Wert bestimmt: $1 = 1.0 = 1 + 0j =$ eins. Ein Computer „kennt“ aber zunächst keine Werte, sondern nur binäre Information. Einer Folge von Einsen und Nullen kann nicht angesehen werden, ob diese jetzt eine ganze Zahl, eine komplexe Zahl, einen Teil eines Bildes oder eine Anweisung eines Computerprogramms darstellen. Daher wird jeder Information ein *Datentyp* zugeordnet, also eine Anweisung, wie die Folge von Einsen und Nullen zu interpretieren ist.

Vorerst beschäftigen wir uns mit vier Datentypen:

- `int` – Ganzzahlen, also `0`, `1`, `2`, `3`, ..., sowie negative Ganzzahlen
- `float` – Fließkommazahlen, also `3.14` oder `1.0`. (Beachten Sie: als Dezimaltrennzeichen verwenden wir einen *Punkt*, kein Komma.)

- `complex` – Komplexe Zahlen, also `(1+2.7j)`
- `str` – Strings, also Zeichenketten wie `"Hallo Welt"`

Der Datentyp eines Werts wird bei seiner „Berechnung“ festgelegt und zusammen mit der Variablen gespeichert, über die der Wert Verfügbar gehalten wird. Dabei gilt die Grundregel, dass keine Information verloren gehen darf. Bei der Addition einer Ganzzahl (`int`) und einer Fließkommazahl (`float`) darf beispielsweise die Information über die Nachkomma-Anteil nicht verloren gehen (selbst, wenn dieser `.0` ist). Betrachten Sie hierzu folgendes Beispiel:

Datentypen bei Addition

```
>>> 1 + 1
2
>>> 1.0 + 1
2.0
>>> 1 + 1.0
2.0
>>> 1.0 + 1.0
2.0
```

In der ersten Zeile (`1 + 1`) sind nur Ganzzahlen beteiligt. Somit ist das Ergebnis auch eine Ganzzahl, und wird folgerichtig als solche (d. h. ohne Nachkommastelle) ausgegeben. In allen anderen Fällen ist immer mindestens eine Fließkommazahl beteiligt; das Ergebnis ist daher immer `2.0` (nicht nur `2`).

Bei der Division wird immer eine Fließkommazahl berechnet, egal ob die Argumente vom Typ `int` oder `float` sind.

Ähnlich verhält es sich bei komplexen Zahlen: Sobald eine komplexe Zahl an der Rechnung beteiligt ist, wird auch das Ergebnis vom Typ `complex` sein. Da `complex`-Werte auch Nachkomma-Werte speichern können, übertrifft diese Regel die bzgl. `float`s.

Wenn Sie sich nicht sicher sind, welchen Datentyp eine Variable hat, können Sie den Befehl

```
type(Ausdruck)
```

benutzen. Dabei steht `Ausdruck` für eine Variable, eine Zahl oder eine komplette Rechnung.

Beispiele zu `type` (1)

```
>>> type(1)
<class 'int'>
>>> type(1.0)
<class 'float'>
>>> type(1j)
<class 'complex'>
>>> type(1+1.0)
<class 'float'>
>>> a=1j**2
>>> type(a)
<class 'complex'>
>>> a
(-1+0j)
```

Wollen Sie erzwingen, dass das Ergebnis in einen bestimmten Typ umgewandelt wird, so können Sie den Datentyp vor einen Ausdruck setzen, und diesen einklammern:

```
Datentyp(Ausdruck)
```

Dabei können Informationen verloren gehen:

Beispiele zu `type` (2)

```
>>> a = int(1 + 1.9)
>>> type(a)
<class 'int'>
>>> a
2
```

in diesem Beispiel etwa wird der Nachkommaanteil abgeschnitten.

Wie bereits erwähnt, sind Strings (Zeichenketten) dadurch erkennbar, dass ihr Inhalt durch doppelte Anführungszeichen ("...") vom restlichen Code abgegrenzt wird. Alternativ können auch einfache Anführungszeichen ('...') verwendet werden. Dies hat den Zweck, es Programmierern einfach zu machen, Strings zu Erzeugen, in denen auch selbst wieder Anführungszeichen vorkommen.

Beispiele zu Strings (1)

```
>>> 'abc'
'abc'
>>> "abc"
'abc'
>>> "abc'def'ghi"
"abc'def'ghi"
>>> 'abc"def'
'abc"def'
```

Auch mit Strings kann „gerechnet" werden; hier sind jedoch nur die Addition (+) und die Multiplikation (∗) mit Ganzzahlen definiert. Die Addition verkettet zwei Strings; die Multiplikation wiederholt einen String mehrere Male:

Beispiele zu Strings (2)

```
>>> "ab" + 'cd'
'abcd'
>>> 3 * "ab"
'ababab'
>>> 0 * "ab"
''
>>> "ab" + "'c'def"
"ab'c'def"
```

Die Befehle `int`, `float`, `complex` können – in begrenztem Maße – auch auf Strings angewandt werden:

Konversion von Strings zu Zahlentypen

```
>>> int("1")
1
>>> int("1.3")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1.3'
>>> float("1.3")
1.3
>>> float("1,3")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: '1,3'
>>> complex("1j")
1j
>>> int("one")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'one'
```

Wie Sie sehen, wird die Darstellung als Text in Zahlen zurückverwandelt, sofern das für den gewünschten Zieltyp möglich ist; andernfalls erhalten Sie eine Fehlermeldung.

Machen Sie sich klar: Für den Computer sind Text und Zahlen unterschiedliche Informationen! Eine *semantische* Interpretation ist nicht möglich. Machen Sie sich daher auch klar, was der Unterschied zwischen diesen drei Additionen ist:

Strings und `int`s (1)

```
>>> x = "1"
>>> y = "2"
>>> x + y
'12'
>>> int(x + y)
12
>>> int(x) + int(y)
3
```

Wir beginnen mit den *String*-Variablen `x` und `y`. Die Addition von Strings ist gleichbedeutend mit der Verkettung; daher ist das Ergebnis von `x + y` auch folgerichtig der *String* `"12"`.

Der Aufruf von `int` in `int(x + y)` erhält als Argument den Wert `x + y`, also den *String* `"12"`. Folgerichtig wird die *Zahl* `12` berechnet.

Im dritten Teilbeispiel `int(x) + int(y)` dagegen werden separat die Zahlen `1` und `2` aus den Variablen `x` und `y` berechnet, und diese dann addiert. Entsprechend kann erst hier das Ergebnis die Zahl `3` sein.

Natürlich können Sie auch beliebige Zahlen in Strings umwandeln; dazu verwenden Sie einfach den Befehl `str`:

Strings und `int`s (2)

```
>>> x = 1
>>> y = 2
>>> str(x + y)
'3'
>>> str(x) + str(y)
'12'
```

Sprechweise: *dynamische Typisierung* und *duck-typing*

In vielen Programmiersprachen wird der Datentyp von Variablen einmal festgelegt und darf sich dann für das weitere Programm nicht mehr ändern. Python dagegen erlaubt *dynamische Typisierung*: Eine Variable `x` kann an einer Stelle des Programms Ganzzahlen speichern und an späterer Stelle Strings. Damit einher geht eine gewisse Ambivalenz; es ist nicht zwingend sofort einsichtig, welchen Datentyp ein Ausdruck hat. Der Python-Interpreter versucht dann, den geeignetsten Datentyp zu „erraten“. Gemäß dem Zitat von James Whitcomb Riley:

Zitat

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

wird Python daher als *duck typed language* bezeichnet.

1.4.3. Ausgabe von Variablen mit `print`

Um den in einer Variablen gespeicherten Wert zu erfahren, haben wir bisher in der Interpreter-Umgebung den Namen der Variable eingegeben. In längeren Codes, wie wir sie im Code-Eingabe-Bereich schreiben, funktioniert dies aus technischen Gründen leider nicht. Stattdessen können wir aber den `print` benutzen. Betrachten Sie das folgende Beispiel:

Beispiel: Ausgabe von Werten mit `print`

```
1  a = 2
2  b = a * 7.5 + 2
3  print(a, b, "konstanter Text", a * "x")
```

Ausgabe: Ausgabe von Werten mit `print`

```
2 17.0 konstanter Text xx
```

Sie erkennen hieraus, dass Sie der Befehl `print` die *Werte der Ausdrücke, die als Argumente übergeben werden*, ausgibt. Das bedeutet, dass `a` eben durch seinen Wert (hier also durch `2`) ersetzt wird. Ich erinnere Sie nochmals daran, dass dies der Grund ist, warum Strings in Anführungszeichen eingefasst werden müssen – sonst könnte der Interpreter die Anweisung *drucke den Buchstaben a* und die Anweisung *drucke den Wert der Variablen* `a` nicht auseinander halten.

Weiter sehen Sie, dass `print` nicht nur einen einziges Argument verarbeiten kann, sondern auch mit einer ganzen *Parameterliste* zurecht kommt. Die einzelnen Ausdrücke werden durch Kommata gelistet aufgelistet und der Reihe nach ausgewertet, bevor sie auf dem Bildschirm erscheinen. Diese Ausdrücke

dürfen einzelne Variablen (`a`, `b`), Konstanten (`3.14`, `"konstanter Text"`) oder komplette „Rechnungen“ (`a * "x"`) sein.

1.5. Kommentare und mehrzeilige Kommandos

Wie Sie bald sehen werden, können Codes lang und komplex werden. Es wird Ihnen helfen, schwer erfassbare Abschnitte durch Fließtext-Kommentare zu ergänzen. Solche Kommentare markieren Sie durch ein Raute-Zeichen (#). Der Interpreter wird alle Zeichen hinter dem Kommentar-Zeichen bis zum Zeilenende ignorieren.

Beispiel: Kommentare

```
print("Normaler Code, der ausgeführt wird")  # Dies wird nicht mehr ausgeführt
#print("auch dies wird nicht ausgeführt")
```

Normalerweise enden Python-Anweisungen mit dem Zeilenumbruch. An manchen Stellen kann es Ihren Code übersichtlicher machen, Anweisungen auf mehrere Zeilen zu verteilen. Dass eine Anweisung trotz Zeilenumbruch über das Zeilenende gelesen werden soll, erreichen Sie, indem Sie einen Backslash (\) setzen:

Beispiel: Mehrzeilige Anweisungen

```
x = 1
a = 2 + \
    3 * x + \
    4 * x**2 + \
    7 * x**3
```

1.6. Formatierte Strings

Wir können Strings erzeugen, in denen die Werte von Variablen als Text dargestellt werden. Zu diesem Zweck haben wir bereits die Funktion `str` kennengelernt. Wir wissen auch, dass wir Strings durch die Addition verketten können. Ein bequemerer Weg kann über *Format-Strings* erreicht werden:

Syntax: Format-String

```
f"normaler Text {Ausdruck} mehr normaler Text {weiterer Ausdruck:Format} ..."
```

Ein Format-String beginnt also mit einem vorangestellten `f`, und wird ebenso von doppelten Anführungszeichen `""` eingeschlossen, wie ein normaler String. Er kann – muss aber nicht – beliebig lange Blocks von Text enthalten, die 1:1 in das Endergebnis übernommen werden. Neu gegenüber normalen Strings sind Blöcke der Form `{Ausdruck}` und `{Ausdruck : Format}`.

Wie schon zuvor auch steht `Ausdruck` für eine Variable, eine Zahl oder eine komplette Rechnung. Der Ausdruck wird zuerst *evaluiert* („ausgerechnet“), und dann in den String eingebaut. Die {geschweiften Klammern} sind nicht Teil des Endprodukts, sondern zeigen dem Interpreter an, dass hier eine Ersetzung gemacht werden muss.

Beispiel: Formatstrings

```
a = 1
b = 2
s = f"a + b = {a + b}"
print(s)
```

Ausgabe: Formatstrings

```
a + b = 3
```

Dieser Code ist im Ergebnis gleichwertig zu

Beispiel: Gleichwertiger Code ohne Formatstrings

```
a = 1
b = 2
s = "a + b = " + str(a + b)
print(s)
```

Wenn Sie tatsächlich {geschweifte Klammern} im Ergebnis brauchen, so erreichen sie dies, indem Sie im Formatstring ein doppeltes Klammerpaar setzen:

Beispiel: Formatstrings mit Escape-Sequenz

```
a = 1
b = 2
s = f"{{a + b}} = {{{a + b}}}"
print(s)
```

Ausgabe: Formatstrings mit Escape-Sequenz

```
{a + b} = {3}
```

Im Syntax-Kasten wurde bereits angedeutet, dass abgetrennt durch einen Doppelpunkt noch weitere Angaben zur Formatierung folgen dürfen. In Abschnitt B.2 finden Sie eine Übersicht der unterstützten Formatzeichen. Hier seien nur einige besonders nützliche Beispiele gezeigt:

Die einfachste Formatvorgabe, die sie setzen können, ist eine Zahl. Diese Zahl gibt dann an, wie viele Zeichen zur Darstellung des Ausdrucks verwendet werden sollen. Auf diese Weise können Sie bequem tabellarische Ansichten erstellen:

Beispiel: Formatstrings mit Vorgabe der Zeichenlänge

```
name1 = "Dusky"
score1 = 9001
name2 = "Joe"
score2 = 666

print(f"{name1:20}: {score1:5}")
print(f"{name2:20}: {score2:5}")
```

Ausgabe: Formatstrings mit Vorgabe der Zeichenlänge

```
Dusky               :  9001
Joe                 :   666
```

Beachten Sie, dass zwischen dem Doppelpunkt und dem Format *kein* Leerzeichen stehen darf (bzw. dass ein solches eine besondere Funktion hat – siehe weiter unten)

Wie Sie sehen, werden Strings linksbündig ausgegeben, während Zahlen rechtsbündig formatiert werden. Diese Standard-Einstellung kann durch ein vorangestelltes <, > oder ^ überschrieben werden:

Beispiel: Formatstrings und Alignment

```
1  text  = "sample"
2  value = 123
3  print( f"|{text:15}| |{value:15}|" )
4  print(f"|{text:<15}| |{value:<15}|")
5  print(f"|{text:>15}| |{value:>15}|")
6  print(f"|{text:^15}| |{value:^15}|")
```

Ausgabe: Formatstrings mit Vorgabe der Zeichenlänge

```
|sample         | |            123|
|sample         | |123            |
|         sample| |            123|
|    sample     | |      123      |
```

Ist der Ausdruck zu lang, um mit der vorgegebenen Zeichenzahl gedruckt zu werden, so ignoriert Python die Zeichenzahl und druckt den vollen Text. Durch einen Punkt vor der Zahl bringen Sie Python dazu, stattdesen die Ausgabe abzuschneiden. Dies funktioniert jedoch nur bei Strings, und kann nicht mit den Zeichen <, > oder ^ kombiniert werden:

Beispiel: Formatstrings und String-Truncation

```
1  text = "very long sample text"
2  print( f"|{text:.4}|" )
```

Ausgabe: Formatstrings und String-Truncation

```
|very|
```

Natürlich können die Effekte durch Hilfsvariablen dennoch kombiniert werden:

Beispiel: Kombination von Formatierungen über Hilfsvariablen

```
1  value = 1234567890
2  step1 = f"{value}"          # zu String
3  step2 = f"{step1:.5}"       # Länge beschränken
4  final = f"|{step2:^10}|"    # zentrieren
5  print(final)
```

Ausgabe: Formatstrings und String-Truncation

```
|  12345   |
```

Bei *Zahlen* dient ein vorangestelltes Leerzeichen im Formatstring als Platzhalter für ein eventuelles Vorzeichen. Alternativ kann auch ein Pluszeichen (+) gesetzt werden, um anzudeuten, dass das Vorzeichen *immer* Teil des Ergebnisses sein soll, selbst wenn die Zahl positiv ist:

Beispiel: Formatstrings und Vorzeichen

```
pos =  10
neg = -10
print(f"|{pos: 15}| |{neg: 15}|")
print(f"|{pos:+15}| |{neg:+15}|")
```

Ausgabe: Formatstrings und Vorzeichen

```
|             10| |            -10|
|            +10| |            -10|
```

Eine vorangestellte `0` füllt den zur Verfügung gestellten Platz mit Nullen auf. Dies ist mit Leerzeichen und Pluszeichen kombinierbar:

Beispiel: Formatstrings und führende Nullen

```
pos =  10
neg = -10
print(f"|{pos: 015}| |{neg: 015}|")
print(f"|{pos:+015}| |{neg:+015}|")
```

Ausgabe: Formatstrings und führende Nullen

```
| 000000000000010| |-00000000000010|
|+00000000000010| |-00000000000010|
```

Speziell für Fließkommazahlen gibt es das Zeichen `f`, das eine Steuerung der Anzeige von Nachkommastellen ermöglicht:

Beispiel: Formatstrings und Fließkommazahlen

```
num = 1.2
print(f"|{num}|")
print(f"|{num:f}|")
print(f"|{num:6.2f}|")
print(f"|{num:<6.1f}|")
print(f"|{num:06.2f}|")
print(f"|{num:+06.2f}|")
print(f"|{num: 06.2f}|")
```

Ausgabe: Formatstrings und Fließkommazahlen

```
|1.2|
|1.200000|
|  1.20|
|1.2   |
|001.20|
|+01.20|
| 01.20|
```

Ein einzelnes `f` stellt die Zahl mit 6 Nachkommastellen dar, und füllt gegebenenfalls mit Nullen auf, falls weniger Dezimalstellen zur Zahl gehören. In der Form `x.yf` werden insgesamt `x` Zeichen zur Darstellung der Zahl bereitgestellt (Komma und Vorzeichen mitgezählt). Die Zahl wird mit `y` Nachkommastellen ausgegeben und gegebenenfalls mit Nullen aufgefüllt. Dies ist kombinierbar mit den Alignment-Zeichen `<`, `>` und `^`. Auch führende Nullen, Leerzeichen oder erzwungenes Vorzeichen funktionieren wie oben beschrieben.

1.7. Obfuscated Code

Sie werden feststellen, dass es zu einer Aufgabe sehr viele *funktionierende* Lösungen gibt. Dass Code funktioniert, reicht uns aber nicht. Code soll auch leicht lesbar und verständlich sein. Bedenken Sie: die Aufgaben, die Sie hier lösen, sind in der Regel kein Selbstzweck, sondern Bausteine für größere Projekte. Wenn die einzelnen Teillösungen schwer zu verstehen sind, werden sie auch umso beschwerlicher als Lösung in andere Probleme einbaubar sein.

Das folgende Beispiel:

Beispiel: Unlesbarer Code

```
1  p = lambda x: int(( -13214 * x**11 + 956318 * x**10 - 30516585 * x**9 +
2                       564961485 * x**8 - 6717043212 * x**7 + 53614486464 * x**6
3                      -291627605005 * x**5 + 1074222731065 * x**4
4                      -2606048429424 * x**3 + 3927289106268 * x**2
5                      -3265905357360 * x + 1116073728000 ) / 19958400)
6
7  print (bytearray(map(p, range(1, 13))).decode())
```

(Quelle: `https://codegolf.stackexchange.com/questions/22533/weirdest-obfuscated-hello-world`) gibt ebenso den Text `Hello World!` auf dem Bildschirm aus, ist aber (auch für Profis) kaum so zu verstehen. Nehmen Sie sich daher die Hinweise zu Best Pratice zu Herzen, die Ihnen in diesem Script mitgegeben werden.

2. Usereingaben und Entscheidungen

> We must believe in free will – we have no choice.
>
> Isaac Bashevis Singer

Im vorigen Kapitel lief unser Programm wie auf Schinen: einmal in Gang gesetzt konnte es nicht von dem vorgeschriebenen Pfad abweichen, das Ergebnis jeder Zeile war vorherbestimmt. In diesem Kapitel wollen wir dies auflockern, indem wir Eingaben des Benutzers annehmen und darauf reagieren.

2.1. Usereingaben – `input`

Die Funktion `input` kann benutzt werden, um Eingaben von der Tastatur zu lesen. Diese werden als String gespeichert und können an eigene Variablen übergeben werden. Optional kann ein *Prompt* angezeigt werden, also ein Text, der den User zur Texteingabe auffordert.

Syntax: `input`

```
variable = input(prompt)
```

In dieser Zeile erkennen Sie ein Schema, das Sie beim Programmieren in Python (und in vielen anderen Sprachen) täglich benutzen werden:

Eine *Funktion* wird aufgerufen, indem ihr Name genannt wird, und in Klammern dahinter eine *Parameterliste* übergeben wird. Dies ist eine durch Kommata getrennte Liste von Werten, wie sie es schon von `print` kennen. Die Funktion berechnet auf Basis der Argumente in der Parameterliste einen Wert, der wiederum in einer (oder mehreren) Variablen gespeichert oder auch verworfen werdem kann. Neben der Berechnung von Werten können Funktionen auch Nebeneffekte haben, wie etwa die Ausgabe von Text auf dem Bildschirm[1].

Beispiel: `input`

```
1 inText   = input("Bitte geben Sie eine Zahl ein: ")
2 inNumber = float(inText)
3 print("Das Quadrat dieser Zahl ist", inNumber ** 2)
```

Ausgabebeispiel `input` (1)

```
Bitte geben Sie eine Zahl ein: 1.5
Das Quadrat dieser Zahl ist 2.25
```

[1] `print` ist eine Funktion. Sie hat den „Nebeneffekt“, dass Text ausgegeben wird, und den Rückgabewert `None`. Mehr dazu in Kapitel 6.

Ausgabebeispiel `input` (2)

```
Bitte geben Sie eine Zahl ein: eins
Traceback (most recent call last):
  File "inputTest.py", line 2, in <module>
    inNumber = float(inText)
ValueError: could not convert string to float: 'eins'
```

Tipp: `input` beim Entwickeln auskommentieren

Sie werden feststellen, dass es einige Zeit dauert, bis Ihr Code das tut, was Sie möchten. Erst nach einigen Ausführungen des Codes werden Sie alle Fehler gefunden und behoben haben. Wenn Sie für jeden Test neu manuell die Eingaben tätigen müssen, die Sie von Ihrem End-User schließlich verlangen, wird Sie das beim Entwickeln Ihres Codes ausbremsen.

Es bietet sich daher an, `input`-Zeilen auszukommentieren und stattdessen konstante Werte vorzugeben. Schreiben Sie zum Beispiel:

`input` auskommentieren

```
1 inText   = "1.5" # input("Bitte geben Sie eine Zahl ein: ")
2 inNumber = float(inText)
3 print("Das Quadrat dieser Zahl ist", inNumber ** 2)
```

bis Sie sich sicher sind, dass die Codezeilen, die Ihrer Eingabe folgen, tatsächlich Ihre Wünsche erfüllen. Vergessen Sie danach nicht, die Eingabemöglichkeit wiederherzustellen!

2.2. Bedingte Codeausführung

Da wir nun eine Möglichkeit gefunden haben, Eingaben des Users in unseren Programmablauf einzuführen, ist das Ergebnis unseres Programmes nicht mehr schon vor der Ausführung vorherbestimmt. Wir wollen darauf aufbauen, und abhängig von Eingaben Entscheidungen treffen: Einzelne Programmteile sollen nur dann ausgeführt werden, wenn eine *Bedingung* erfüllt ist. Hierzu jedoch zunächst etwas Theorie:

2.2.1. Booleans – Wahrheitswerte

Mathematischen Ausdrücken können *Wahrheitswerte* zugeordnet werden. Diese Wahrheitswerte sagen aus, ob es sich um eine „wahre“ oder „falsche“ Aussage handelt. Ein Beispiel für einen solchen Ausdruck ist:

```
1 + 5 * 8 + 1 == 42
```

Der Wahrheitswert dieses Ausdrucks ist offensichtlich *wahr* (bzw. `True`).

Beachten Sie bitte, dass für den *Vergleich* zweier Zahlen das *doppelte Gleichheitszeichen* == verwendet wird. Das einfache Gleichheitszeichen = dient ausschließlich der *Wertzuweisung* an Variablen.

Tabelle 2.1 listet weitere Vergleichsoperatoren auf.

Vergleich	Zeichen	Vergleich	Zeichen
Gleichheit	`==`	Ungleichheit	`!=` oder `<>`
Kleiner als	`<`	Größer als	`>`
Kleiner oder gleich	`<=`	Größer oder gleich	`>=`

Tabelle 2.1.: Vergleichsoperatoren in python3

Das Ergebnis eines Vergleichs ist selbst eine Information, die in einer Variablen gespeichert werden kann. Der *Datentyp* eines solches Vergleichs nennt sich `bool`[2]. Ein solcher Boolean kann nur entweder `True` oder `False` sein.

Theoretisch reicht also ein einziges Bit, um die Information `True`/`False` zu kodieren. Aus technischen Gründen können jedoch nur Gruppen zu mindestens 8 Bit – also 1 Byte – verarbeitet werden. Intern werden `bool`s daher als `int`s abgelegt und anders herum `int`s wie `bool`s behandelt. Als `True` gilt dabei jeder Wert, der von `0` verschieden ist. Sie können Booleans also auch als eine besondere Darstellungsform von `int`s verstehen.

Mit diesen *Booleans* kann nun auch wieder gerechnet werden. Wenig sinnvoll (aber durchaus möglich) sind die Grundrechenarten (Addition, Subtraktion, ...). Hier zählt `True` als der `int 1` während `False` wie eine `0` behandelt wird.

Interessanter dagegen sind *logische Verknüpfungen*: Wir können uns eine Gesamtaussage vorstellen, die aus zwei oder mehreren Teilaussagen besteht. So kann die Gesamtaussage *Es ist angenehm warm* dann erfüllt (`True`) sein, wenn sowohl die Teilaussage *Es ist wärmer als 20°C*, ALS AUCH die Teilaussage *Es ist kälter als 25°C* erfüllt ist. Formal wird dies mit dem *logischen Operator* `and` ausgedrückt: Nur wenn beide Booleans `True` sind, ist das Ergebnis von `and` auch `True`. Folgendes Beispiel zeigt Ihnen die Anwendung des logischen Operators `and`:

Beispiel: `and`

```
1  temperature = float(input("Bitte geben Sie die aktuelle Temperatur ein: "))
2
3  warmEnough = temperature > 20
4  coldEnough = temperature < 25
5
6  pleasantTemperature = warmEnough and coldEnough
7
8  print("Die Aussage 'es ist angenehm warm' ist", pleasantTemperature)
```

Ausgabebeispiel `and`

```
Bitte geben Sie die aktuelle Temperatur ein: 18
Die Aussage 'es ist angenehm warm' ist False
```

Hier wird `warmEnough` zu `False` ausgewertet, während `coldEnough` den Wert `True` zugewiesen bekommt. Das Ergebnis von `False and True` ist nach Boole'scher Algebra wiederum `False`.

Beachten Sie, dass die `and`-Zeile auch ohne Hilfsvariablen auskommt:

```
pleasantTemperature = (temperature > 20) and (temperature < 25)
```

[2]Häufig spricht man auch von *Booleans*, benannt nach dem englischen Mathematiker George Boole.

wird genauso ausgewertet wie der oben gezeigte Code.

Neben dem logischen Und (`and`) gibt es auch das logische Oder (`or`). Auch hier werden zwei Teilaussagen zu einer Gesamtaussage verknüpft. Beim Oder reicht es aber, wenn nur wenigstens eine Aussage `True` ist. Ich kann also die Idee *Ich habe keine Probleme, wenn mir jemand zur Seite steht, oder wenn ich keine Hose trage*[3] im Code ausdrücken als:

```
hakunaMatata = somebodyHelpsMe or notWearingPants
```

Dabei wird `hakunaMatata` sowohl `True`, wenn nur eine der beiden Wahrheitswerte `somebodyHelpsMe` und `notWearingPants` gleich `True` sind, oder auch, wenn *beide* den Wert `True` haben.

Schließlich existiert noch das logische Nicht (`not`), das einen Wahrheitswert einfach in sein Gegenteil verkehrt. `not True` ist `False`, und `not False` ist `True`.

2.2.2. `if`-Blöcke

Wir können dafür sorgen, dass ein Teil des Codes nur dann ausgeführt wird, wenn ein Wahrheitswert gleich `True` ist. Dazu verwenden wir das Schlüsselwort `if`:

Syntax: `if` (1)

```
normaler_Code

if Wahrheitswert :
    Anweisungen

normaler_Code
```

`Anweisungen` sind beliebige Python-Befehle, wie Sie schon einige kennen gelernt haben, und wie sie noch weitere dazu lernen werden; dasselbe gilt für `normaler_Code`. Beides können einzeilige Befehle sein oder auch längere Anweisungsblocks.

Während `normaler_Code` *immer* ausgeführt wird, beachtet der Python-Interpreter die `Anweisungen` des `if`-Blocks nur, wenn `Wahrheitswert` gleich `True` ist. Zum Block `Anweisungen` gehört aller Code, der *eingerückt* ist, der also „nach rechts“ verschoben wurde.

Wie richtig Einrücken?

Python-Code darf *entweder* durch Leerzeichen *oder* durch Tabulatoren eingerückt werden, nicht jedoch durch eine Mischung aus beiden. Leerzeichen werden empfohlen. Die Anzahl der Leerzeichen je Einrückungsebene kann frei gewählt werden, muss aber über das gesamte Code-Dokument einheitlich gehalten werden. Empfohlen werden vier Leerzeichen pro Ebene.

Viele Code-Editoren setzen automatisch Leerzeichen, wenn die Tabulator-Taste gedrückt wird, und erlauben so ein komfortables Arbeiten. Einige erkennen sogar automatisch Block-Strukturen wie `if`, und machen automatisch eine Einrückung, wenn nach dem Doppelpunkt Enter gedrückt wird.

Das `if` kann zu einer *wenn-sonst*-Struktur ausgeweitet werden, indem ein `else`-Block angefügt wird:

[3] frei nach der Regensburger Philosophin M. Mühlbauer, die mit dem Credo *Keine Hose – keine Probleme* schon manchen zum Lachen gebracht hat.

Syntax: `if` (2)

```
normaler_Code

if Wahrheitswert :
    Wenn_Anweisungen
else :
    Sonst_Anweisungen

normaler_Code
```

Wie zu erwarten, wird der Block `Wenn_Anweisungen` nur ausgeführt, wenn `Wahrheitswert` gleich `True` ist. Andernfalls (und nur dann!) befolgt der Python-Interpreter den Block `Sonst_Anweisungen`.

Beispiel: `if` und `and`

```
1 temperature = float(input("Bitte geben Sie die aktuelle Temperatur ein: "))
2
3 if (temperature > 20) and (temperature < 25) :
4     print("Es ist angenehm warm.")
5 else :
6     print("Es ist entweder zu heiß oder zu kalt.")
```

Schließlich kann diese Struktur nochmals erweitert werden, um mehrere Bedingungen aneinander zu reihen. Dazu verwenden wir das Schlüsselwort `elif`, was für `else if` steht:

Syntax: `if` (3)

```
normaler_Code

if Wahrheitswert_1 :
    Anweisungen_1
elif Wahrheitswert_2 :
    Anweisungen_2
elif ...
    ...
else :
    Sonst_Anweisungen

normaler_Code
```

Das `else` ist hierbei optional. Es dürfen beliebig viele `elif`-Blöcke aufgereiht werden.

Beachten Sie: von den `Anweisungen`-Blocks in dieser Struktur wird nur ein einziger ausgeführt! Python prüft zuerst `Wahrheitswert_1`. Wenn dieser `True` war, wird `Anweisungen_1` ausgeführt und danach *sofort* mit `normaler_Code` fortgesetzt, unabhängig davon, ob `Wahrheitswert_2` oder andere auch wahr wären. Die Liste der Bedingungen wird von oben nach unten abgearbeitet und nur *der erste erfüllte Bedingungsblock* wird ausgeführt.

Betrachten Sie dazu folgenden Code:

Beispiel: `elif`

```
temperature = float(input("Bitte geben Sie die aktuelle Temperatur ein: "))

if   temperature < 20 :
    print("Es ist zu kalt.")
elif temperature > 25 :
    print("Es ist zu heiß")
else :
    print("Es ist angenehm warm.")
```

Das **and**, das wir zuvor gebraucht haben, ist nun schon in der Anordnung der Bedingungen absorbiert. Die Prüfung `temperature > 25` wird nur dann überhaupt ausgeführt, wenn `temperature < 20` bereits **False** war.

Durchdenken Sie bitte auch dieses *fehlerhafte* Beispiel:

Fehlerhafter Code: `elif`

```
i = 23

if   i < 25 :
    print("Diese Zahl ist kleiner als 25.")
elif i > 20 :
    print("Diese Zahl ist größer als 20.")
```

Die Zahl 23 erfüllt zwar beide Bedingungen (`i < 25` und `i > 20`). Dennoch wird nur die Ausgabe `Diese Zahl ist kleiner als 25.` erfolgen, da die zweite Prüfung (`i > 20`) nie durchgeführt wird.

if-Blöcke können auch ineinander verschachtelt werden:

Beispiel: Verschachtelte `if`-Blöcke

```
i = int(input("Bitte geben Sie eine Ganzahl ein: "))

if i % 2 == 0 :
    if   i > 100 :
        print(i, "ist eine große gerade Zahl.")
    elif i >   0 :
        print(i, "ist eine kleine gerade Zahl.")
    else :
        print(i, "ist eine negative gerade Zahl.")
else :
    print(i "ist ungerade.")
```

Zuerst findet also die Prüfung statt, ob `i` gerade ist (`i % 2`). Nur, wenn diese Bedingung erfüllt ist, wird der innere **if**-Block behandelt. Durch die Einrückungen ist eindeutig festgelegt, zu welcher Bedingung welche Anweisung gehört. Hier beispielsweise ist *der gesamte innere* ***if****-Block* der Bedingung `i % 2` untergeordnet.

Das oben gezeigte Beispiel Beispiel: *Verschachtelte* ***if****-Blöcke* ist im Ergebnis äquivalent zum folgenden Code:

Beispiel: Redundanz bei `if` mit logischen Operatoren

```
i = int(input("Bitte geben Sie eine Ganzahl ein: "))

if   i % 2 == 0 and i > 100 :
    print(i, "ist eine große gerade Zahl.")
elif i % 2 == 0 and i <   0 :
    print(i, "ist eine große gerade Zahl.")
elif i % 2 == 0 :
    print(i, "ist eine negative gerade Zahl.")
else :
    print(i "ist ungerade.")
```

Möglicherweise finden Sie diese „flachere“ Form mit weniger Hierarchie-Ebenen leichter zu lesen. Beachten Sie aber auch, dass der Ausdruck `i % 2` jetzt ganze dreimal im Code steht. Nicht nur, dass Programme dieser Form langsamer ablaufen (der Ausdruck `i % 2` wird tatsächlich bis zu dreimal berechnet!), es ergibt sich hieraus auch eine Fehlerquelle! Stellen Sie sich vor, Sie wollen Ihr Programm zu späterer Zeit anpassen und nun eine andere Bedingung an `i` stellen. In dieser Form mit redundanten Prüfungen müssen Sie also auch *drei* Codestellen ändern. Häufig aber vergisst man diese Stellen im Falle von redundanten Code.

Redundanten Code vermeiden

Sie mögen sich für sehr gewissenhaft halten, und ich glaube Ihnen das gerne. So sicher Sie sich aber auch sind, dass Sie beim Überarbeiten Ihres Codes alle relevanten Stellen überprüfen werden: vermeiden Sie Redundanz! Produktiver Code erreicht schnell Längen von mehreren Tausend Zeilen und wird über Monate hinweg entwickelt. Sie machen sich – und Ihren Kollegen, die auf Ihrer Arbeit aufbauen wollen – das Leben unnötig schwer, wenn Sie Code-Teile wiederholen. Hinterfragen Sie sich jedes Mal, wenn Sie einen Copy&Paste-Schritt machen wollen, ob eine alternative Struktur nicht geeigneter wäre.

Durchdenken Sie zur Verdeutlichung nochmal diese beiden Beispiele:

Gültigkeitsprüfung: Reihe von `if`s

```
playerCount = int(input(
    "Bitte Spielerzahl eingeben: "
))

if playerCount < 2 :
    print("Ungeeignet für",
          playerCount,
          "Spieler."
    )
if playerCount > 5) :
    print("Ungeeignet für",
          playerCount,
          "Spieler."
    )
```

Gültigkeitsprüfung: logisches Oder

```
playerCount = int(input(
    "Bitte Spielerzahl eingeben: "
))

if  playerCount < 2 or
    playerCount > 5  :
        print("Ungeeignet für",
              playerCount,
              "Spieler."
        )
```

Platzhalter für Anweisungen: `pass`

Es ist schwierig, die vielen Effekte, die beim Programmieren auftreten, gleichzeitig im Kopf zu behalten. Arbeiten Sie daher kleinschrittig und prüfen Sie nach wenigen neu geschriebenen Code-Zeilen durch Ausführen, ob Ihr Code auch tatsächlich das tut, was Sie erwarten.

Wenn Sie diesen Rat beherzigen, werden Sie
a) schnell funktionierenden Code schreiben, und
b) in folgende Situation kommen:

Block-Strukturen wie `if` verlangen, dass in jedem Teil-Block Anweisungen stehen. Wenn Sie `else` schreiben, so *müssen* Sie hierfür auch Anweisungen bereit stellen. Da Sie aber vielleicht erst einen anderen Abschnitt fertig testen wollen, möchten Sie vielleicht den `else`-Block vorerst leer lassen. In diesem Fall können Sie den Befehl `pass` benutzen. Dieser macht buchstäblich *nichts*, und wurde genau zu dem Zweck zum Sprachumfang von Python hinzugefügt, um die Schrittweise Entwicklung von Code zu erlauben.

2.2.3. Der Ternäre Operator

Eine häufige Aufgabe beim Programmieren ist die *bedingte Zuweisung* eines Wertes. Abhängig vom Wert eines Ausdrucks soll der Wert einer Variablen gesetzt werden. Sie wissen bereits, dass Sie dies mit einem `if`-Block erledigen können:

Bedingte Wertzuweisung mit `if`-Block

```
if Bedingung :
    variable = Wert1
else :
    variable = Wert2
```

Für diese Aufgabe existiert eine Kurzform des *ternären Operators*[4], die sich schon fast wie englischer Prosa-Text liest:

Bedingte Wertzuweisung mit dem Ternären Operator

```
variable = Wert1 if Bedingung else Wert2
```

Sie können diese Ausdrucksform sogar an die Stelle anderer Ausdrücke setzen und beispielsweise an Funktionen als Argument übergeben:

Beispiel: Ternärer Operator als Argument

```
x = float(input("Bitte geben Sie eine Zahl ein: "))
print("Der Betrag von", x, "ist:", x if x > 0 else -x)
```

[4] Operatoren brauchen *Argumente*, aus denen Werte berechnet werden können. Man klassifiziert nach der Zahl der notwendigen Argumente. So gibt es das *unäre* `not`, das *binäre* `+` und eben den ternären Operator, der *drei* Argumente braucht.

3. Module Laden

> Everything is awesome!
>
> Emmet Brickowski

Die Sprache Python kennt insgesamt nur 33 Schlüsselworte. Aus diesen wenigen Sprachelementen lassen sich erste Algorithmen bauen, die wiederum in komplexeren Routinen eingebaut werden können. Einer der großen Vorteile von Python ist es, dass für die weit meisten Aufgaben, die im Programmierer-Alltag auftauchen, bereits vorgefertigte Routinen zur Verfügung stehen.

Damit Routinen genutzt werden können, müssen diese zunächst im Arbeitsspeicher vorliegen. Kaum ein Programm wird *alle* Routinen benutzen, die in der *Standardbibliothek* der Sprache Python angeboten werden. Um also nicht eine unsinnig große Menge an Speicherplatz zu belegen, bevor auch nur eine einzige Zeile nützlichen Codes ausgeführt wird, können wir selbst bestimmen, welche *Module* geladen werden sollen. Ein Modul ist eine Sammlung von Routinen, die einem gemeinsamen „Thema“ zugeordnet sind (beispielsweise mathematische Funktionen).

3.1. Der Befehl `import`

Geladen wird ein Modul mit dem Befehl `import`:

Syntax

```
import module
```

Sobald ein Modul geladen ist, können wir die darin zusammen gefassten Funktionen aufrufen. Zwei Module können Funktionen mit gleichem Namen haben. Damit klar ist, welche Funktion wirklich gemeint ist, muss der Modulname beim Funktionsaufruf mit genannt werden:

Syntax

```
module.function(arguments)
```

Beispielsweise existieren die Module `math` und `cmath`. Beide stellen mathematische Funktionen zur Verfügung; jedoch ist `math` auf relle Zahlen ausgelegt, `cmath` dagegen auf komplexe Zahlen. Wir können die Funktionen aus beiden Modulen so benutzen:

Beispiel: Funktionen aus `math` und `cmath`

```
1 import math
2 import cmath
3
4 print( math. sin( math.pi / 2) )
5 print( cmath.sin(cmath.pi / 2) )
```

Ausgabe

```
1.0
(1+0j)
```

Module können ihrerseits wieder aus Unter-Modulen bestehen, so dass sich ein „Modul-Pfad“ ergibt. Ein Beispiel ist die Komponente `pyplot` aus dem Modul `matplotlib`, das zur graphischen Darstellung von Daten genutzt werden kann[1]. Importiert man die `matplotlib`, so stehen die Funktionen von `pyplot` unter dem länglichen Präfix `matplotlib.pyplot` zur Verfügung.

Es ist lästig, für jeden Funktionsaufruf den vollen Präfix `matplotlib.pyplot` anzugeben. Stattdessen können für diese Modulpfade neue Symbole vergeben werden:

Syntax

```
import long.module.path as alias
```

Es ist beispielsweise sehr geläufig, das angesprochene Modul `matplotlib.pyplot` unter dem *Alias* `plt` zu laden:

Beispiel: `import` mit Alias

```
1 import matplotlib.pyplot as plt
2
3 plt.figure()    # statt matplotlib.pyplot.figure()
```

Wenn Sie nur einzelne Funktionen eines Moduls benutzen, können Sie auch diese auch alternativ mit `from` laden:

Syntax

```
from module import function
```

Um Namenskollisionen zu umgehen, besteht auch hier die Möglichkeit, ein Alias zu vergeben:

Syntax

```
from module import function as alias
```

Dieser Alias darf dann aber keinen Punkt (`.`) enthalten.

Das obige Beispiel: *Funktionen aus* `math` *und* `cmath` kann also auch folgendermaßen geschrieben werden:

Beispiel: `from ... import`

```
1 from  math import sin
2 from  math import pi
3 from cmath import sin as csin
4
5 print(  sin(pi / 2) )
6 print( csin(pi / 2) )
```

[1] Wir werden die Funktionen dieses Moduls ausführlicher in Kapitel 10 besprechen. Für hier soll es Ihnen genügen zu wissen, dass dieses Modul existiert.

`import` oder `from ... import`?

Welche der beiden Methoden – `import` oder `from ... import` – am besten funktioniert, ist letztlich Frage des Geschmacks. Die `import`-Methode lädt ein Modul als Gesamtpaket, verlangt aber bei jedem Aufruf ein Präfix. Dagegen sind bei `from ... import` jeweils eigene Zeilen für jede geladene Funktion und Konstante nötig. In jedem Fall sollten Sie konsistent bleiben, d. h. nur entweder `import` oder `from ... import` benutzen.

In diesem Kurs verwende ich nur die `import`-Methode. Durch diese ist es leichter, das Verhalten von Funktionen abzuschätzen, da bereits sofort der Kontext mitgeliefert wird. (Eine Funktion aus `cmath` wird i. d. R. einen Wert vom Typ `complex` berechnen, während `math` i. d. R. Werte vom Typ `float` berechnet. Solche Details können große Auswirkungen haben, und es ist oft gut, im Code direkt daran erinnert zu werden.)

`import`-Zeilen am Anfang des Codes

Funktionen, die mit `import` (bzw. `from ... import`) geladen werden, sind i. d. R. für den gesamten Code relevant. Daher sollten die `import`-Anweisungen auch die ersten Zeilen Ihres Codes darstellen und *nur* dort auftauchen. Auf diese Weise kann ein Leser Ihres Codes sich sofort auf die verwendeten Methoden einstellen und weiß, wo er oder sie nachlesen kann, für welche Module ihre Aliase stehen.

Die für Python verfügbaren Module haben i. d. R. einen Webauftritt, in der alle Funktionen dokumentiert sind. Die Suchmaschine der Wahl sollte mit dem Begriff *Python [Modulename] Documentation* eine ausführliche Beschreibung zum Funktionsumfang liefern. Auf diese Art finden Sie u. a. den Link

`https://docs.python.org/2/library/math.html`

und damit eine Beschreibung des Python-Moduls `math`. Stöbern Sie in dieser Auflistung, und gewöhnen Sie sich an die dort verwendete Sprache. Viele komplexe Aufgaben können durch Verwendung eines geeigneten Moduls sehr schnell gelöst werden. Dazu müssen Sie als ProgrammiererIn jedoch die entsprechenden Funktionen identifizeren und sich ihre Benutzung selbst anlesen können. Angesichts der Vielzahl an Aufgaben und verfügbaren Pakete kann kein Kurs erschöpfend auf die verfügaren Funktionen eingehen.

3.2. Eigene Module

Module in Python können auf verschiedene Art umgesetzt werden. In den meisten Fällen handelt es sich aber tatsächlich einfach um normalen Code, wie Sie ihn bereits gesehen haben und im weiteren kennen lernen werden. Das heißt, dass die `.py`-Dateien, die Sie hier zu schreiben lernen, auch ihrerseits als Module genutzt werden können. Wenn Sie also ihren Code in der Datei `myModule.py` speichern, so können Sie durch die Zeile `import myModule` (*ohne* Erweiterung `.py`!) diesen Code in eine andere Code-Datei einbinden. Stellen Sie sich hierzu vor, dass der `import`-Befehl den Inhalt von `myModule.py` kopiert und einfügt.

Damit der Python-Interpreter weiß, aus welchem Verzeichnis die Module geladen werden sollen, muss theoretisch auch eine Angabe des Verzeichnisses erfolgen. In Pyhton geschieht dies implizit, d. h. der Interpreter sucht in einer Reine von Verzeichnissen und lädt das erste Modul, das einen passenden Namen hat. Der erste Ort, an dem gesucht wird, ist das aktuelle Arbeitsverzeichnis. Im Anschluss werden Ordner durchsucht, die bei der Installation von Python als „Standard-Pfade“ festgelegt wurden.

Beispiel: In Ihrem aktuellen Arbeitsverzeichnis befinden sich die Dateien `foo.py` und `bar.py`:

Beispiel: `foo.py`

```
1  print("loaded foo")
```

Beispiel: `bar.py`

```
1  import foo
2  print("executing bar")
```

Ausgabe: Ausführung von `bar.py`

```
loaded foo executing bar
```

Ein Modul kann nur ein einziges Mal geladen werden. Jeder weitere Versuch, dasselbe Modul mit `import` erneut zu laden, wird ignoriert. Die folgende Abwandlung von `bar.py` erzeugt also dieselbe Ausgabe wie oben:

Beispiel: `bar.py`

```
1  import foo
2  import foo
3  print("executing bar")
4  import foo
```

Befindet sich das zu ladende Modul in einem Unterordner des aktuellen Arbeitsverzeichnisses, so kann dies durch Punkte angedeutet werden. Wir nehmen an, dass das Hauptmodul `bar.py` im aktuellen Arbeitsverzeichnis liegt; das zu ladende Modul `foo.py` dagegen sei im Ordner `subfolder`. In dem Fall schreiben wir:

Beispiel: `bar.py`

```
1  import subfolder.foo
2  print("executing bar")
```

um das gewünschte Verhalten zu erreichen.

Namenskollisionen mit anderen Modulen

In verschiedenen Ordnern können Dateien mit gleichem Dateinamen liegen. Sie können beispielsweise eine Datei namens `math.py` anlegen und diese in Ihrem Code-Verzeichnis speichern. Dies hat allerdings zur Folge, dass in allen Codes, die in diesem Ordner liegen, die Zeile `import math` nicht mehr die Python-Funktionen lädt, sondern was auch immer in Ihrer Datei geschrieben steht.

Python-Module sind zahlreich und es werden regelmäßig neue Pakete veröffentlicht. Daher kann auch keine Liste an „verbotenen“ Namen angegeben werden. Stattdessen kann ich Ihnen hier nur mitgeben: Wenn Sie beim Laden eines Moduls Probleme bekommen, sollten Sie die Namen der Dateien im aktuellen Arbeitsverzeichnis durchsehen. Eine eventuelle Namenskollision könnte Ursache Ihrer Probleme sein.

Im Code kann auch der Name des aktuell ausgeführten Moduls abgefragt werden. Die Variable `__name__` enthält einen String, der den aktuellen Modulnamen speichert. Für das *Hauptmodul* (also die Code-Datei, die ursprünglich gestartet wurde) ist dies immer `"__main__"`.

Beispiel: `foo.py`

```
1 print("Module name:", __name__)
```

Beispiel: `bar.py`

```
1 import foo
2 print("Module name:", __name__)
```

Ausgabe: Ausführung von `bar.py`

```
Module name: foo
Module name: __main__
```

4. Datenstrukturen

> Multimedia? As far as I'm concerned, it's reading with the radio on!
>
> Rory Bremmer

Bis hierhin haben wir sehr kleine Datenmengen behandelt. Unser bisheriges „Arbeitsmaterial“ waren Variablen, die je einen einzelnen Wert repräsentieren. Eine Stärke von Computern ist es aber gerade, große Datenmengen schnell zu verarbeiten. Hier werden wir Möglichkeiten kenenn lernen, nahezu beliebig große Datenmengen im Speicher zu halten und zu manipulieren.

4.1. Speichermodell

Bevor wir das Verhalten der verschiedenen Speicherstrukturen verstehen können, die Python uns zur Verfügung stellt, müssen wir uns mit der Art und Weise vertraut machen, in der Daten im Speicher abgelegt werden.

Man kann sich den Arbeitsspeicher als langes Band von kleinen, nummerierten Speicherzellen vorstellen. Jede Zelle fässt genau ein Byte. Um einen Wert zu lesen oder zu schreiben muss dem Prozessor die Nummer der Zelle mitgeteilt werden, die verändert wird. Diese Nummer wird *Adresse* oder *Pointer* genannt. Wenn wir im Code Variablen benutzen, übersetzt der Compiler diese in Adressen.

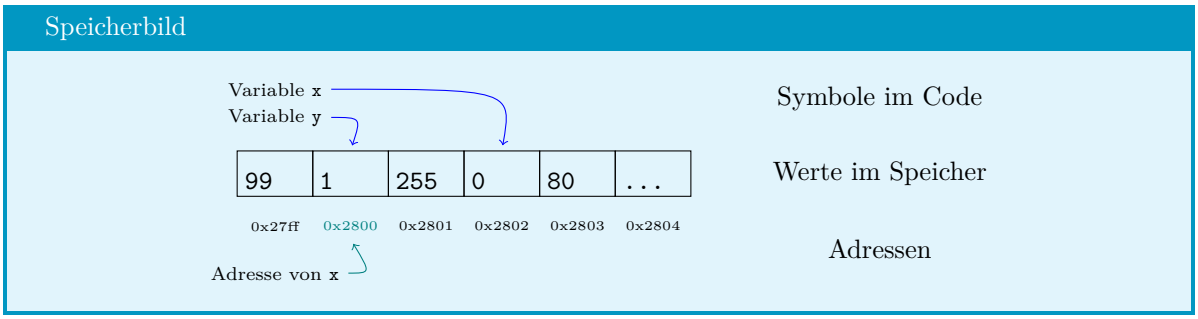


Abbildung 4.1.: Speicherbild: nummerierte Zellen

Während wir der Einfachheit halber oft sagen, dass eine Variable einen *Wert* speichert, ist tatsächlich die Information hinterlegt, *wo der Wert selbst zu finden ist*, also die Adresse des Wertes. Dies hat den Vorteil, dass Aufgaben sehr effizient erledigt werden können, wenn große Datenmengen bewegt werden müssen: anstatt viele Megabytes zu kopieren, muss nur eine Referenz an die Stelle gesetzt werden, wo die zu kopierenden Daten bereits im Speicher liegen. Für uns als ProgrammiererInnen heißt dies aber auch, dass wir diese Speicherstruktur im Hinterkopf behalten müssen.

Stellen Sie sich vor, Sie verwalten eine Liste. Diese soll über den Variablennamen `originalList` ansprechbar sein. Nun wollen Sie eine Arbeitskopie dieser Liste anlegen und *in dieser Kopie* Werte verändern. Sie wollen also folgendes Speicherbild erreichen:

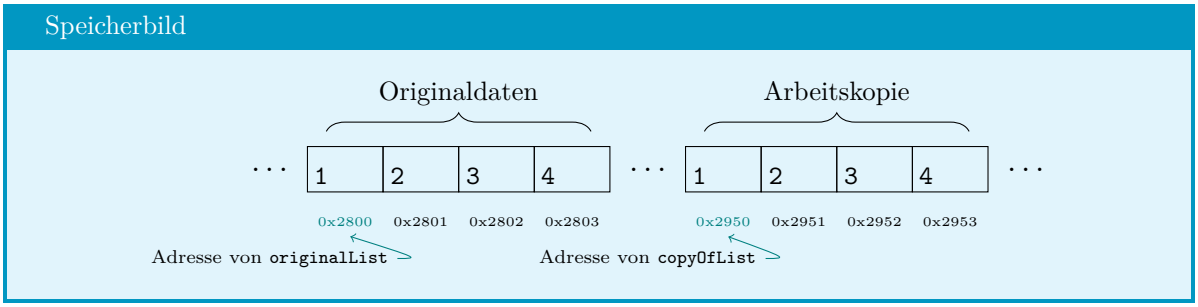


Abbildung 4.2.: Speicherbild: Arbeitskopie

Wenn Sie nun die Codezeile

```
copyOfList = originalList
```

tippen, werden Sie aufgrund dieser Arbeitsweise von Python aber folgendes Speicherbild erzeugen:

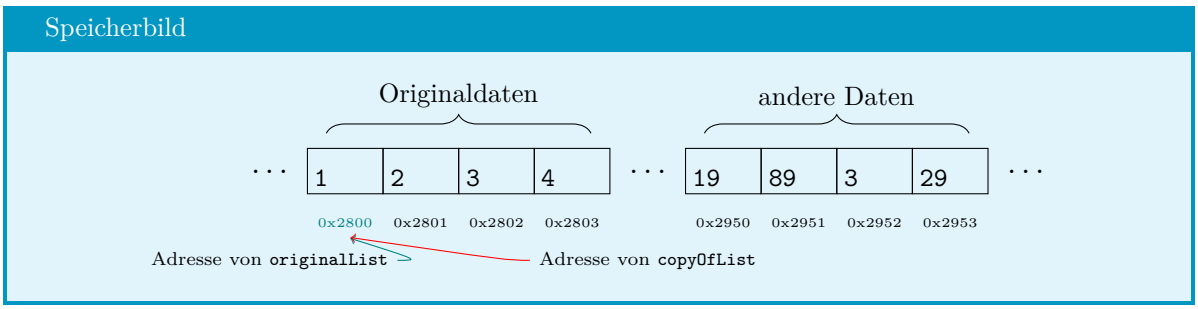


Abbildung 4.3.: Speicherbild: Zwei Referenzen auf dieselben Daten

Anstatt eine Kopie der Liste anzulegen, haben Sie nur eine *Kopie der Referenz* erzeugt. Ein Zugriff über das Symbol `copyOfList` ändert also immer noch Ihr Original!

Um nun das gewünschte Ziel zu erreichen, müssen Sie stattdessen die Funktion `copy` aus dem Modul `copy` benutzen:

Schema: Shallow Copy anlegen

```
import copy

# Code zum Aufbau der Liste originalList

copyOfList = copy.copy(originalList)
```

4.2. mutable und immutable objects

Python kennt zwei große Gruppen von Objekten: veränderbare Objekte (*mutable objects*) und unveränderliche Objekte (*immutable objects*).

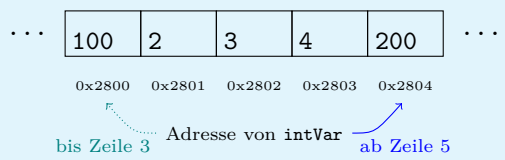
Die *Werte* von immutable objects dürfen sich nicht mehr ändern, sobald sie einmal im Speicher abgelegt wurden. Wenn eine Variable geändert wird, die ein immutable object speichert, so wird ein neues Objekt im Speicher konstruiert, nicht aber die alte Stelle überschrieben.

Betrachten Sie dazu das folgende Beispiel:

Codebeispiel

```
1  # int-Variablen sind immutable
2
3  intVar = 100
4
5  intVar = 200
```

Speicherbild



In dem hier gezeigten Code ändern wir augenscheinlich den Wert von `intVar`. Der Typ `int` ist jedoch immutable! Daher wird der Python-Interpreter dafür sorgen, dass mit Zeile 5 ein neues `int`-Objekt im Speicher angelegt wird, das den neuen Wert speichert. Die Referenz von `intVar` geht nun auf diese neue Speicherstelle. Der Wert 100, der in der alten Speicherstelle lag, wird hingegen nicht angerührt.

Bei mutable objects hingegen wird wirklich der Inhalt der Speicherzellen selbst überschrieben. Sie erkennen, dass hier das Problem der vermeintlichen Arbeitskopie auftritt.

Im Moment haben wir nur immutable objects kennen gelernt. In diesem Kapitel werden die ersten mutable objects eingeführt. Am Ende des Kapitels finden Sie eine Übersichtstabelle zu den Ihnen bis dahin bekannten Speicherstrukturen.

Speicheradresse ausfindig machen mit `id`

Der Befehl `id` kann auf alle Datenobjekte angewandt werden, und gibt die Adresse im Speicher aus, also die Nummer der Speicherzelle.

Neben Variablen (`id(x)` – Adresse der Variablen `x`) kann der Befehl auch auf Datentypen (`id(int)` – Adresse der „Beschreibung des Typs“) und auf Konstanten (`id(1)` – Adresse des Werts `1`) angewandt werden.

Wenn Sie hier ein wenig experimentieren, werden Sie feststellen, dass „kleine“ Ganzzahlen in relativer Nähe zueinander liegen, während „große Zahlen“ (> 256) signifikant andere Adressen erhalten. Dies liegt daran, dass die Entwickler von Python erwartet haben, dass diese Zahlen sehr häufig als (Zwischen-)Ergebnisse auftreten. Daher werden diese bereits „auf Verdacht vorbereitet“. Größere Zahlen werden erst im Speicher angelegt, wenn dies wirklich nötig wird.

Vergleichsoperator `is` vs. `==`

Wir haben bereits den Operator == kennengelernt, der uns mitteilt, ob zwei Objekte gleich sind. Hierbei ist mit *Gleichheit* gemeint, dass sie denselben Wert speichern. Wie Sie gesehen haben, können Kopien voneinander an verschiedenen Speicherorten abgelegt werden.

Der Operator `is` vergleicht nicht die Werte, sondern die Speicheradressen, und gibt folglich `True` zurück, wenn zwei Variablen dasselbe Objekt referenzieren. Die folgenden beiden Codezeilen sind also äquivalent:

Übersetzung von `is`

```
print( id(a) == id(b) )
print(    a  is    b  )
```

Beachten Sie, dass diese Bedeutung des Operators `is` manchmal unintuitives Verhalten erzeugt:

Unintuitives Verhalten von `is`

```
5 == 4 is not True
```

dieser Ausdruck wird zu `True` ausgewertet, obwohl offensichtlich $5 \neq 4$ gilt.

Dies lässt sich folgendermaßen begründen:

Python wertet zunächst den ersten Teil der Aussage (`5 == 4`) zu `False` aus. Dieser Wert ist ein häufig gebrauchter, und wurde daher von Python bereits vorbereitet, hatte also schon vor diesem ersten Rechenschritt eine Adresse. Diese Adresse wird dem Ausdruck `5 == 4` zugeordnet.

Als nächstes wird die zweite Teilaussage (`not True`) ebenfalls zu `False` ausgewertet. Wieder erkennt Python den vorbereiteten Wert, und weist `not True` die Adresse von `False` zu.

Der Operator `is` vergleicht nun die Adressen von `False` und `False`, und kommt folgerichtig zu dem Ergebnis, dass diese gleich sind. Mit anderen Worten, `False is False` ist `True`.

4.3. `list`s

Der Datentyp `list` repräsentiert – wie der Name vermuten lässt – Listen. Die aufgelisteten Werte können dabei ganz verschiedener Natur sein: `list`s können `int`s, `float`s, ... und sogar andere Listen enthalten. Die Elemente derselben Liste müssen nicht vom selben Typ sein.

4.3.1. Anlegen und Auslesen

Erstellt werden Listen, indem man in [eckigen Klammern] die Elemente der Liste durch Kommata getrennt aufzählt:

Syntax: Liste anlegen

```
listVariable = [listItem1, listItem2, ...]
```

Listen dürfen auch leer sein. Eine leere Liste wird als `[]` geschrieben.

Auf die Elemente einer Liste wird mittels ihres *Index* zugegriffen, d. h. der Nummer innerhalb der Liste. Dabei hat das erste Element der Liste den Index `0`! Der Index wird in [eckigen Klammern] dem Symbol der Listenvariable nachgestellt, um anzugeben, dass man ein einzelnes Element der Liste ansprechen will:

Syntax: Listenelement ansprechen

```
listVariable[listItems]
```

Indices können auch negativ sein! In diesem Falle wird „von hinten herein“ gezählt. `-1` referenziert also das letzte Element der Liste, `-2` das vorletzte, usw.

Ist das referenzierte Listenelement selbst eine Liste, so kann auf dessen Elemente ebenfalls durch Nennung des Index in einer eigenen Klammer zugegriffen werden.

`list`s können auch in normale Variablen „entpackt“ werden. Dazu verwendet man den Zuweisungsoperator =, gibt aber auf der linken Seite so viele Elemente an, wie sie in der Liste sind:

Entpacken von Listen

```
>>> numbers = [1, 2, 3]
>>> a, b, c = numbers
>>> b
2
```

4.3.2. Slicing

Aus einer Liste kann auch ein Teil herausgegriffen werden. Man nennt dies *slicing*:

Syntax: Slicing

```
listVariable[start : end : stride]
```

Mit dieser Syntax wird eine *neue Liste* berechnet, die beim Index `start` beginnt, die Elemente bis *ausschließlich* dem Index `end` beinhaltet, und deren Elemente in der Quell-Liste einen Abstand von `stride` haben. Wird `stride` ausgelassen, werden alle Elemente zwischen `start` und `end` in die neue Liste übernommen.

Wird `start` oder `end` ausgelassen, so versteht Python dies als: „vom Anfang“ bzw. als „bis zum Ende“.

4.3.3. Addition und Multiplikation bei Listen

Die Addition von Listen führt zur Verkettung, wie Sie das schon von Strings kennen. Genauso wie dort führt die Multiplikation mit einer Ganzzahl zu einer *Wiederholung* der Listenelemente:

Multiplikation von Listen

```
>>> [1, 2] + [3]
[1, 2, 3]
>>> 3 * [1, 2]
[1, 2, 1, 2, 1, 2]
```

4.3.4. Beispiel

Beispiel: Listenzugriffe

```
1  myList = [1, 2, -5, 3.14, "some text", [3, 2], (1+1j)]
2
3  print(myList[0])         # erstes Element
4  print(myList[-1])        # letztes Element
5  print(myList[-2][0])     # erstes Element des vorletzten Elements
6  print(myList[1:3])       # slicing: Elemente 1 und 2 (ausschließlich 3)
7  print(myList[-2:])       # slicing: vorletztes Element bis Ende
8  print(myList[:5:2])      # slicing: Elemente mit Indices 0 bis 4 in 2er-Schritten
9  print(myList[::2])       # slicing: alle mit geradem Index
10
11 myList += [[]]           # an die Liste eine leere Liste anhängen
12 print(myList[-1])        # letztes Element
13 print(myList)            # gesamte Liste
```

Ausgabe

```
1
(1+1j)
3
[2, -5]
[[3, 2], (1+1j)]
[1, -5, 'some text']
[1, -5, 'some text', (1+1j)]
[]
[1, 2, -5, 3.14, "some text", [3, 2], (1+1j), []]
```

Beachten Sie besonders die Klammern in Zeile 11: `myList += [[]]`. Die äußeren Klammern geben an, dass das Objekt, das addiert wird, eine Liste ist. Dies ist notwendig, da nur die Addition zwischen Listen erklärt ist; `myList += 3` würde (für den Python-Interpreter) keinen Sinn ergeben. Alles, was *in diesen Klammern steht*, wird an die Liste angehängt. In diesem Fall ist dies `[]`, also eine leere Liste. Machen Sie sich klar, dass im Gegensatz zu `myList += [[]]` (anhängen einer leeren Liste) durch den Befehl `myList += []` *nichts* zu `myList` hinzugefügt wird.

`list`s sind mutable. Das heißt, für Kopien muss der `copy`-Befehl aus dem Modul `copy` verwendet werden, oder mittels der Slicing-Syntax eine Kopie erstellt werden:

Beispiel: mutable list

```
1  import copy
2
3  originalList = [1, 2, 3]
4  refCopy = originalList
5  truCopy = copy.copy(originalList)
6  altCopy = originalList[:]
7
8  refCopy += [4]
9  truCopy += [5, 5]
10 altCopy += [6, 6]
11
12 print("original    : ", originalList)
13 print("copy.copy   : ", truCopy)
14 print("slicing copy: ", altCopy)
```

Ausgabe

```
original    : [1, 2, 3, 4]
copy.copy   : [1, 2, 3, 5, 5]
slicing copy: [1, 2, 3, 6, 6]
```

Wie Sie sehen, wird in Zeile 8 ein Element an `refCopy` angehängt. Da `refCopy` aber auf die Speicherstelle von `originalList` verweist, ändert sich also die ursprüngliche Liste damit ebenfalls.

`truCopy` ist tatsächlich eine neue Liste, die eine *Kopie* der ursprünglichen Liste enthält. Sie wurde an einer von `originalList` unabhängigen Speicherstelle angelegt, und spürt damit die Änderung durch Zeile 8 nicht. Auf dieselbe Weise bleibt `originalList` von der Änderung durch Zeile 9 unbeeinflusst.

Dasselbe gilt für `altCopy`: der Slicing-Operator bewirkt dasselbe wie der Befehl `copy.copy`.

Achtung: `list`s werden intern als Abfolge von Adressen der Elemente in der Liste realisiert. Wenn das referenzierte Objekt wiederum eine Liste ist, so kann selbst über eine „echte“ Kopie die Original-Liste geändert werden. Die Funktion `deepcopy` aus dem Modul `copy` umgeht dies, indem wirklich *rekursiv* Kopien von allen Ebenen der Liste angelegt werden:

Beispiel: Kopien mit `deepcopy`

```
1  import copy
2
3  originalList = [1, 2, [1, 2]]
4  normCopy = copy.copy(originalList)
5  deepCopy = copy.deepcopy(originalList)
6  normCopy[-1] += [3]
7
8  print("copy.copy    : ", originalList)
9  print("copy.deepcopy: ", deepCopy)
```

Ausgabe

```
copy.copy    : [1, 2, [1, 2, 3]]
copy.deepcopy: [1, 2, [1, 2]]
```

4.3.5. Methoden

Methoden sind Programmroutinen, die ein Datenobjekt verändern, oder auf Basis des Datenobjekts Ergebnisse berechnen. `list`s sind solche Datenobjekte. Jede Klasse (d. h. „Art“ von Objekten) hat seine eigenen Methoden. In Kapitel 7 wird dies im Detail behandelt. Hier sei zunächst vorausgeschickt, wie wir mit solchen Methoden umgehen.

`append`

Eine solche Methode ist `append`. Sie wird verwendet, um Elemente zu einer Liste hinzuzufügen. Aufgerufen wird eine Methode, indem man das zugrundeliegende Datenobjekt nennt, und, getrennt durch einen Punkt, die Methode anhängt. Methoden sind *Funktionen*, also folgt wie üblich eine Parameterliste in runden Klammern.

Beispiel: Methode `append`

```
1  numbers = [1, 2]
2
3  numbers.append(3)
4  numbers.append([4])
5
6  print(numbers)
7
8  numbers += [5]
9  numbers += [[6]]
10
11 print(numbers)
```

Ausgabe

```
[1, 2, 3, [4]]
[1, 2, 3, [4], 5, [6]]
```

Wie Sie sehen, wird in den Zeilen 3 und 4 jeweils ein Element zur Liste hinzgefügt (die Zahl `3` und die Liste `[4]`). Denselben Effekt hat der *Operator* `+=`. Während im ersten Fall aber *Elemente* als Argumente übergeben werden, muss beim Operator eine *Liste* genannt werden, mit der die Verknüpfung stattfindet.

`insert`

Ähnlich funktioniert die Methode `insert`: Sie fügt einen Wert zur Liste hinzu. Im Gegensatz zu `append` jedoch kann mit `insert` auch die Position innerhalb der Liste festgelegt werden: Beim Einfügen muss dazu sowohl der *Index* (die Position in der Liste, an der eingefügt werden soll) als auch das Element selbst genannt werden:

Beispiel: Methode `insert`

```
numbers = [1, 2]

numbers.insert(1, 99)
print(numbers)

numbers = numbers[:1] + [-99] + l[1:]
print(numbers)
```

Ausgabe

```
[1, 99, 2]
[1, -99, 99, 2]
```

Auch negative Indices können angegeben werden. `l.append(x)` und `l.insert(-1, x)` haben also dieselbe Auswirkung.

Indices beginnen bei 0!

Beachten Sie, dass das *erste* Element einer Liste den Index `0` hat! Der Index `1` bezeichnet also das *zweite* Element.

`remove`

Das Gegenstück zu `insert` ist `remove`: Es löscht einen bestimmten Wert aus der Liste. Als Parameter wird der *Wert* selbst angegeben, nicht der Index. Taucht der Wert in der Liste mehrfach auf, so wird das erste Element gelöscht, das dem Parameter gleicht. Ist der Parameter gar nicht in der Liste, so wird eine Fehlermeldung ausgegeben.

Beispiel: Methode `remove`

```
numbers = [1, 2, 4, 3, 4, 4, 5]

numbers.remove(4)
print(numbers)

# numbers.remove(8)   -- Fehler: 8 nicht in numbers
```

Ausgabe

```
[1, 2, 3, 4, 4, 5]
```

`sort` und `reverse`

Wie der Name vermuten lässt, dienen diese Methoden dazu, `list`s zu sortieren bzw. in der Reihenfolge umzudrehen. Es versteht sich von selbst, dass Sortieren nur dann möglich ist, wenn ein sinnvolles Sortierkriterium gegeben ist. Zahlen werden aufsteigend nach Wert sortiert, Strings lexikographisch (also alphabetisch mit bestimmten Regeln für Zahlen und Sonderzeichen). Listen aus gemischten Elementen (also aus Zahlen und Strings) können nicht sortiert werden.

Wir werden in Kapitel 6 eine Möglichkeit kennen lernen, diese Funktionalität zu erweitern.

Beispiel: Methode `sort` und `reverse`

```
numbers = [5, -2, 3, 4, 4, 4, 5]

numbers.sort()
print(numbers)

numbers.reverse()
print(numbers)

numbers = [1, "broccoli"]
# l.sort()   -- Kann 1 nicht mit "broccoli" vergleichen
```

Ausgabe

```
[-2, 3, 4, 4, 4, 5, 5]
[5, 5, 4, 4, 4, 3, -2]
```

`pop`

Die Methode `pop` kombiniert die Aufgaben „lese das letzte Element der Liste“ und „entferne das letzte Element der Liste“. Dies ist nützlich, um „Aufgabenstapel abzuarbeiten“.

Beispiel: Methode `pop`

```
jobs = ["go have a coffee",
        "think of some nice examples",
        "write the chapter"]

print( "next job  : ", jobs.pop() )
print( "jobs to do: ", jobs)
```

Ausgabe

```
next job  : write the chapter
jobs to do: ['go have a coffee', 'think of some nice examples']
```

Weitere Methoden

Neben den oben gezeigten Methoden existieren noch weitere, die hier nicht erschöpfend erklärt werden können. Stattdessen möchte ich Sie dazu ermutigen, sich mit der offiziellen Dokumentation der Sprache auseinander zu setzen.

Unter:
`https://docs.python.org/3/tutorial/datastructures.html`
finden Sie (knappe) Erklärungen zu allen Methoden, die sowohl die `list` als auch alle weiteren hier besprochenen Speicherstrukturen zur Verfügung stellen.

4.4. tuples

tuples sind die immutable Cousins der list: auch sie repräsentieren Listen. Wie bei list spricht man Syntaxelemente über einen Index in [eckigen Klammern] an. Angelegt werden sie ähnlich, jedoch mit runden Klammern. Addition und Multiplikation funktionieren wie bei lists:

Beispiel: tuples

```
1  tup_numbers = (1, 2, 3)
2  print( tup_numbers + (4, 5) )
3  print( 2 * tup_numbers)
4  print( t[0] )
5
6  a, b, c = tup_numbers
7  print(b)
```

Ausgabe

```
(1, 2, 3, 4, 5)
(1, 2, 3, 1, 2, 3)
1
2
```

Viele Funktionen in Python geben tuples zurück. Ein Beispiel hierfür ist die Funktion divmod, die sowohl Quotient als auch Rest einer Division in einem Schritt berechnet:

Beispiel: divmod

```
1  dm = divmod(11, 3)
2
3  print("11 / 3 = ", dm[0], " Rest ", dm[1])
```

Ausgabe

```
11 / 3 = 3 Rest 2
```

Da tuples *immutable* sind, existieren keine Funktionen, die diese verändern, wie z. B. sort oder append. Es ist jedoch möglich, aus einem tuple eine list mit gleichen Inhalten zu generieren, und diese dann – nach Bearbeitung – wieder in einen tuple zurückzuverwandeln:

Beispiel: Type conversion mit list und tuple

```
1  tup_numbers = (5, 2, 3)
2
3  lst_numbers = list(tup_numbers)
4  print(lst_numbers)
5
6  lst_numbers.sort()
7
8  tup_numbers = tuple(lst_numbers)
9  print(tup_numbers)
```

Ausgabe

```
[5, 2, 3]
(2, 3, 5)
```

Sie kennen diese Art der Typumwandlung bereits aus Abschnitt 1.4.2.

Klammer-Typen

Beachten Sie im letzten Beispiel genau die Ausgabe: `tuple`s werden in (runden Klammern) ausgegeben, `list`s dagegen in [eckigen Klammern].

Auch, wenn hier der Eindruck entsteht, der `tuple` `tup_numbers` wäre verändert worden, bleibt die Aussage, dass `tuple`s *immutable* sind. Das *alte* Objekt `tup_numbers` wurde in Zeile 8 verworfen. An einer neuen Stelle im Speicher wird ein *neuer* `tuple` konstruiert, der mit dem *alten* `tup_numbers` nichts zu tun hat.

4.5. `set`s und `frozenset`s

`set`s sind eine Variante von `list`s. Der Unterschied besteht darin, dass es in `set`s keine Doubletten gibt. Jedes Element eines `set`s ist einmalig; versucht man, dasselbe Element ein zweites Mal hinzuzufügen, so passiert einfach gar nichts.

Man erstellt `set`s wie `list`s, jedoch mit {geschweiften Klammern}. Das Ansprechen einzelner Elemente geschieht wieder wie bei `list`s durch Nennung des Index in [eckigen Klammern].

Beispiel: `set`s

```
1  basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
2  print(basket)
3  print(basket[0])
```

Ausgabe

```
{'orange', 'banana', 'pear', 'apple'}
orange
```

Ähnlich wie `tuple`s können `set`s aus `list`s (oder allen anderen Datenstrukturen) konstruiert werden:

Beispiel: Type conversion mit `set`s

```
1  lst_numbers = [1, 2, 1, 3, 1, 4]
2  set_numbers = set(lst_numbers)
3  uniqueList = list(set_numbers)
4  print(set_numbers)
5  print(uniqueList)
```

Ausgabe

```
{1, 2, 3, 4}
[1, 2, 3, 4]
```

Wie schon zuvor funktioniert die Addition wie bei `list`s. Die Multiplikation ist nicht definiert, da ein `set` nie doppelte Elemente enthalten darf. Auf `set`s können dieselben Operationen wie auf `list`s angewandt werden. Hinzu kommen einige s

So, wie es zur `list` das *immutable* Analogon `tuple` gibt, hat das `set` im `frozenset` sein *immutable* Gegenstück. `frozenset`s haben keinen eigenen Typ von Klammern, und werden stattdessen über ihren Typnamen aus Listen (beliebigen Typs) generiert:

Beispiel: Type conversion mit `frozenset`s

```
1  lst_numbers = [1, 2, 1, 3, 1, 4]
2  fst_numbers = frozenset(lst_numbers)
3  uniqueList  = list(fst_numbers)
4
5  print(fst_numbers)
6  print(uniqueList)
```

Ausgabe

```
frozenset({1, 2, 3, 4})
[1, 2, 3, 4]
```

4.6. Strings

Strings wurden bereits in Abschnitt 1.4.2 vorgestellt. Hier möchte ich Sie nur darauf hinweisen, dass Strings eine besondere Art von `tuple`s sind, nämlich *immutable Listen von einzelnen Buchstaben.*

Mit diesem Wissen ist es für Sie leicht, das folgende Verhalten nachzuvollziehen:

Beispiel: Strings als `tuple`s

```
1  powerlevel = "over 9000"
2  tup_powerlevel = tuple(powerlevel)
3
4  print(tup_powerlevel)
```

Ausgabe

```
('o', 'v', 'e', 'r', ' ', '9', '0', '0', '0')
```

Strings lassen sich auch *indizieren.* `stringVariable[i]` gibt das *Zeichen an der Stelle* `i` zurück. Beachten Sie, dass die Zählung auch hier bein `0` beginnt.

4.7. ranges

ranges werden in Kapitel 5 von Bedeutung sein. Sie repräsentieren Ganzzahlen zwischen bestimmten Grenzen. Das Schlüsselwort `range` kann auf drei verschiedene Arten genutzt werden:

- `range(N)` erzeugt eine Liste der Zahlen von einschließlich `0` bis ausschließlich `N`. Beispielsweise steht `range(5)` für die Zahlen `0, 1, 2, 3, 4`.
- `range(start, end)` erzeugt eine Liste der Zahlen von einschließlich `start` bis ausschließlich `end`. Beispielsweise steht `range(2, 5)` für die Zahlen `2, 3, 4`.
- `range(start, end, stride)` erzeugt eine Liste der Zahlen von einschließlich `start` bis ausschließlich `end` mit Schritten der Größe `stride`. Beispielsweise steht `range(2, 5, 2)` für die Zahlen `2, 4`.

Auf die einzelnen Elemente dieser ranges kann wieder mit Index in [eckigen Klammern] zugegriffen werden:

Beispiel: ranges

```
1  rng_numbers = range(2, 5, 2)
2  print(rng_numbers[0], rng_numbers[1])
```

Ausgabe

```
2 4
```

ranges speichern nicht die Liste selbst, sondern nur die Daten, aus denen die Elemente generiert werden können. Man nennt sie daher auch *generator objects*. Mehr dazu später. Wo die einzelnen Objekte explizit gebraucht werden, kann wieder eine Type Conversion zu den bekannten Datenstrukturen eingesetzt werden:

Beispiel: Type conversion mit ranges

```
1  rng_numbers = range(2, 5, 2)
2  lst_numbers = list(rng_numbers)
3
4  print(rng_numbers)
5  print(lst_numbers)
```

Ausgabe

```
range(2, 5, 2)
[2, 4]
```

4.8. dictionaries

dicts verhalten sich wie lists, werden aber nicht über Zahlen sondern über beliebige *Schlüssel* indiziert. Beim Erstellen werden in {geschweiften Klammern} Schlüssel-Wert-Paare aufgeführt, die über einen Doppelpunkt miteinander verbunden werden:

Beispiel: `dicts`

```
houseStark = {"Sigil" : "A grey direwolf on a white field",
              "Words" : "Winter Is Coming",
              "Seat"  : "Winterfell"}

print(houseStark)
print(houseStark["Words"])
```

Ausgabe

```
{'Sigil': 'A grey direwolf on a white field', 'Words': 'Winter Is Coming',
 'Seat': 'Winterfell'}
Winter Is Coming
```

Wird versucht, den Wert eines Schlüssels zu lesen, der noch nicht angelegt wurde, so bewirkt dies eine Fehlermeldung. Schreibender Zugriff dagegen legt ein neues Schlüssel-Wert-Paar an:

Beispiel: `dicts` : neue Schlüssel

```
houseStark = {"Sigil" : "A grey direwolf on a white field",
              "Words" : "Winter Is Coming", "Seat"  : "Winterfell"}

# print(houseStark["Founder"]) -- Fehler: Schlüssel 'Founder' existiert nicht
houseStark["Founder"] = "Bran the Builder"
print(houseStark["Founder"])    # Kein Fehler: Schlüssel und Wert hinzugefügt
```

Ausgabe

```
Bran the Builder
```

Aus `dicts` lassen sich `lists`, `sets`, ... generieren. Dabei wird *per default* die Menge der Schlüssel übersetzt. Die Methoden `values`, `keys` und `items` erlauben aber auch, gezielt die Werte, Schlüssel oder Wert-Schlüssel-Paare abzugreifen:

Beispiel: `dicts` : spezielle Methoden

```
houseStark = {"Sigil" : "A grey direwolf on a white field",
              "Words" : "Winter Is Coming", "Seat"  : "Winterfell"}

l_keys1  = list( houseStark          )
l_keys2  = list( houseStark.keys()   )
l_values = list( houseStark.values() )
l_items  = list( houseStark.items()  )

print("direct conversion: ", l_keys1 )
print("method keys()    : ", l_keys2 )
print("method values()  : ", l_values)
print("method items()   : ", l_items )
```

Ausgabe

```
direct conversion:  ['Sigil', 'Words', 'Seat']
method keys()    :  ['Sigil', 'Words', 'Seat']
method values()  :  ['A grey direwolf on a white field', 'Winter Is Coming',
   'Winterfell']
method items()   :  [('Sigil', 'A grey direwolf on a white field'), ('Words',
   'Winter Is Coming'), ('Seat', 'Winterfell')]
```

dicts sind nicht zwingend geordnet

Die Zuordnung von Schlüssel zu Wert ist eine nicht-triviale Aufgabe, die im Hintergrund eine aufwändige Maschinerie betreibt. Damit diese schnell und effizient arbeiten kann, erlaubt der Python-Interpreter, dass die Reihenfolge der Schlüssel-Wert-Paare geändert wird. Sie können also leider nicht sicher sein, in welcher Reihenfolge Werte aus einem `dict` entnommen werden. Wo dies wichtig wird, benutzen Sie z. B. die Type-Conversion zu `list`s und die Methode `sort`.

4.9. Spezielle Funktionen für Container

in

Mit dem Schlüsselwort `in` kann geprüft werden, ob ein Objekt Teil einer Datenstruktur ist. Zurück gegeben wird ein `boolean`:

Beispiel: Operator `in`

```
1  myList = [1, 2, -5, 3.14, "some text", [3, 2], (1+1j)]
2
3  print(3.14 in myList)        # True
4  print("meaning" in myList)   # False
```

Insbesondere funktioniert das Schlüsselwort `in` auch mit `dict`s. Beachten Sie, auf welche Menge (Schlüssel, Werte, oder Schlüssel-Wert-Paare) Sie sich beziehen:

Beispiel: `dicts` : Operator `in`

```
1  houseStark = {"Sigil" : "A grey direwolf on a white field",
2                "Words" : "Winter Is Coming", "Seat"  : "Winterfell"}
3
4  print( "Words"      in houseStark )   # True
5  print( "words"      in houseStark )   # False -- Groß/Kleinschreibung
6  print( "Winterfell" in houseStark )   # False -- nur keys werden herangezogen
7  print( ("Seat", "Winterfell") in houseStark.items() )    # True
```

Vergleichsoperatoren

Alle Listen können miteinander verglichen werden. Dazu dienen die üblichen Operatoren `<`, `==`, `>`. Der Gleichheitsoperator `==` vergleicht alle Elemente der Listen miteinander, und gibt nur dann `True` zurück,

wenn alle Elemente übereinstimmen. Bei < und > wird – wie schon bei Strings – die *lexikographische* Vergleichsmethode angelegt, also „wie im Telefonbuch“. Betrachten Sie hierzu folgende Aufstellung:

```
"Aaron" < "Bart" < "Bartolomäus" < "Cäsar"
[1, 5, 2] < [1, 4] < [1, 4, 1] < [2, 1, 1]
```

Denken Sie auch daran, dass ein String wie ein `tuple` aus Buchstaben behandelt wird; so können Sie sich die Vergleichsregeln leicht ableiten.

`len`

Der Operator `len` gibt die Zahl der Elemente in einer Liste zurück. Das funktioniert für alle hier besprochenen Listen-Typen, inclusive `dict`s:

Beispiel: `dict`s : operator `len`

```
1  myList = [5, 9, 2]
2
3  print( len(myList) )    # 3
4  print( len([]) )        # 0
```

`reversed` und `sorted`

Wie die Namen schon andeuten, geben diese Funktionen in der Reihenfolge umgedrehte bzw. sortierte Listen zurück. Tatsächlich ist der Rückgabetyp beider Funktionen `list`. Im Gegensatz zu den *Methoden* `sort` und `reverse` wird aber eine Kopie angelegt:

Beispiel: `sort` vs. `sorted`

```
1  myList = [5, 9, 2]
2  sortedList = sorted(myList)
3
4  print("myList after sorted: ", myList)
5  print("sortedList         : ", sortedList)
6
7  myList.sort()
8  print("reversed(myList)   : ", reversed(myList))
9  print("myList after sort  : ", myList)
```

Ausgabe: `sort` vs. `sorted`

```
myList after sorted: [5, 9, 2]
sortedList         : [2, 5, 9]
reversed(myList)   : [9, 5, 2]
myList after sort  : [2, 5, 9]
```

Dies lässt sich auch auf `dict`s anwenden. Achten Sie darauf, dass ohne weitere Angaben nur die Schlüssel des `dict`s herangezogen werden:

Beispiel: dicts und sorted

```
houseStark = {"Sigil" : "A grey direwolf on a white field",
              "Words" : "Winter Is Coming", "Seat"  : "Winterfell"}

print( sorted(houseStark) )
print( sorted(houseStark.items()) )
```

Ausgabe: dicts und sorted

```
['Seat', 'Sigil', 'Words']
[('Seat', 'Winterfell'), ('Sigil', 'A grey direwolf on a white field'),
  ('Words', 'Winter Is Coming')]
```

Wie schon angesprochen ist die Anordnung der Elemente in dicts nicht fest. Daher kann auch die Funktion reversed nicht mit solchen aufgerufen werden.

zip

Mit dem Befehl zip können Listen zu Tabellen zusammengeschlossen werden. Jede einzelne Liste liefert die Daten einer Spalte. Das Ergebnis ist eine Datensammlung, die man zum Beispiel wieder zu einer *Liste von Zeilen* machen kann:

Beispiel: zip

```
categories     = ["House", "Sigil", "Words", "Seat"]
houseStark     = ["Stark", "A grey direwolf on a white field",
                  "Winter Is Coming", "Winterfell"]
houseLannister = ["Lannister", "A gold lion, on a crimson field",
                  "Hear Me Roar!", "Casterly Rock"]
houseTyrell    = ["Tyrell", "A golden rose on a green field",
                  "Growing Strong", "Highgarden"]

zipper = zip(categories, houseStark, houseLannister, houseTyrell)
houses = list(zipper)
print( houses )
```

Ausgabe: zip

```
[('House', 'Stark', 'Lannister', 'Tyrell'),
 ('Sigil', 'A grey direwolf on a white field', 'A gold lion,
   on a crimson field', 'A golden rose on a green field'),
 ('Words', 'Winter Is Coming', 'Hear Me Roar!', 'Growing Strong'),
 ('Seat', 'Winterfell', 'Casterly Rock', 'Highgarden')]
```

min und max

Wie der Name vermuten lässt, durchsuchen diese beiden Befehle einen Container nach seinem kleinsten bzw. größten Element. Dabei werden dieselben Regeln angewandt, die schon für die Vergleichsoperatoren (< und >) besprochen wurden:

Beispiel: `min` und `max`

```
1  dataNumbers = [4, 3.7, -2112]
2  dataStrings = ["Victoria", "Epsi", "Charlotte"]
3  dataLists   = [ [1], [1, 2], [2, 2]]
4
5  print( min(dataNumbers), max(dataNumbers) )
6  print( min(dataStrings), max(dataStrings) )
7  print( min(dataLists  ), max(dataLists  ) )
```

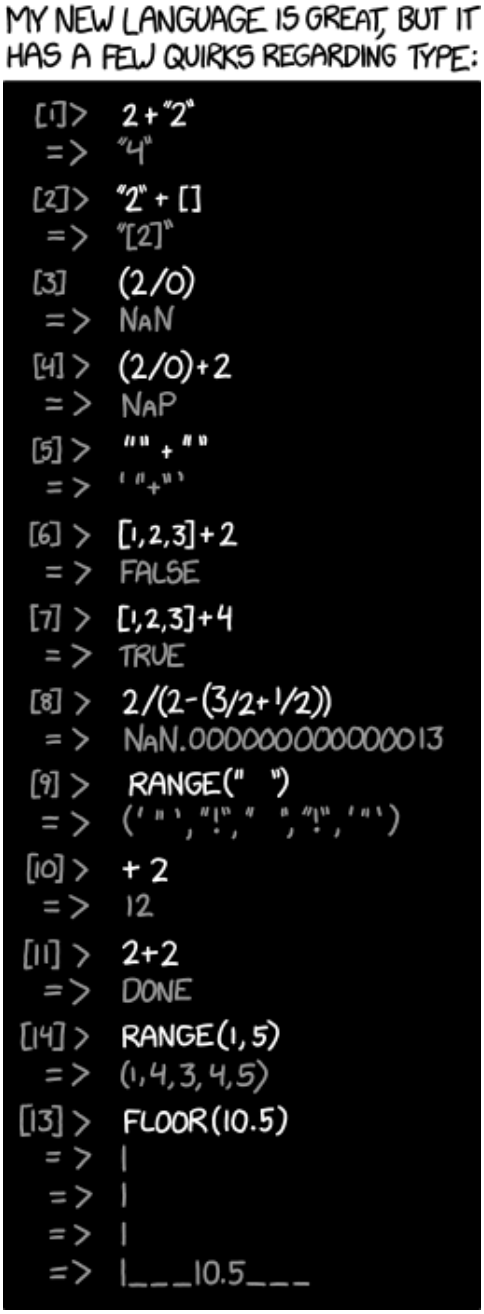
Ausgabe: `min` und `max`

```
-2112 4
Charlotte Victoria
[1] [2, 2]
```

4.10. Überblick

Datenstruktur	Klammern	Mutable	Besonderer Nutzen
`list`	[eckig]	ja	All-Purpose-Listentyp
`tuple`	(rund)	nein	Schreibgeschützte Listen
`set`	{geschweift}	ja	keine Doubletten
`frozenset`	{geschweift} mit Präfix erstellt aus Type Conversion	nein	Schreibgeschützte Listen ohne Doubletten
String	''Doppelte Anführungszeichen''	nein	Texte
`range`	– keine –	nein	Bereiche von Ganzzahlen
`dict`	{geschweift} Doppelpunkt trennt Schlüssel : Wert	ja	Zuordnungen

Tabelle 4.1.: Überblick über die verschiedenen Container-Datentypen in Python



„`colors.rgb(''blue'')` yields `''#0000FF''`. `colors.rgb(''yellowish blue'')` yields `NaN`. `colors.sort()` yields `''rainbow''`“

Abbildung 4.4.: Eingabeschemata in anderen Programmiersprachen
Quelle: `https://xkcd.com/1537/`

5. Schleifen

> Insanity: doing the same thing over and over again and expecting different results
>
> Albert Einstein

Computer können dazu benutzt werden, die immer gleichen (lästigen) Aufgaben wiederholt und in schneller Folge auszuführen. Es ist möglich, bei jeder Wiederholung einen einzelnen Eingabewert zu ändern und so z. B. Berechnungen für einen ganzen Wertebereich durchzuführen, oder Messwerte von einem Gerät zu überwachen.

Zeichnet man ein *Flussdiagramm* eines solchen Programms (wie in Abbildung 5.1), so findet sich in der Regel ein Programmteil, der zur Vorbereitung dient und in gewohnter Weise „von oben nach unten“ abgearbeitet wird. An diesen schließt sich ein Abschnitt an, der einige Male wiederholt werden soll, und daher im Flussdiagramm als Bogen dargestellt wird. Nach diesem Teil könnte die Ausgabe der Ergebnisse stattfinden, die wiederum in gewohnter *linearer* Weise (also von oben nach unten) bearbeitet wird.

Die Form dieses Flussdiagramms motiviert den Namen *Schleife* für eine solche Struktur.

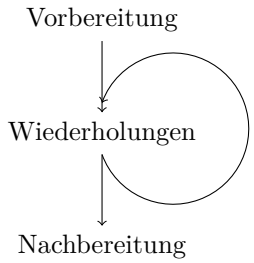


Abbildung 5.1.: Programmflussdiagramm mit Schleife

In der Regel ist die Zahl der Schleifendurchläufe an eine Bedingung geknüpft; genauso sind aber auch *Endlosschleifen* möglich. In diesem Kapitel werden wir verschiedene Schleifentypen und ihre Anwendungsfelder kennen lernen.

Laufende Programme zum Beenden zwingen: `STRG + C`

Macht man einen Fehler bei der Formulierung der Bedingung, so kann man unbeabsichtigt eine Endlosschleife erstellen. Ein solches Programm wird von sich selbst aus nie beendet. Wir können zu jeder Zeit aber das Beenden erzwingen, indem wir in der Konsole die Tastenkombination `STRG + C` drücken.

5.1. `while`-Loops

5.1.1. Grundstruktur

Mit dem Schlüsselwort `while` wird ein Codeblock eingeleitet, der so lange wiederholt wird, bis eine Bedingung nicht mehr erfüllt ist, d. h. bis ihr Wahrheitswert zu `False` ausgewertet wird. Die Syntax lautet:

Syntax: `while` (1)

```
while Wahrheitswert :
    Schleifenkoerper
```

Betrachten Sie hierzu folgendes Beispiel zum Zinseszins: „berechnet“ wird, nach wie vielen Jahren ein Startkapital bei gegebenem Zinssatz über einen Grenzwert hinauswächst[1].

Beispiel: Zinseszins (1)

```
1  capital  = float(input("Bitte geben Sie Ihr Startkapital ein: "))
2  interest = float(input("Bitte geben Sie den Zinssatz ein: "    ))
3  limit    = float(input("Bitte geben Sie Ihr Zielkapital ein :" ))
4
5  years    = 0
6
7  while capital < limit :
8      capital *= 1 + interest
9      years   += 1
10
11 print("Nach", years, "Jahren ist das Sparziel erreicht.")
```

Nachdem die Variablen `capital, interest, limit` und `years` ihre Werte zugewiesen bekommen haben, wird überprüft, ob `capital < limit`. Wenn dies erfüllt ist, werden die Zeilen 8 und 9 ausgeführt. Danach springt die Codeausführung zurück zu Zeile 7. Es wird solange der Schleifenkörper in Zeile 8 und 9 wiederholt, bis die Bedingung in Zeile 7 nicht mehr erfüllt ist. Erst dann wird mit der Ausführung in Zeile 11 fortgesetzt. So erklärt sich folgendes Ausführungsbeispiel:

Ausführungsbeispiel: Zinseszins (1)

```
Bitte geben Sie Ihr Startkapital ein: 100
Bitte geben Sie den Zinssatz ein: .1
Bitte geben Sie Ihr Zielkapital ein :300
Nach 12 Jahren ist das Sparziel erreicht.
```

Der Schleifenkörper wird nie ausgeführt, wenn die Bedingung nicht bereits vor Beginn der Schleife erfüllt war. Geben Sie im obigen Beispiel etwa ein Startkapital ein, das größer als das Zielkaptital ist, so folgt die Ausgabe `Nach 0 Jahren ist das Sparziel erreicht.` Die Zeile `years += 1` wird nie ausgeführt.

[1] Natürlich ließe sich dies auch über die Logarithmus-Funktion realisieren; hier aber soll die Funktionsweise von Schleifen gezeigt werden

Endlosschleifen

Manchmal möchte man, dass ein Programm auf unbestimmte Zeit läuft. Code, der die Messwerte einer Wetterstation verarbeitet, soll dies vielleicth „für immer“ tun. In diesem Fall bietet sich eine Endlosschleife an:

Code: Endlosschleife

```
1  while True:
2    Anweisungen
```

Da `True` eine Konstante ist, wird sich nichts daran ändern, dass sie immer eben zu `True` ausgewertet wird. Das Programm läuft bis in alle Ewigkeiten (oder zumindest, bis es vom Betriebssystem beendet wird). Üblicherweise enthält `Anweisungen` dann Code, der das Programm dennoch zu einem Ende führt. Diese Anweisungen könnten aber an komplexe Bedingungen geknüpft sein, die für die Form von `while` zu sperrig sind.

5.1.2. Eingriff in den Kontrollfluss

Es gibt Situtaionen, wo die Ausführung einer Schleife an mehrere, voneinander unabhängige Bedingungen geknüpft sind. Dies lässt sich mit logischen Operatoren (Siehe Abschnitt 2.2.1) umsetzen. Häufig ist es aber übersichtlicher, das Schlüsselwort `break` in einem eigenen `if`-Block zu benutzen. Mit `break` wird der Schleifenkörper verlassen, die Ausführung wird hinter der Schleife fortgesetzt, ganz als ob die Bedingung hinter `while` selbst nicht mehr erfüllt wäre. Zusätzlich zur Übersichtlichkeit bietet diese `break`-Methode die Möglichkeit, an einem beliebigen Punkt innerhalb der Schleife zu prüfen, ob der Schleifenkörper verlassen werden soll.

Beispiel: `break`

```
1  count = 0
2  value = 0
3
4  while count < 12 :
5      newValue = float(input("Bitte geben Sie einen weiteren Wert ein: "))
6      value += newValue
7      count += 1
8
9      if value > 9000 :
10         value = "over nine thousand!!"
11         break
12
13     print(
14         "Bisheriger Gesamtwert nach Eingabe von", count,
15         "Werten:", value
16     )
17
18 print("Gesamtwert:", value)
```

Das Einlesen von `newValue` sowie die Updates an `value` und `count` werden bei jedem Durchlauf der Schleife durchgeführt. Die Ausgabe des Zwischenstandes (Zeilen 13-16) hingegen nur, wenn die Zweite Bedingung (`value > 9000`) nicht erfüllt war. So erklärt sich folgendes Ausführungsbeispiel:

Ausführungsbeispiel: `break`

```
Bitte geben Sie einen weiteren Wert ein: 1
Bisheriger Gesamtwert nach Eingabe von 1 Werten: 1.0
Bitte geben Sie einen weiteren Wert ein: 50
Bisheriger Gesamtwert nach Eingabe von 2 Werten: 51.0
Bitte geben Sie einen weiteren Wert ein: 9000
Gesamtwert: over nine thousand!!
```

Ähnlich kann **`continue`** benutzt werden, um einen Teil des Schleifenkörpers zu überspringen, ohne die Schleife selbst zu verlassen. Der Befehl **`continue`** springt also zur Überprüfung der **`while`**-Bedingung zurück:

Beispiel: `continue`

```
1  container = []
2
3  while len(container) < 3 :
4      value = int(input("Bitte geben Sie eine gerade Zahl ein: "))
5
6      if value % 2 :
7          print(value, "ist ungerade und daher nicht zulässig.")
8          continue
9
10     container.append(value)
11     print("Bisher eingegeben: ", container)
```

Ausführungsbeispiel: `continue`

```
Bitte geben Sie eine gerade Zahl ein: 2
Bisher eingegeben:  [2]
Bitte geben Sie eine gerade Zahl ein: 3
3 ist ungerade und daher nicht zulässig.
Bitte geben Sie eine gerade Zahl ein: 4
Bisher eingegeben:  [2, 4]
Bitte geben Sie eine gerade Zahl ein: 6
Bisher eingegeben:  [2, 4, 6]
```

Garantierte erste Ausführung

Wir können Endlosschleifen zusammen mit **`break`** nutzen, um zu garantieren, dass der Schleifenkörper mindestens einmal ausgeführt wird, unabhängig davon, ob die Schleifenbedingung zu Anfang bereits erfüllt war:

Beispiel: `continue`

```
1  while True:
2      Schleifenkoerper
3      if Bedingung :
4          break
```

Die Überprüfung auf `Bedingung` wird so an das Schleifenende geschoben.

5.1.3. `else` bei `while`

Wie schon angesprochen wird der Schleifenkörper nicht ausgeführt, wenn die Bedingung schon vor der ersten Durchführung nicht erfüllt war. In diesem Fall kann ein optionaler `else`-Block ausgeführt werden:

Syntax: `while` (2)

```
while Wahrheitswert :
    Schleifenkoerper
else :
    AlternativCode
```

Betrachten wir dies an einem erweiterten Beispiel:

Beispiel: Zinseszins (2)

```
1  capital  = float(input("Bitte geben Sie Ihr Startkapital ein: "))
2  interest = float(input("Bitte geben Sie den Zinssatz ein: "    ))
3  limit    = float(input("Bitte geben Sie Ihr Zielkapital ein :" ))
4
5  years    = 0
6
7  while capital < limit :
8      capital *= 1 + interest
9      years   += 1
10 else :
11     print("Das Startkapital ist bereits groß genug.")
12
13 print("Nach", years, "Jahren ist das Sparziel erreicht.")
```

Die Zeile `Das Startkapital ist bereits groß genug.` wird genau dann ausgegeben, wenn für `capital` ein Wert größer oder gleich `limit` eingegeben wurde. Weiterhin wird für diesen Fall `Nach 0 Jahren ist das Sparziel erreicht.` als zweite Zeile ausgegeben.

Garantierte erste Ausführung mit `else`

Statt der oben gezeigten Form mit `break` könnte auch mit `else` dafür gesorgt werden, dass der Schleifenkörper mindestens einmal ausgeführt wird. Hierzu kopiert man einfach den kompletten Code des Schleifenkörpers in den `else`-Block.

Dies erfüllt zwar seinen Zweck, ist aber eine Fehlerquelle und sollte vermieden werden. Wenn Sie später Codestellen ändern, müssen Sie auch daran denken, dieselben Änderungen im `else`-Block umzusetzen. Dies wird leicht vergessen.

5.2. `for`-Loops

5.2.1. Grundstruktur

Schleifen mit `for` führen Code für jedes Element aus einer Datenstruktur wie in Kapitel 4 beschrieben aus. Nacheinander wird jedes Element des Containers über eine Hilfsvariable ansprechbar gemacht und

dann Code ausgeführt, der von dieser Hilfsvariablen abhängig sein kann.

Syntax: `for`

```
for Variable in Container :
    Schleifenkoerper
```

Der Code lässt sich also fast wie englische Sprache lesen: „für jedes Objekt in *Container* mache *Schleifenkörper*“.

Beispiel: `for` (1)

```
1  tasklist = ["write the script", "drink some coffee", "drink some more coffee"]
2
3  print("Your tasks today:")
4  for task in tasklist :
5      print("*", task)
```

Ausgabe: `for` (1)

```
Your tasks today:
* write the script
* drink some coffee
* drink some more coffee"
```

Als *Container* dienen besonders häufig `range`-Objekte. Wann immer eine durchzählbare Menge von Zahlen gebraucht wird, kann ein solches `range`-Objekt genutzt werden[2]:

Beispiel: `for` (2)

```
1  print("the first 10 square numbers are:")
2  for i in range(10) :
3      print(f"{i}² = {i**2}")
```

Ausgabe: `for` (2)

```
the first 10 square numbers are:
0² = 0
1² = 1
2² = 4
3² = 9
4² = 16
5² = 25
6² = 36
7² = 49
8² = 64
9² = 81
```

[2]Das folgende Beispiel und einige weitere enthalten die Option `sep=""`. Damit wird der Funktion `print` mitgeteilt, dass kein Leerzeichen zwischen den einzelnen Werten gedruckt werden soll. Vorerst können Sie diese Option ignorieren. In Kapitel 6 wird diese Technik erklärt.

Falls Sie von einer anderen Sprache kommen: Keine Indices

Das Konzept einer for-Schleife existiert in fast allen Programmiersprachen. Häufig ist dort die Funktionalität jedoch auf Zahlen eingeschränkt. Will man *über die Elemente eines Containers iterieren*, muss man dort die Schleife über die Indices laufen lassen. Das sieht dann beispielsweise so aus:

Beispiel: `for` mit Indices

```
tasklist = ["write the script", "drink some coffee", "drink more coffee"]
N = len(tasklist)

print("Your tasks today:")
for i in range(N) :
    print("*", tasklist[i])
```

Dieses Beispiel funktioniert zwar, hat aber mehr Fehlerquellen. Man muss mindestens dafür sorgen, dass der Wert `N` zu jeder Zeit die korrekte Anzahl von Elementen in `tasklist` enthält. Sprachen wie C oder BASIC machen solche Strukturen leider nötig. In Python dagegen können wir diese Aufgabe getrost dem Interpreter überlassen.

Falls sie bereits eine andere Sprache kennen, sind sie es vielleicht gewohnt, for-Schleifen über Indices laufen zu lassen. Gewohnen Sie sich dies in Python ab. Nicht nur vermeiden Sie so eine Fehlerquelle; Ihr Code wird tatsächlich auch performanter, wenn Sie die „Python-Hausmittel“ voll ausschöpfen.

In einem `for`-Befehl können auch Container entpackt werden:

Beispiel: `for` (3)

```
books = [
    ("Frank Herbert", "Dune"),
    ("Douglas Adams", "The Hitchhikers Guide To The Galaxy"),
    ("Randall Munroe", "What If"),
    ("Isaac Asimov", "Foundation"),
    ("Willy Russell", "Educating Rita"),
    ("Moving Pictures", "Terry Pratchett")
]

print("You should definitively read:")
for author, title in books :
    print("*", title, "by", author)
```

Ausgabe: `for` (3)

```
You should definitively read:
* Dune by Frank Herbert
* The Hitchhikers Guide To The Galaxy by Douglas Adams
* What If by Randall Munroe
* Foundation by Isaac Asimov
* Educating Rita by Willy Russell
* Moving Pictures by Terry Pratchett
```

Natürlich *müssen* Tupel *nicht* enptackt werden:

Beispiel: `for` (4)

```
import math
vectors = [(1, 1), (4, 7), (-1, 2)]

for v in vectors :
  print("vector", v, "has length", math.hypot(v[0], v[1]))
```

Ausgabe: `for` (4)

```
vector (1, 1) has length 1.4142135623730951
vector (4, 7) has length 8.06225774829855
vector (-1, 2) has length 2.23606797749979
```

Objekt und Index: `enumerate`

Manchmal wird sowohl das Objekt als auch seine Position in der Liste benötigt. Mit der Funktion `enumerate` lässt sich ein Tupel aus genau diesen beiden Objekten erzeugen:

Beispiel: `for` mit `enumerate`

```
tasklist = ["write the script", "drink some coffee", "drink more coffee"]

print("Your tasks today:")
for i, task in enumerate(tasklist) :
    print(f"{i + 1}. {task}")
```

Ausgabe: `for` mit `enumerate`

```
Your tasks today:
1. write the script
2. drink some coffee
3. drink more coffee
```

Beachten Sie, dass die Zahlen, wie von `enumerate` erzeugt bei `0` beginnen.

Iteration über zwei Listen zugleich: `zip`

Nicht selten müssen die Daten in zwei Listen zueinander in Bezug gesetzt werden. Stellen Sie sich beispielsweise vor, Sie haben zwei Messreihen aufgenommen, und wollen nun die Abweichungen dieser Messreihen berechnen. Hierzu können Sie den Ihnen bereits bekannten Befehl `zip` benutzen:

Beispiel: `for` mit `zip`

```
data1 = [1.7, 2.2, -4.1]
data2 = [1.8, 2.0, -3.8]

print("Difference in datasets:")
for i, t in enumerate(zip(data1, data2)) :
    print(f"Datapoint {i}: {t} differs by : {t[0] - t[1]}")
```

Ausgabe: `for` mit `zip`

```
Datapoint 0: (1.7, 1.8) differs by : -0.10000000000000009
Datapoint 1: (2.2, 2.0) differs by : 0.20000000000000018
Datapoint 2: (-4.1, -3.8) differs by : -0.2999999999999998
```

Wie erwähnt können *alle* Container für `for`-Schleifen verwendet werden, die wir aus Kapitel 4 kennen[3]. Als Beispiel sei die Ausgabe eines `dict`s gezeigt:

Beispiel: `for` mit `dict`s

```
houseStark = {"Sigil" : "A grey direwolf on a white field",
              "Words" : "Winter Is Coming",
              "Seat"  : "Winterfell"}

print("Summary of House Stark:")
for key, value in houseStark.items() :
    print(f"{key:5}: {value}")
```

Ausgabe: `for` mit `dict`s

```
Summary of House Stark:
Sigil: A grey direwolf on a white field
Words: Winter Is Coming
Seat : Winterfell
```

5.2.2. Eingriffe in den Kontrollfluss, `else`

Auch bei `for` können die Befehle `break` und `continue` eingesetzt werden, und verhalten sich dort genauso, wie bei `while`. Außerdem existiert auch bei `for` eine optionale `else`-Klausel. Der Code hierin wird ausgeführt, wenn die `for`-Schleife normal zum Ende kam, also *nicht* durch `break` abgebrochen wurde.

[3] In Kapitel 7 werden wir noch Möglichkeiten kennen lernen, weitere *iterierbare* Objekte zu erstellen.

Beispiel: `for` mit `break`, `continue` und `else`

```
1  tasks = [
2      ("hidden", "watch Fullmetal Alchemist"),
3      ("open", "work very hard on Python"),
4      ("open", "drink all the coffee")
5  ]
6  search = ["watch", "work"]
7
8  for keyword in search :
9      for ID, (state, task) in enumerate(tasks) :
10         if state == "hidden" : continue
11         if keyword in task :
12             print(keyword, "was found in task ID", ID)
13             break
14     else :
15         print(keyword, "was not found in the tasks.")
```

Ausgabe: `for` mit `break`, `continue` und `else`

```
watch was not found in the tasks.
work was found in task ID 1
```

Machen Sie sich klar, was hier passiert: In der äußeren Schleife sorgen wir dafür, dass wir nacheinander nach zwei Begriffen in der Aufgabenliste suchen. Diese Begriffe werden mit `keyword` „greifbar gemacht“.

Für die innere Schleife betrachten wir ein Tupel aus einer ID und einem weiteren Tupel, das die Elemente von `tasks` umfasst. Hierfür verwenden wir die Symbole `ID`, `state` und `task`. Da `state` und `task` aus den Elementen von `tasks` gebildet werden (also für sich eine eigene Einheit bilden), müssen diese in Klammern gefasst werden.

Für jedes Element aus `tasks` wird zunächst der `state` betrachtet. Ist dieser gleich `"hidden"`, so wird das Element in der Analyse übersprungen (wir führen `continue` aus.) Andernfalls wird geprüft, ob `keyword` in der Aufgabenbeschreibung `task` gefunden wurde. Ist dies der Fall, so ist die Analyse des aktuellen Elements der Aufgabenliste `tasks` abgeschlossen (wir führen `break` aus), und das nächste `keyword` kann betrachtet werden.

In dem Fall, wo das `keyword` gefunden werden kann (also z. B. für `"work"`) wird also ein `break` ausgelöst und folglich der Code bei `else` übersprungen. Dagegen löst das `keyword` `"watch"` den `if`-Block in Zeile 11 nicht aus (da die Prüfung mit Zeile 10 bereits übersprungen wird). Daher läuft die `for`-Schleife vollständig durch, und der `else`-Block wird ausgeführt.

5.3. List Comprehension

Stellen Sie sich vor, Sie brauchen eine Liste mit den Quadraten aller geraden Ganzzahlen. Sie können diese Aufgabe jetzt schon so lösen:

Beispiel: `for` zum Erstellen einer Liste

```
evenSquares = []
N = 20

for i in range(2, N, 2) :
    evenSquares.append(i**2)

print(evenSquares)
```

Ausgabe: `for` zum Erstellen einer Liste

```
[4, 16, 36, 64, 100, 144, 196, 256, 324]
```

Dieselbe Konstruktion können Sie verkürzt auch schreiben als:

Beispiel: List Comprehension

```
N = 20
evenSquares = [ i**2 for i in range(2, N, 2) ]
print(evenSquares)
```

Die Abstrakte Syntax lautet also:

Syntax : List Comprehension (1)

```
variable = [ expression for element in iterable ]
print(evenSquares)
```

Dabei sind:

- `iterable` ein beliebiger Datencontainer, wie in allen vorigen Beispielen
- `element` eine Variable, über die nacheinander die einzelnen Elemente aus `iterable` durchgelesen werden. Auch hier können Tupel entpackt werden. In diesem Fall müssen entsprechend mehrere Variablen genannt werden.
- `expression` ist ein beliebiger Ausdruck, der die zu erzeugenden Listenelemente beschreibt.

List Comprehension kann auch ineinander verschachtelt werden (wird dann aber schnell unübersichtlich). Die folgende Zeile erzeugt die sogenannte „Telefonmatrix“:

Beispiel: Telefonmatrix

```
telephone = [
    [3 * row + column + 1 for column in range(3)]
    for row in range(3)
]

for line in telephone :
    print(line)

print(telephone)
```

Der Code erzeugt also eine *Liste von Listen*:

Ausgabe: Telefonmatrix

```
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Sprechende Variablennamen

Im obigen Beispiel *Telefonmatrix* taucht die Variable `row` zum ersten Mail in Zeile 2 auf, obwohl erst in Zeile 3 erklärt wird, welche Werte `row` haben wird, oder um welche Art von Werten (Integer, Fließkommazahlen, Strings, Container, ...) es sich überhaupt handelt. *Strukturell* kommen wir also in die Gefahr von schwer lesbarem Code, und haben damit eine Fehlerquelle. Durch die Benennung der Variablen ist aber dennoch schnell klar, was hier passiert. Die Variablen `row` und `column` bezeichnen die Nummer von Zeile und Spalte einer Matrix, sind also Integers zwischen 0 und einer Obergrenze, die sich aus der Größe der Matrix ergibt.

Solche Überlegungen zur Benennung der Objekte sind extrem wichtig in auch nur mittelgroßen Projekten. Geben Sie nicht der Versuchung nach, Variablen aus Bequemlichkeit einfach `a, b, c, ...` zu nennen, sondern überlegen Sie, welche Art von Information darin abgelegt werden soll, und schreiben dies dann auch aus! Auch Abkürzungen sollten vermieden werden, da sich hierbei oft Fehler einschleichen. Hieß die Variable nun `numElmL` oder `nel` oder `nElements`? Diese Frage stellt sich nicht, wenn Sie sich die kleine Mühe machen, konsequent `numberElementsList` zu tippen; Sie sparen sich dadurch die große Mühe, häufig zurück zu scrollen und Fehler auszumerzen, die sich daraus ergeben, dass Sie mal `numElmL` und mal `nel` getippt haben.

Bei der List Comprehension können Sie auch die Aufnahme eines Wertes in die Ergebnis-Liste an eine Bedingung knüpfen. Sie erreichen dies, indem Sie einfach eine `if`-Klausel anfügen:

Syntax : List Comprehension (2)

```
variable = [ expression for element in iterable if condition ]
print(evenSquares)
```

Als Beispiel soll eine Liste aller Primzahlen bis zu einer Obergrenze `N` berechnet werden. Wir nutzen dazu aus, dass eine Primzahl exakt zwei Ganzzahl-Teiler hat. Wir erstellen also zuerst für jede Zahl zwischen 2 und `N` alle Ganzzahl-Teiler, und akzeptieren nur solche Zahlen in unserem Ergebnis, bei dem diese Liste der Teiler die Länge `2` hat[4].

[4] Der gezeigte Code ist sowohl bezüglich Rechenzeit als auch bezüglich Speicherbedarf schlecht, illustriert aber schön die Technik, um die es uns hier geht.

Beispiel: List Comprehension mit Bedingung

```
N = 100
primeNumbers = [
  i for i in range (2, N)        # übernehme alle Zahlen i zwischen 2 und N ...
  if len (                       # ... für die die die Anzahl der Teiler ...
    [j for j in range (1, i+1)   # ... (d.h. die Zahlen zwischen 1 und i ...
    if i % j == 0]               # ... die i ganzzahlig teilen) ...
  ) == 2                         # ... gleich 2 ist.
]

print(primeNumbers)
```

Ausgabe: List Comprehension mit Bedingung

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73,
79, 83, 89, 97]
```

List Comprehension lässt sich auch für `set`s und `dict`s umsetzen. Die Synax verlangt dann nur eine minimale Anpassung:

Syntax : List Comprehension (3)

```
setVariable  = {       Ausdruck for Element in Container if Bedingung }
dictVariable = { key : Ausdruck for Element in Container if Bedingung }
```

Natürlich sind die `if`-Klauseln auch für `set`s und `dict`s optional.

Beispiel: List Comprehension für `set`s und `dict`s

```
# Die ersten 10 Quadratzahlen als set:
print( {i**2 for i in range (10)} )
# Die ersten 10 Quadratzahlen als dict
print( {i : i**2 for i in range (10)} )
```

Ausgabe: List Comprehension für `set`s und `dict`s

```
{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Aufwändige Berechnungen vorbereiten

In wissenschaftlichen Simulationen treten manche Ausdrücke wie e^k gehäuft für die immer gleichen Werte von k auf. Anstatt die Exponentialfunktion immer wieder neu auszuwerten lohnt es sich oft, die Werte einmal in einer Liste vorzubereiten und so den Rechenaufwand zu minimieren. List Comprehension bietet sich hierzu perfekt an.

Für besonders gleichmäßige Verteilungen (d. h. wenn k alle Ganzzahlen zwischen 0 und einer Obergrenze sind) sollten `list`s verwendet werden, da diese leichter ausgelesen werden können. Allgemeinere Lookup-Tables lassen sich gut in `dict`s abbilden.

6. Funktionen

> Before software can be reusable it first has to be usable.
>
> Ralph Johnson

Viele Aufgaben wiederholen sich, oder lassen sich verallgemeinern. Folgender Code berechnet beispielsweise den Wert von Eulers Zahl, also exp(1), kann aber auch dazu verwendet werden, andere Potenzen von e zu finden:

Beispiel: Exponentialfunktion

```
1  x0           = 1.0
2  x            = 1.0
3  denominator  = 1.0
4  EulersNumber = 1.0
5  iterations   = 10
6
7  for k in range(1, iterations) :
8      x            *= x0
9      denominator  *= k
10     EulersNumber += x / denominator
11
12 print(f"exp({x0}) is approximately {EulersNumber}")
```

Nun ist es zwar schön, eine solche allgemeine Rechenvorschrift zu haben; wir müssen nur den Wert für `x0` in der ersten Zeile austauschen, und erhalten ein neues Ergebnis. Wollen wir aber an mehreren Stellen die Exponentialfunktion verwenden, so müssten wir diesen Code jedes Mal kopieren und neu einfügen. Schlimmer noch: sollte uns auffallen, dass wir im obigen Code einen Fehler gemacht haben, so müssen wir diesen *für jede Kopie separat* ausmerzen. Und selbst, wenn wir auf Anhieb perfekt gearbeitet haben, „belegt“ unser Code jetzt u. a. die Symbole `x` und `x0`. Projekte, die diesen Code verwenden wollen, müssen also andere Namen finden.

Natürlich wissen Sie, dass es im Modul `math` die *Funktion* `math.exp` gibt, die ebenfalls die Exponentialfunktion berechnet, aber ohne all diese lästigen Nebeneffekte auskommt. Hier sollen Sie lernen, wie dies möglich ist, und wie Sie solche Funktionen selbst erstellen.

6.1. Grundlagen

Funktionen sind „abgetrennte Codebereiche“, die unabhängig vom restlichen Code ausgeführt werden. In eine Funktion können Parameter (oft auch Argumente genannt) eingespeist werden (man spricht von *übergeben*), und ein Ergebnis wird von Funktionen *zurückgegeben*. Zur Berechnung dieses Ergebnisses wird normaler Code ausgeführt; dieser verhält sich aber „wie in einer Box“, d. h. verwendet seine eigenen, *lokalen* Variablen. Diese können dieselben Namen tragen wie Variablen an anderen Stellen Ihres Programms; nichts desto trotz sind die Variablen in einer Funktion unabhängig vom restlichen

Code. (Ebenso, wie es zwei Menschen mit dem Namen *Theophrastus* geben kann[1], die aber dennoch zwei getrennte Personen sind, können also auch zwei Variablen mit demselben Namen vorliegen, ohne *dieselbe* Variable zu sein. Stellen Sie sich vor, eine Variable hat einen „Vor- und Nachnamen“. Der Vorname ist dabei das Symbol, das Sie benutzen, um die Variable zu benennen. Der Nachname ist dann der Kontext, also die Funktion, in der dieses Symbol gerade verwendet wird.)

Ein solcher Codebereich wird eingeleitet durch eine `def`-Zeile und wird durch Einrückung vom restlichen Code abgetrennt, ähnlich, wie wir das schon von `if`-Blöcken und `for`-Schleifen kennen:

Syntax: Funktionen

```
def Funktionsname (Parameterliste) :
    normaler Code
    ...
    return Ergebnis
```

`Parameterliste` ist eine durch Kommata getrennte Liste von Variablennamen. Über diese Variablennamen werden „Informationen von außen in die Funktion eingeschleust“ (Übergabe von Werten). Verstehen Sie die Parameterliste als einen Punkt, an dem Variablen angelegt werden. Die Werte dieser Variablen werden beim Aufruf der Funktion festgelegt, und können in der Funktion (und nur dort) wie normale Variablen verwendet werden. Eine Parameterliste kann auch leer sein. In dem Fall müssen aber immer noch leere Klammern () gesetzt werden.

Wie besprochen steht innerhalb einer Funktion Code, wie Sie ihn schon kennen bzw. noch kennenlernen werden. Alle Mittel, die Sie bisher gesehen haben auch auch solche, wie wir sie im Weiteren noch besprechen, können auch innerhalb von Funktionen verwendet werden.

In der Regel soll eine Funktion ein Ergebnis berechnen, z. B. den Wert der Exponentialfunktion. Dieses Ergebnis wird mit dem Befehl `return` zurück an das Hauptprogramm[2] geschickt und zugleich die Funktion verlassen. Aller Code in der Funktion, der hinter `return` steht, wird also ignoriert. Eine Funktion darf durchaus mehrere `return`-Anweisungen haben. Sinnvoll kann das sein, wenn eine Fallunterscheidung gemacht wird (z. B. mit `if`). Genauso ist es auch zulässig, überhaupt keine `return`-Anweisung zu benutzen. Stellen Sie sich zum Beispiel eine Funktion vor, die lediglich eine formatierte Ausgabe auf dem Bildschirm bewirken soll. Hier wird kein Ergebnis berechnet. Entsprechend kann auch die `return`-Anweisung entfallen.

Aufgerufen (d. h. ausgeführt) wird die Funktion durch Nennung ihres Namens. In Klammern dahinter werden die Werte aufgeführt, die über die Parameterliste an die Funktion übergeben werden sollen. Für jede Variable der Parameterliste muss auch ein Wert angegeben werden. Ist die Parameterliste leer, so müssen dennoch leere Klammern () beim Funktionsaufruf stehen.

Ein Funktionsaufruf macht also mehrere Dinge:

- Eine *Kopie* der Werte in der Parameterliste wird an die Funktion übergeben
- Die Programmausführung springt in die Funktion
- Nach Ende der Funktion wird das Programm an der ürsprünglichen Stelle fortgeführt.

Nach Ende des Funktionsaufrufs wird an der aufrufenden Stelle ein Wert erhalten (eben der Rückgabewert der Funktion). Dieser kann in einer Variablen gespeichert werden, oder an andere Funktionen weitergegeben werden.

[1] citation needed
[2] bzw. an die aufrufende Stelle – aus Funktionen heraus können auch Funktionen aufgerufen werden.

Beispiel: Exponentialfunktion als Funktion

```
def expFunction(x0) :
    x           = 1.0
    denominator = 1.0
    result      = 1.0
    iterations  = 10

    for k in range(1, iterations) :
        x           *= x0
        denominator  *= k
        result += x / denominator

    return result

EulersNumber = expFunction(1.0)              # Aufruf, Speichern in Variable
print(f"Eulers Number is {EulersNumber}")
print(f"e³ = {expFunction(3.0)}")            # Weitergabe an andere Funktion
```

Ausgabe: Exponentialfunktion als Funktion

```
Eulers Number is 2.7182815255731922
e³ = 20.063392857142855
```

Wird kein Rückgabewert genannt, so weist Python automatisch den Pseudo-Rückgabewert `None` zu. Es handelt sich dabei um eine Konstante, die eben genau für die Abwesenheit eines Wertes steht, und sollte auch nicht mit dem Text `"None"` verwechselt werden.

Beispiel: Funktion „ohne“ Rückgabewert

```
import math

def printBoxed(text, boxSize) :
    # Draws text in a box like:
    #
    # +------------+
    # |   text     |
    # +------------+

    lenText     = len(text)
    countSpaces = boxSize - lenText - 2         # 2 spaces for |borders|
    spacesLeft  = math.floor(countSpaces / 2)   # round down
    spacesRight = math.ceil (countSpaces / 2)   # round up

    print("+" +                (boxSize - 2) * "-"            + "+")
    print("|" + spacesLeft * " " + text + spacesRight * " " + "|")
    print("+" +                (boxSize - 2) * "-"            + "+")

reVal = printBoxed("Don't forget to be awesome!", 60)
print("The function returned:" , reVal)
```

Ausgabe: Funktion „ohne“ Rückgabewert

```
+------------------------------------------------------------+
|               Don't forget to be awesome!                  |
+------------------------------------------------------------+
The function returned: None
```

Wie erwähnt wird die Ausführung der Funktion sofort beendet, sobald der Interpreter auf eine `return`-Anweisung stößt. Daher haben die Zeilen 4 und 7 im folgenden Beispiel keinerlei Auswirkung:

Beispiel: Funktion mit mehreren `return`-Anweisungen

```
1  def max (a, b) :
2      if a > b :
3          return a
4          print("this line will never be executed")
5      else :
6          return b
7          print("and neither will this one")
8
9  print(max(1, 2))
10 print(max(2, 1))
```

Ausgabe: Funktion mit mehreren `return`-Anweisungen

```
2
2
```

Funktionen können auch mehrere Rückgabewerte haben. Diese werden dann durch Kommata getrennt bei `return` aufgeführt und vom Python-Interpreter automatisch zu einem `tuple` „gepackt“. An der aufrufenden Stelle kann dieser `tuple` auch wieder automatisch entpackt werden:

Beispiel: Rückgabe eines `tuple`s

```
1  def returnATuple () :
2      return 1, 2, 3
3
4  fullTuple       = returnATuple()
5  one, two, three = returnATuple()
6
7  print(fullTuple)
8  print(one, two, three)
```

Ausgabe: Rückgabe eines `tuple`s

```
(1, 2, 3)
1 2 3
```

In Zeile 2 könnte gleichermaßen auch stehen: `return (1, 2, 3)`. Der Effekt wäre exakt derselbe.

Wichtig ist, immer im Kopf zu behalten, dass die Variablen, die in einer Funktion benuzt werden, vom restlichen Programm unabhängig sind. Es darf also sich widersprechende Definitionen geben; die Definitionen gelten nur jeweils innerhalb einer Funktion:

Beispiel: Gleiches Symbol, unterschiedliche Inhalte

```
def foobar() :
    a = 7
    print("in foobar: a =", a)

a = 999
print("on module level: a =", a)
func()
print("on module level: a =", a)
```

Ausgabe: Gleiches Symbol, unterschiedliche Inhalte

```
on module level: a = 999
in foobar: a = 7
on module level: a = 999
```

Das `a` in `foobar` erhält also seine eigene Speicherstelle. Je nach Kontext sucht der Python-Interpreter aus, ob die `foobar`-Speicherstelle oder die Modul-Level-Speicherstelle mit dem Symbol `a` gemeint ist. In der Vorstellung von Vor- und Nachnamen: In diesem Beispiel gibt es zwei Menschen (Speicherstellen) mit dem Vornamen `a`. Diese sind jedoch zwei unterschiedliche Menschen; sie haben die Nachnamen `foobar` und `module level`.

In Python dürfen Funktionen auch ineinander verschachtelt werden. Dies kann zum Beispiel nützlich werden, wenn eine Funktion aus mehreren, sich wiederholenden Teilaufgaben besteht, die sonst nirgendwo gebraucht werden:

Beispiel: Verschachtelte Funktionen

```
def veryComplexTask(listOfObjects) :
    def subTask(element) :
        # ...
        pass

    for element in listOfObjects :
        subTask(element)

    # ...
```

Ein weiterer Grund, Funktionen ineinander zu verschachteln wird in Abschnitt 6.6 gezeigt.

6.2. By-Value- und By-Reference-Übergabe

Wir hatten festgestellt, dass die Variablen, die wir in einer Funktion verwenden, vom restlichen Code abgetrennt sind. Wie aber verhält es sich mit den Parametern?

Sehen wir uns dazu folgenden Code an:

Beispiel: Übergabe von Referenz by Value

```
1  def foobar(data) :
2      data.append(1)
3      data = data + [2]
4      print("in foobar: data =", data)
5  data = []
6  foobar(data)
7  print("on module level: data =", data)
```

Oben wurde bereits gesagt, dass bei einem Funktionsaufruf eine *Kopie* des Originalobjekts (hier also `data`) an die Funktion übergeben wird. Dann würden wir also erwarten, dass in Zeile 8 die Liste `data` wieder unverändert als leere Liste vorliegt. Kurioserweise lautet die Ausgabe aber:

Ausgabe: Übergabe von Referenz by Value

```
in foobar: data = [1, 2]
on module level: data = [1]
```

Wie lässt sich dies erklären?

In Abschnitt 4.1 hatten wir uns klar gemacht, dass Variablen in Python in Wirklichkeit primär die *Adressen* der eigentlichen Information speichert. Diese Adresse wird auch als Kopie übergeben. In Zeile 2 erteilen wir nun den Befehl, an die Liste an der Stelle `data` die Zahl 1 anzuhängen. Die Liste wird dadurch im Speicher nicht verschoben; daher sehen wir später in Zeile 8 das Ergebnis dieser Änderung.

In Zeile 3 dagegen wird eine *neue* Liste berechnet (nämlich `[1, 2]`). Diese neue Liste wird an einer eigenen Speicherstelle angelegt, überschreibt also nicht die alten Daten an der Adresse `data`. Wir speichern in Zeile 3 die Adresse in der *lokalen* Variablen `data`. Das bedeutet, die Funtion `foobar` weiß nun nicht mehr, wo die Original-Liste sich befindet, sondern kann nur noch die Liste `[1 ,2]` ausgeben – und dies geschieht auch in Zeile 4. Die ursprüngliche Liste besteht aber weiterhin. Wo diese sich befindet, ist immer noch in der Variablen `data` des Modul-Levels gespeichert. Wir sehen daher in Zeile 8 das Ergebnis von `append` (da dies die Adresse von `data` nicht verändert hat), nicht aber das Ergebnis von `data = data + [2]` (da hierfür eine neue Liste angelegt wurde).

Speicherbild

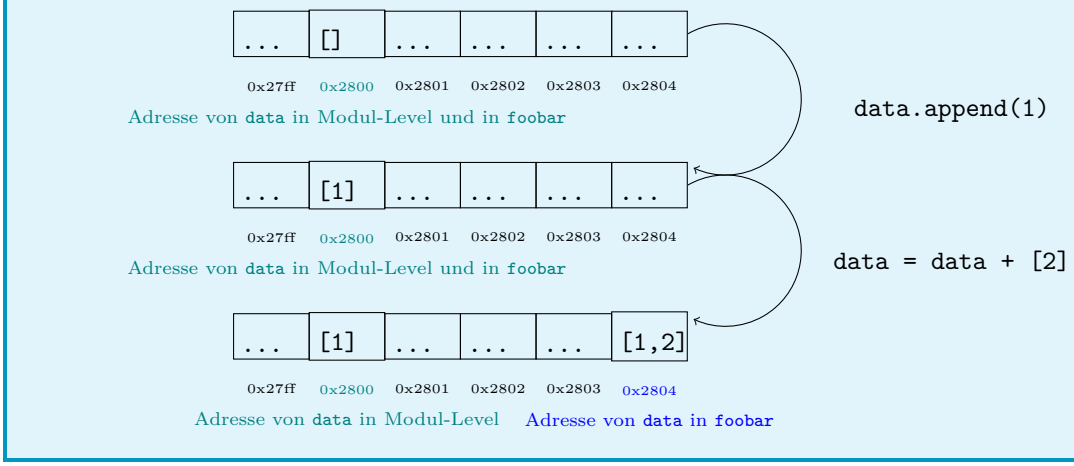


Abbildung 6.1.: Entwicklung einer `list` als Funktionsparameter

Durchdenken wir nun auch dieses Beispiel:

Beispiel: Übergabe von Immutables

```
1  def foobar(data) :
2      data = 2
3      print("in foobar: data =", data)
4
5  data = 1
6  foobar(data)
7  print("on module level: data =", data)
```

Ausgabe: Übergabe von Immutables

```
in foobar: data = 2
on module level: data = 1
```

Wir erinnern uns, dass `1` ein Wert vom Typ `int` und damit *immutable* ist. Der Wert an der Stelle `data` darf also nicht verändert werden. Wenn nun also Zeile 2 verlangt, den neuen Wert `2` „in `data`“ zu speichern, so wird eine neue Speicherstelle angelegt, und deren Adresse in der Instanz `data` in `foobar` gespeichert.

Speicherstruktur von `lists`

Die oben gezeigten Speicherbilder sind eine Vereinfachung der Realität. Wie erwähnt kann jede Zelle des Speichers nur ein einziges Byte fassen, also nicht einmal einen ganzen `int`-Wert. Stattdessen finden Sie dort einen *descriptor*, also ein Datenpaket, das unter anderem aus einer Ganzzahl (Anzahl der Elemente in der Liste) und einem Pointer (tatsächlicher Speicherort des ersten Elements der Liste, also eine Adresse) enthält. Dadurch bleibt die Adresse `data` konstant, selbst wenn der Listeninhalt „umziehen“ muss.

In systemnahen Sprachen wie C müssen Sie das mit im Auge behalten. Bei der Arbeit in Python dagegen können Sie sich weit mehr auf „das Große und Ganze“ konzentrieren.

6.3. Scopes, `global`, `nonlocal`

Bisher habe ich Ihnen verschwiegen, dass es möglich ist, innerhalb von Funktionen auf Variablen des Modullevels zuzugreifen. Dabei dürfen aber nur solche Operationen durchgeführt werden, die den Speicherort der Variablen nicht verändern. Auf Modulebene dagegen sind die Symbole in den Funktionen nie zugänglich. Sehen Sie sich hierzu das folgende Beispiel an:

Beispiel: lokale Variablen

```
1  foo = ["foo"]
2
3  def foobar() :
4      cat = "confused"       # lokale Variable
5      print(foo)             # liest die Variable auf Modulebene
6      foo.append("bar")      # schreibender Zugriff, keine Änderung der Adresse
7      #foo = foo + ["baz"]  ! Fehler: Neue Adresse
8
9  foobar()
10
11 print(foo)
12 # print(cat)             ! Fehler: 'cat' ist Variable von foobar.
```

Ausgabe: lokale Variablen

```
['foo']
['foo', 'bar']
```

Man spricht hierbei von *Scopes*: Jedes Symbol hat einen zugeordneten Bereich, in dem dieses Symbol gültig ist (in dem es „existiert“). Dieser Gültigkeitsbereich (Scope) reicht von dem Punkt, an dem das Symbol *definiert* wird (d. h. bei der ersten Wertzuweisung) bis zum *Ende der aktuellen Einrückungsebene*. Die Variable `cat` des obigen Beispiels gehört also zum Scope der Funktion `foobar`, und darf damit nur in den Zeilen 4-7 benutzt werden. Ein Zugriff wie in Zeile 11 dagegen führt zu einem Fehler.

Die Variable `foo` dagegen existiert in allen Zeilen des Programms (1-11), da sie auf Modulebene definiert wurde. Der Scope der Funktion `foobar` ist vom Scope der Modulebene umschlossen, *gehört also dazu*. Sie können auch sagen, „es ist nicht möglich, in Funktionen hineinzusehen, wohl aber aus ihnen heraus zu schauen“.

Zugriffe klar halten

Sie können sich vermutlich jetzt schon vorstellen, dass es schwierig wird, den Überblick zu behalten, welche Variablen an welcher Stelle sichtbar sind, und wann ein Schreibzugriff zulässig ist. Das hier Gezeigte soll Sie daher auch nur dazu in die Lage versetzen, Codes von anderen Autoren zu lesen und korrekt zu interpretieren. Im Allgemeinen ist davon abzuraten, Konstruktionen zu programmieren, bei denen Variablen über Scope-Grenzen hinweg genutzt werden. Halten Sie sich am besten an den Leitsatz, dass Information nur über die Parameterliste in eine Funktion gelangt, und nur über den Rückgabewert aus dieser heraus kommt.

Konstanten

Ein Fall, in dem es sinnvoll sein kann, von diesem Leitsatz abzuweichen, sind Naturkonstanten. Wenn Sie den Flug eines Balles durch die Luft modellieren, wollen Sie nicht in jeder Funktion die Graviationskonstante als Parameter mit übergeben. Es ist aber auch im Allgemeinen keine gute Idee, den Wert in Ihren Funktionen „hard zu coden“. (Andernfalls müssten Sie bei einer Änderung Ihr gesamtes Programm durcharbeiten, und würden sehr wahrscheinlich eine oder mehrere Stellen übersehen).

In diesem Fall ist es also durchaus angebracht, Code zu schreiben wie:

Beispiel: globale Konstante

```
1  GRAVITATIONALCONSTANT = 6.674E-11
2
3  def forceBetweenBodies(radius, mass1, mass2) :
4    return GRAVITATIONALCONSTANT * mass1 * mass2 / (radius**2)
```

Konstanten werden häufig in GROSSBUCHSTABEN gesetzt, um die ProgrammiererInnen daran zu erinnern, dass diese nicht geändert werden sollten.

Wenn versucht wird, innerhalb einer Funktion ein Symbol des Modul-Levels umzudefinieren, so legt Python stattdessen eine neue *lokale* Variable an. Das heißt, der Bezug auf die ursprüngliche Variable geht verloren, und in der Funktion ist nur der neue Wert verfügbar. Auf die Modulebene hat dies keinen Einfluss.

Soll tatsächlich von der Funktion aus der Wert der Variablen auf Modulebene geändert werden, so kann dies mit dem Befehl `global` geschehen. Mit `global Symbol` wird dem Python-Interpreter mitgeteilt, dass mit `Symbol` tatsächlich immer das Objekt der Modulebene gemeint ist.

Beispiel: lokale und globale Variablen

```
1  foo = "foo"
2  bar = "bar"
3
4  def foobar() :
5      foo = "fu"          # lokale Variable
6
7      global bar          # Variable 'bar' aus dem Modulelevel
8      bar = "baz"
9
10     print("foobar: foo =", foo)
11     print("foobar: bar =", bar)
12
13 foobar()
14
15 print("on module level: foo =", foo)
16 print("on module level: bar =", bar)
```

Ausgabe: lokale und globale Variablen

```
foobar: foo = fu
foobar: bar = baz
on module level: foo = foo
on module level: bar = baz
```

Mit *Modulebene* ist dabei immer die *unterste Ebene* in Ihrem Code gemeint. Wenn Sie den Effekt von `global` nur auf die nächsthöhere Ebene anwenden wollen (z. B. bei verschachtelten Funktionen), so erreichen Sie dies durch das Schlüsselwort `nonlocal`. Die Syntax und Verwendung ist dabei exakt dieselbe. Jedoch kann `nonlocal` nicht dazu verwendet werden, um auf Objekte der Modulebene zuzugreifen.

Grund für diese Unterscheidung sind technische Details im Unterbau von Python, auf die wir hier leider nicht eingehen können.

6.4. Optionale Parameter, Positionale Parameter und Keyword-Parameter

Oft schreiben wir Funktionen, die fast jedes Mal mit denselben Parametern aufgerufen werden. Es kann lästig werden, diese immer wieder und wieder einzugeben. Denken Sie dazu an das Beispiel *Funktion „ohne“ Rückgabewert* zurück: die Funktion `printBoxed` braucht einen Wert `boxSize` für die Breite der Box. Meistens werden wir aber immer dieselbe Breite erzeugen wollen, um dem User unseres Programms ein einheitliches Interface zu bieten. Das bedeutet leider zunächst, dass wir bei jedem Aufruf unserer Funktion `printBoxed` diesen Standard-Wert immer wieder mit übergeben müssen.

In Python ist es jedoch möglich, solche Standardwerte zu definieren. Beim Aufruf dürfen diese dann einfach weggelassen werden, können jedoch auch explizit durch einen anderen Wert ersetzt werden. Sie erreichen solche *optionalen Parameter*, indem sie in der ersten Zeile der Funktion (in der Funktionssignatur) hinter einen Parameter ein `= defaultWert` setzen.

Syntax: Funktionen mit optinalem Parameter

```
def Funktionsname (Parameter1, ..., OptionalerParameter = Defaultwert) :
    normaler Code
```

Im Beispiel unserer Box kann das so aussehen:

Beispiel: Funktion mit optionalem Parameter

```
1  import math
2
3  def printBoxed(text, boxSize = 60) :
4      # Draws text in a box like:
5      #
6      # +------------+
7      # |   text     |
8      # +------------+
9
10     lenText     = len(text)
11     countSpaces = boxSize - lenText - 2        # 2 spaces for |borders|
12     spacesLeft  = math.floor(countSpaces / 2)  # round down
13     spacesRight = math.ceil (countSpaces / 2)  # round up
14
15     print("+" +                (boxSize - 2) * "-"           + "+")
16     print("|" + spacesLeft * " " + text + spacesRight * " " + "|")
17     print("+" +                (boxSize - 2) * "-"           + "+")
18
19 printBoxed("Don't forget to be awesome!")
20 printBoxed("This is a small box", 25)
```

Ausgabe: Funktion mit optionalem Parameter

```
+------------------------------------------------------------+
|               Don't forget to be awesome!                  |
+------------------------------------------------------------+
+----------------------+
|  This is a small box  |
+----------------------+
```

Wird also nur ein Parameter angegeben, so setzt der Interpreter für den Zweiten automatisch der default-Wert `60` ein.

Funktionen dürfen beliebig viele optionale Parameter haben. Diese Parameter mit Vorgabewert müssen aber *hinter* den normalen Parametern stehen. Im Aufruf dürfen Parameter auch nur „von hinten weg“ weggelassen werden:

Beispiel: Funktion mit mehreren optionalen Parametern (1)

```
1  def defaults(a, b, c=0, d=0) :
2    print(a, b, c, d)
3
4  #def defaults(a, b=0, c)    ! Fehler: c müsste auch einen default-Wert haben
5
6  defaults(1, 2)              # ok, Ausgabe: 1 2 0 0
7  defaults(1, 2, 3, 4)        # ok, Ausgabe: 1 2 3 4
8  defaults(1, 2, 3)           # ok, Ausgabe: 1 2 3 0
9  # defaults(1, 2,  , 4)      ! Fehler: Unzulässige Syntax
```

Um das Verhalten zu erreichen, das in der (fehlerhaften) Zeile 9 des obigen Beispiels angedeutet wird, kann *im Funktionsaufruf* durch ein Gleichheitszeichen erklärt werden, welcher Parameter denn gemeint ist. Die Reihenfolge muss dann nicht eingehalten werden! Allerdings müssen diejenigen Parameter, die rein über ihre Position in der Liste identifiziert werden sollen, zuerst genannt werden. Man nennt solche Parameter, die über ein Schlüsselwort zugewiesen werden *Keyword-Parameter*, da sie eben über einen Schlüssel identifiziert werden. Im Gegensatz dazu nennt man Parameter, die nur über ihre Position in der Liste zuordbar sind *Positionale Parameter*.

Beispiel: Funktion mit mehreren optionalen Parametern (2)

```
1  def defaults(a, b, c=0, d=0) :
2    print(a, b, c, d)
3
4  defaults(1, 2, d = 4)       # ok, Ausgabe 1 2 0 4
5  defaults(a=9, b=8)          # ok, Ausgabe 9 8 0 0
6  defaults(9, c=7, b=8)       # ok, Ausgabe 9 8 7 0
7  # defaults(1, 2, c=1, c=2)  ! Fehler: Doppelte Zuweisung von c
8  # defaults(9, a=7, b=0)     ! Fehler: Doppelte Zuweisung von a
9  # defaults(a=9, 3)          ! Fehler: Positionaler Parameter nach Keyword-P.
```

Die Funktion `print`[3] erlaubt es, das „Trennzeichen“ zwischen den einzelnen Informationsblöcken frei zu wählen. Dazu dient der *Keyword-Parameter* `sep`. Standardmäßig ist dieser als Leerzeichen (`" "`) festgelegt.

[3] Tatsächlich ist `print` genau eine solche Funktion, wie Sie sie in diesem Kapitel kennen lernen. Die meisten Befehle in Python sind tatsächlich solche Funktionen und aus „kleineren“ Befehlen zusammengesetzt

Beispiel: Optionaler Paramter `sep` in der Funktion `print` (1)

```
print(1, 2)
print(1, 2, sep=",")        # Ausgabe: 1,2
print(1, 2, sep="---")      # Ausgabe: 1---2
print(1, 2, sep="")         # Ausgabe: 12
```

Daneben gibt es auch den Keyword-Parameter `end`. Dieser enthält Zeichen, die am Ende jedes Aufrufs von `print` gedruckt werden sollen. Üblicherweise ist das ein Zeilenumbruch.

Beispiel: Optionaler Paramter `sep` in der Funktion `print` (2)

```
print("The Planet Arrakis")
print("The Planet Arrakis", end="")
print("The Planet Arrakis", end="---\n")
print("The Planet Arrakis")
```

Ausgabe: Optionaler Paramter `sep` in der Funktion `print` (2)

```
The Planet Arrakis
The Planet ArrakisThe Planet Arrakis---
The Planet Arrakis
```

6.5. Variadische Funktionen

Sie können Funktionen auch so schreiben, dass eine beliebige Zahl von Werten angenommen wird. Denken Sie hierzu zum Beispiel an den Befehl `print`: Sie können hier eine beliebig lange Liste von Werten und Ausdrücken angeben, die von derselben Funktion verarbeitet werden. Man sagt also, der Befehl `print` ist eine *variadische Funktion*, da die Zahl der Parameter nicht festgelegt ist.

Um dieses Verhalten zu erreichen müssen Sie nicht etwa eine endlos lange Signatur tippen, sondern können ein neues Syntax-Konstrukt nutzen:

Syntax: Variadische Funktionen (1)

```
def Funktionsname (Parameterliste, ...,
                   *variadischerParameter,
                   optionaleParameter = default, ...) :
    normaler Code
```

Zwischen normalen und den optionalen Parametern oder *hinter* den optionalen Parametern können Sie also durch einen Stern (∗) vor dem Parameternamen andeuten, dass hier eine beliebig lange Kette von Werten akzeptiert werden soll. Diese Werte werden automatisch zu einem `tuple` zusammengepackt, die im Code den Namen trägt, den Sie für `variadischerParameter` angegeben haben.

Beispiel: Durchschnitt vieler Werte mit variadischem Parameter

```
def average(*args) :
  total = sum(args)
  return total / len(args)

print(average(1, 2, 3, 4, 5))    # Ausgabe: 3.0
```

Die *fünf einzelnen Werte* `1`, `2`, `3`, `4`, `5` werden also zu *einem einziglen* `tuple` `args = (1, 2, 3, 4, 5)` zusammengefasst.

Aus naheliegenden Gründen darf es nur einen variadischen Parameter pro Funktion geben – andernfalls könnte der Python-Interpreter nicht zuordnen, welcher Wert zu welcher Liste gehören sollte.

Analog zu den variadischen Parameter gibt es auch variadische *Keyword*-Parameter, also eine beliebig lange Folge von Schlüsseln und zugehörigen Werten. Diese werden mit einer ähnlichen Syntax erklärt:

Syntax: Variadische Funktionen (2)

```
def Funktionsname (Parameterliste, ...
                   *variadischerParameter,
                   optionaleParameter = default, ...
                   **variadischerKeywordParameter) :
    normaler Code
```

Python packt diese variadischen Keyword-Argumente automatisch zu einem `dict` zusammen.

Beispiel: Variadische Keyword-Parameter

```
def presentMe(**kwargs) :
  print("Here are a few details about me:")
  for (key, value) in kwargs.items() :
      print(key, ": ", value)

presentMe(
    nickname="The Blue Chameleon",
    talents="drinking coffee",
    cellphoneNumber="ask nicely"
)
```

Ausgabe: Variadische Keyword-Parameter

```
Here are a few details about me:
nickname :  The Blue Chameleon
talents :  drinking coffee
cellphoneNumber :  ask nicely
```

Wie angedeutet können Sie positionale Parameter, Keyowrd-Parameter, variadische Parameter und variadische Keyword-Parameter in beliebiger Kombination verwenden. Jedoch muss die Reihenfolge dieser Gruppen sowohl bei der Deklaration als auch beim Aufruf eingehalten werden.

Beispiel: Verschiedene Arten von Parametern im Zusammenspiel

```
def parameterShowcase(a, b, c="default-Param", *args, **kwargs) :
    print("a      :", a)
    print("b      :", b)
    print("c      :", c)
    print("args   :", args)
    print("kwargs:", kwargs)
    print()

parameterShowcase(1, 2)
parameterShowcase(1, 2, "explicit Param", 9, 8, 7)
parameterShowcase(1, 2, "explicit Param", 9, 8, 7, key1="value", key2="VALUE")
parameterShowcase(9, 8, 7, 6, 5, key="value")
parameterShowcase(9, 8, 7, key="value")
```

Ausgabe: Verschiedene Arten von Parametern im Zusammenspiel

```
a      : 1
b      : 2
c      : default-Param
args   : ()
kwargs: {}

a      : 1
b      : 2
c      : explicit Param
args   : (9, 8, 7)
kwargs: {}

a      : 1
b      : 2
c      : explicit Param
args   : (9, 8, 7)
kwargs: {'key1': 'value', 'key2': 'VALUE'}

a      : 9
b      : 8
c      : 7
args   : (6, 5)
kwargs: {'key': 'value'}

a      : 9
b      : 8
c      : 7
args   : ()
kwargs: {'key': 'value'}
```

Dieses automatische Verpacken kann auch Rückgängig gemacht werden: Wenn Sie im Funktionsaufruf vor einen Container (`lists`, `tuples`, ...) ein Sternchen (`*`) setzen, so werden die Elemente des Containers nacheinander den Parametern der Funktion zugeordnet

Beispiel: Entpacken von Listen in Funktionsaufrufen

```
1  import math
2
3  def vectorLength(x, y = 0, z = 0) :
4      return math.sqrt(x*x + y*y + z*z)
5
6  vector2D = [3, 4]
7
8  print(vectorLength(*vector2D))          # x = 3, y = 4, z = 0
9  #print(vectorLength(vector2D))          ! kein Entpacken sondern x = [3, 4]
```

`dict`s können auf zweierlei Weise entpackt werden: Mit einem einfachen Sternchen `*`, ebenso wie auch normale Listen entpackt werden, oder mit einem doppelten Sternchen `**`, um so den Keyword-Parametern zugewiesen zu werden:

Beispiel: Entpacken von `dict`s in Funktionsaufrufen

```
1  params = {"sep" : ", ", "end" : "#\n"}
2
3  print("The Planet Caladan", "Home of House Atreides",  params)
4  print("The Planet Caladan", "Home of House Atreides",  *params)
5  print("The Planet Caladan", "Home of House Atreides", **params)
```

Ausgabe: Entpacken von `dict`s in Funktionsaufrufen

```
The Planet Caladan Home of House Atreides {'sep': ', ', 'end': '#\n'}
The Planet Caladan Home of House Atreides sep end
The Planet Caladan, Home of House Atreides#
```

In Zeile 3 wird also `params` einfach als dritter Parameter gedruckt: Wir erhalten die Darstellung des gesamten `dict`s.

Zeile 4 wird umgesetzt zu

```
print("The Planet Caladan", "Home of House Atreides", "sep", "end")
```

und entsprechend sieht auch die Ausgabe auf dem Bildschrim aus.

Zeile 5 schließlich interpretiert Python als

```
print("The Planet Caladan", "Home of House Atreides", sep=', ', end='#\n')
```

weswegen das Komma zwischen `The Planet Caladan` und `Home of House Atreides` eingeschoben wird.

6.6. Funktionen und Module als Parameter und Rückgabewerte

Python folgt der Philosophie: *Alles ist ein Objekt*. Ein Objekt ist eine Sammlung von Daten, die ein beliebig abstraktes Konstrukt beschreibt. Die Zahl `5` ist ebenso ein Objekt wie eine `list`, ein `dict`, eine Funktion wie `printBoxed` oder `print` oder sogar ein ganzes Modul wie `math`. Jedes Objekt kann auch als Parameter an eine Funktion übergeben werden.

Im Folgenden Beispiel werden wir diese Möglichkeit nutzen, um mit Python die Ableitung von beliebigen Funktionen anzunähern[4].

Falls Ihr mathematisches Wissen aus der Schule schon etwas verblichen ist: Die Ableitung einer Funktion f als:

$$\frac{\mathrm{d}}{\mathrm{d}x} f(x) \coloneqq \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

Dieses *mathematische Rezept* gibt uns eine Anleitung, wie wir eine neue Funktion f' erhalten. Wir wollen nun in ähnlicher Manier neue Funktionen f' aus bekannten Funktionen f bauen.

Während den Grenzübergang $\varepsilon \to 0$ nur in der Mathematik möglich ist, können wir eine sinnvolle Abschätzung (Annäherung) dieser Ableitung erhalten, indem wir den Limes einfach ignorieren und für ε einen beliebigen, sehr kleinen Wert einsetzen. Mit dieser Idee schreiben wir den folgenden Code:

Beispiel: numerisches Ableiten

```python
import math

def derivative (f, epsilon = 1e-7) :
    def result(x) :
        return (f(x + epsilon) - f(x)) / epsilon
    return result

dsin = derivative(math.sin)

for x in range(20) :
    print(int(20 + 10 * dsin(x / 3)) * " ", "*")
```

Ausgabe: numerisches Ableiten

```
                          *
                          *
                        *
                      *
                    *
                  *
                *
              *
            *
          *
          *
           *
             *
                *
                   *
                      *
                         *
                           *
                           *
                           *
```

[4] Der gezeigte Code hat zahlreiche Schwächen, und soll nur das Konzept von Funktionen als Parameter und Rückgabewert illustrieren. Besuchen Sie Vorlesungen wie *Numerik* oder *Numerical Recipies*, um über optimierte Methoden zu lernen.

Sehen wir uns im Detail an, was hier geschieht:

Die Funktion `derivative` erhält zwei Parameter, wovon der Parameter `epsilon` optional ist und per default auf `1e-7` gesetzt wird. Für das Objekt `f` nehmen wir hier an, dass es sich wie eine Funktion verhält, die genau einen Parameter verarbeitet.

In der Funktion `derivative` wird nun eine neue Funktion `result` definiert. Diese Funktion verarbeitet auch einen einzigen Parameter. Sie berechnet einen einzigen Wert nach der Definition des Differentialquotienten (also die Ableitung, die wir oben gesehen haben). Zur Berechnung dieses Wertes benutzt die Funktion `result` das Symbol `f`, also eine beliebige Funktion, die zuerst an `derivative` übergeben wurde. Da `result` im Scope von `derivative` steht, und da das Symbol `f` nicht überschrieben wird, darf `result` dieses Symbol quasi mitnutzen: `result` „sieht“, welche Funktion an `derivative` übergeben wurde, und kann somit den Differentialquotienten berechnen.

Von `derivative` wird nun aber nicht nur ein einzelner Wert $f'(x)$ zurückgegeben, sondern *die gesamte Funktion* f': Die „Vorschrift“, wie für eine jetzt fest gegebene Funktion `f` die Ableitung angenähert wird, ist das Ergebnis/der Rückgabewert von `derivative`. Wir haben also eine *Funktion, die eine Funktion berechnet*!

In Zeile 8 kommt die Funktion `derivative` auch zum Einsatz: Gegeben die Funktion `math.sin` berechnen wir nun die Vorschrift, nach der die Ableitung des Sinus berechnet werden kann, und speichern diese *Vorschrift* im Symbol `dsin`.

Die Zeilen 10 und 11 dienen dazu, einen rudimentären Plot unserer Funktion zu erstellen. In Kapitel 10 werden wir Mittel kennen lernen, ansprechendere Grafiken zu erzeugen. Für den Moment soll Sie nur interessieren, dass das soeben erstellte Objekt `dsin` genauso aufgerufen werden kann, als hätten Sie es mit den folgenden Zeilen explizit programmiert:

Beispiel: expliziter Differentialquotient

```
1  import math
2
3  def dsin (x) :
4      return (sin(x + epsilon) - sin(x)) / epsilon
```

6.7. Lambdas

Lambdas sind „Funktionen in anderem Gewand“. Zunächst können Sie sich Lambdas als Kurzform für besonders einfache Funktionen vorstellen, denn sie können und müssen mit einer einzigen Zeile geschrieben werden. Sie werden erstellt mit der Syntax:

Syntax: Lambdas

```
Symbol = lambda Parameterliste : Rückgabewert
```

Dies ist äquivalent zu folgendem Code-Template:

Syntax: Äquivalent von Lambdas

```
def Symbol (Parameterliste) :
    return Rückgabewert
```

Für die Parameterliste gilt dasselbe wie oben gesagt: Positionale, Optioale, Keyword- und variadische Parameter sind allesamt erlaubt und verhalten sich ebenso, wie Sie das jetzt von normalen Funktionen kennen. Auch aufgerufen werden Lambdas auf dieselbe Art wie normale Funktionen.

Beispiel: Durchschnitt vieler Werte Lambda-Funktion

```
1  average = lambda *args : return sum(args) / len(args)
2  print(average(1, 2, 3, 4, 5))     # Ausgabe: 3.0
```

Lamdas werden vor allem dann genutzt, wenn eine Funktion nur genau an einer Stelle gebraucht wird. Ein Beispiel hierfür gibt die Funktion `sorted`:

`sorted` gibt eine sortierte Kopie eines Containers (`list`, `tuple`, ...) zurück. Wenn ein Container die Elemente e_i hat, dann werden diese so angeordnet, dass gilt:

$$e_1 \leq e_2 \leq \dots$$

Es ist dabei aber auch möglich, das Sortierkriterium selbst zu spezifizieren. Wenn eine Funktion f aus den Werten e_i vergleichbare Werte berechnet (z. B. Zahlen), so kann `sorted` dazu verwendet werden, die Elemente so anzuordnen, so dass gilt:

$$f(e_1) \leq f(e_2) \leq \dots$$

Wir erreichen dies, indem wir `sorted` den optionalen Keyword-Parameter `key` mitgeben. Hier nennen wir genau die Funktion f, die unser Sortierkriterium beschreibt.

Beispiel: Sortieren mit Lambda

```
1  data = [5, 1, 4, 7]
2
3  print( sorted(data) )                          # default: Aufsteigende Sortierung
4  print( sorted(data, key = lambda x : -x) )     # Absteigende Sortierung
```

Ausgabe: Sortieren mit Lambda

```
[1, 4, 5, 7]
[7, 5, 4, 1]
```

Es gilt also in diesem Beispiel: $f(x) = -x$. Wenn die Werte $-x$ aufsteigend sortiert werden, ist dies gleichwertig zum absteigenden Sortieren der Werte x.

Beachten Sie, dass die Lambda-Funktion *direkt an dem Ort* definiert wird, wo sie auch gebraucht wird, und den restlichen Code nicht „verstopft“. Natürlich müssen Sie diese Aufgabe nicht über ein vor Ort definiertes Lambda lösen; folgende beiden Versionen sind auch möglich (aber etwas unhandlicher):

Beispiel: Sortieren – Alternativen

```
1  def funcSorter(x) :
2      return -x
3  lambdaSorter = lambda x : -x
4
5  data = [5, 1, 4, 7]
6  print( sorted(data, key = funcSorter), sorted(data, key = lambdaSorter) )
```

7. Klassen

> What's the object-oriented way to wealth? – Inheritance
>
> Anonymous

In ihrem Grundaufbau sind Computer Maschinen, die mit *einzelnen Zahlen* umgehen. *Gruppen* von Zahlen sind alles, was wir brauchen, um unsere Welt zu beschreiben[1]. Dazu müssen diese Zahlen aber eine gewisse Ordnung und Beziehung zueinander haben und wir brauchen eine geeignete Methode zur Interpretation dieser Zahlen. Die Zahlen 98, 108, 117, 101 können zum Beispiel *in dieser Reiheinfolge* als ASCII-Codes der Buchstaben des Wortes *blue* interpretiert werden; es könnten aber auch die Hausnummern der Adressen meiner ersten vier Lebensgefährtinnen sein[2].

Für Texte haben wir den Datentyp `string` kennengelernt. Hinter dem Datentyp steht eine aufwändige Maschinerie, die eben genau diese Ordnung und Interpretation von Zahlengruppen bietet. In diesem Kapitel wollen wir uns damit beschäftigen, eigene solche Konstrukte zu erschaffen: Wir erstellen hier *Klassen*.

Wir wollen uns als Beispiel vorstellen, dass ich in eine Diskussion verwickelt wurde, wer weniger Glück in den vergangenen Beziehungen hatte. Wir werden Python benutzen, um diese Frage mit wissenschaftlicher Methodik zu beantworten. Weiter unten sind die Daten meiner und Charlottes früheren Beziehungen tabellarisch aufgelistet. Diese Tabelle werden wir in Code übersetzen, und damit unsere Streitfrage eindeutig beantworten.

7.1. Klassen als Container

Zunächst sollten Sie sich eine Klasse als Datencontainer vorstellen, wie wir sie schon in Kapitel 4 kennengelernt haben[3]. Dort wurden mehrere z. B. Werte zu einem kompakten `dict` zusammengefasst. Um einen dieser zusammen gepackten Werte zu greifen, brauchten wir einen *Schlüssel*.

Klassen haben in dieser Hinsicht eine ähnliche Struktur: Sie sind eine Sammlung von Attributen, zu denen man Werte speichern kann. Die Attribute einer Klasse richten sich natürlich danach, was modelliert werden soll. In unserem Beispiel sind das die Eigenschaften der früheren PartnerInnen. Attribute können dann so etwas sein wie *Name*, *Körpergröße*, *positive Eigenschaften* oder *negative Eigenschaften*.

Wir können uns in dem Kontext eine Tabelle vorstellen, in der wir unsere Verflossenen auflisten könnte. Die Spaltenüberschriften dieser Tabelle wären dann die Attribute. Jede Zeile dieser Tabelle nennen wir eine *Instanz* der Klasse. Eine Instanz beschreibt in diesem Fall also einen konkreten Menschen, der (unter anderem) durch die Attribute *name*, *height*, *upsides* und *downsides* beschrieben werden kann. Eine Instanz braucht außerdem noch ein *Symbol*, über den klar wird, welche Zeile der Tabelle gemeint ist

[1] Ich bin gerne zu einer philosophischen Diskussion über diese Aussage bereit. Für diesen Kurs wollen wir aber zumindest annähern, dass Zahlen zur Beschreibung von Dingen genügen, die wir mit einem Computer bearbeiten wollen

[2] Sind sie nicht. Ich kann mich nur an die Hausnummer meiner vierten Freundin erinnern (es war die 12), und meine erste Freundin hat so weit von der nächsten Stadt gelebt, dass ich mir nicht sicher bin, ob das Haus überhaupt eine Hausnummer hatte.

[3] Tatsächlich sind die dort gezeigten Datentypen Beispiele für Klassen

(schließlich ist es zumindest denkbar, dass zwei gleich große Menschen mit demselben Namen existieren, an denen wir dieselben Vorzüge und Nachteile sehen).

Eine Liste von Liebschaften

Stefans romantische Vergangenheit

Symbol	Name	Height	Upsides	Downsides
`exgf1`	Steffie	1.65 m	intelligent, beautiful, has a dog	too attached to her mother, doesn't like meeting people
`exgf2`	Arista	1.60 m	very intelligent, beautiful, good taste in music	doesn't like coffee, doesn't like coding
`exgf3`	Katja	1.81 m	intelligent, very beautiful, musician	moody, cheated on me

Charlottes gebrochene Herzen

Symbol	Name	Height	Upsides	Downsides
`exbf1`	Sebastian	1.78 m	intelligent, handsome, likes to listen	obsessed with catching bugs, always late
`exbf2`	Reginald	1.84 m	very intelligent, handsome, plays in a band	flirty with everyone, complains a lot
`exbf3`	Sönke	1.81 m	intelligent, quite handsome, cheerful nature	superficial, racist

Ich sollte anmerken, dass diese Tabelle keine echten Menschen beschreibt. Weder die Namen noch die Eigenschaften stimmen mit tatsächlichen ExpartnerInnen überein. Die genannten Eigenschaften sollen anschauliche Beispiele geben, sind aber zwingend nicht als Wertung von Eigenschaften im echten Leben zu verstehen[a].

[a] Still, racism sucks.

Dies wollen wir nun in Code abbilden. Wir beschreiben also die *Klasse Expartner*. Dazu brauchen wir ein neues Syntaxelement:

Syntax: Klassen

```
class Klassenname  :
    Attribut1 = Standardwert
    Attribut2 = Standardwert
    ...
```

Wie also schon bei Funktionen gibt die Einrückungsebene an, was zur Klasse gehört, und was dem restlichen Code zugeordnet wird. Der *Standardwert* wird beim Erstellen der Instanzen zugewiesen, steht also „in unserer Tabelle“, wenn eine neue Zeile erstellt wird.

Wir legen eine Instanz mit der folgenden Syntax an:

Syntax: Instanzen anlegen

```
Instanz = Klassenname()
```

Beachten Sie die leeren Klammern! Diese sind nötig, um Python mitzuteilen, dass wirklich eine *Instanz* angelegt werden soll. Ohne die Klammern erstellen Sie eine Kopie der Klasse, also eine neue Tabelle.

Sie können nun sowohl lesend als auch schreibend auf die Tabelle zugreifen, indem Sie Klasse und Attribut mit einem Punkt verbinden:

Syntax: Zugriff auf Klassenelemente

```
Instanz.Attribut = Wert    # Schreibender Zugriff
print(Instanz.Attribut)    # Beispiel für lesenenden Zugriff
```

Damit können wir die obige Tabelle so in Code übersetzen:

Beispiel: Klassen als reine Container

```
class Expartner :
    name      = None
    height    = None
    upsides   = {}
    downsides = {}

exgf1.name      = "Steffie"
exgf1.height    = 1.65
exgf1.upsides   = {"intelligent", "beautiful", "has a dog"}
exgf1.downsides = {"too attached to her mother", "doesn't like meeting people"}

...

print(exgf1.name)    # Ausgabe: Steffie
```

Wir wählen hier {sets} für die Attribute `upsides` und `downsides`, da eine Eigenschaft für jede Exfreundin nur einmal vergeben werden darf. Es soll also keine Instanz von `Expartner` geben, die `{"funny", "funny"}` ist.

Klassennamen mit Großbuchstaben

Es ist Konvention, Klasennamen mit einem Großbuchstaben beginnen zu lassen, während die Instanzen – wie alle Variablen – mit einem Kleinbuchstaben beginnen sollten. Die Attribute verhalten sich auf eine gewisse Weise wie Variablen und werden daher auch mit Kleinbuchstaben beschrieben.

Wir haben oben die Instanzen der Klasse `Expartner` als Tabelle dargestellt. Dies soll nicht den Eindruck vermitteln, als würden diese nur als Einheit existieren. Jede Instanz existiert komplett unabhängig von den anderen. Sie können sich vorstellen, dass die oben gezeigte Tabelle mit einer Schere in einzelne Zeilen geschnitten wurde. Durch das gemeinsame Format (d. h. das Vorhandensein gleicher Attribute) gibt es aber einen thematischen Zusammenhang.

Das bedeutet, dass auch die Gedanken zu lokalen und globalen Variablen auf Klassen anwendbar sind. Instanzen einer Klasse können in Funktionen angelegt werden, und existieren dann nur dort. Sie können auch als Parameter an Funktionen übergeben werden, und von diesen verändert werden. Änderungen an den Attributen sind auch an der aufrufenden Stelle zu sehen solange der Speicherort der Instanz selbst sich nicht ändert:

Beispiel: Instanzen als Funktionsparameter

```
# Klasse Expartner und Instanz exgf1 wie oben

def makeNameUppercase(ex) :
    ex.name = ex.name.upper()

def showEx(ex) :
    print(ex.name)
    print("  Height      : " + str(ex.height))
    print("  Upsides     : " + str(ex.upsides)  [1:-1])
    print("  Downsides   : " + str(ex.downsides)[1:-1])

makeNameUppercase(exgf1)
showEx(exgf1)
```

(Das Slicing (`[1:-1]`) soll hierbei nur die `set`-Klammern {} verbergen.)

Ausgabe: Instanzen als Funktionsparameter

```
STEFFIE
  Height      : 1.65
  Upsides     : 'beautiful', 'has a dog', 'intelligent'
  Downsides   : "doesn't like meeting people", 'too attached to her mother'
```

Um zu verstehen, warum diese Änderung durch `makeNameUppercase` auch weitergegeben wird, sollten wir uns nochmal ein Bild des Arbeitsspeichers zeichnen:

Speicherbild

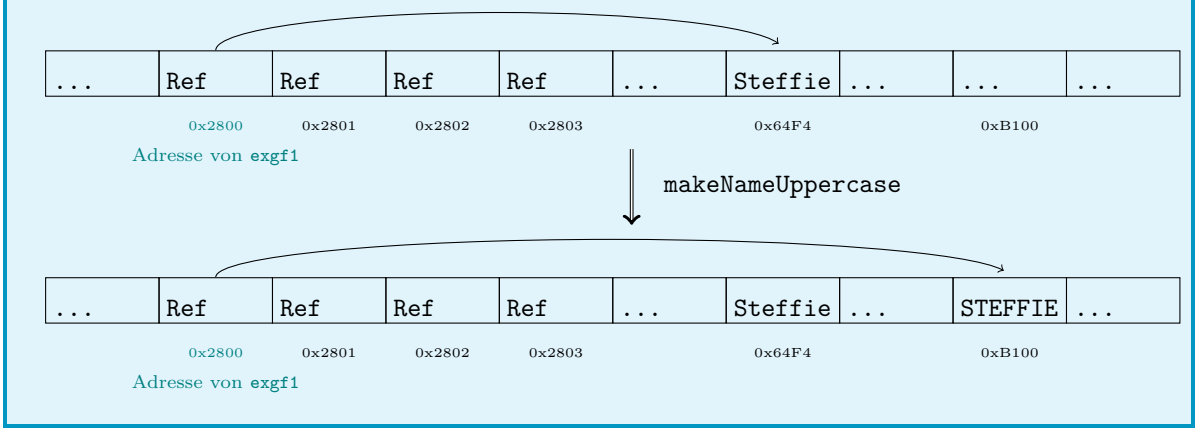


Abbildung 7.1.: Speicherbild: Klasse als Sammlung von Referenzen

Das Symbol `exgf1` ist eine Referenz auf eine Speicherstelle, die selbst wiederum vier Referenzen enthält: Auf `name`, `height`, `upsides` und `downsides`. Wenn wir nun eine Änderung des Namens programmieren, so wird die Referenz `name` eben auf eine neue Speicherstelle geschickt. Der Ort, an dem diese Referenz zu finden ist, ändert sich aber nicht; `exgf1` muss nicht „umziehen“[4].

[4] Genau wie alle Exfreundinnen, mit denen ich bisher zusammen gewohnt habe.

Im folgenden Beispiel dagegen wird in der Funktion eine neue Speicherstelle angelegt; damit sind die Änderungen nach Funktionsaufruf auch nicht mehr sichtbar:

Beispiel: Lokale Instanz

```
1  # Klasse Expartner, Instanz exgf1 und showEx wie oben
2
3  def makePerfect(ex) :
4      ex = Expartner()
5      ex.name = "The perfect one"
6      ex.height = 1.70
7      ex.upsides = {"very intelligent", "very beautiful", "good taste in music",
8                    "has a dog", "musician", "likes coffee"}
9      print("in makePefect:")
10     showEx(ex)
11
12 makePerfect(exgf1)
13 print("in module level:")
14 showEx(exgf1)
```

Ausgabe: Lokale Instanz

```
in makePefect:
The perfect one
  Height      : 1.7
  Upsides     : 'good taste in music', 'has a dog', 'musician', 'likes coffee',
                'very intelligent', 'very beautiful'
  Downsides   :
in module level:
STEFFIE
  Height      : 1.65
  Upsides     : 'has a dog', 'beautiful', 'intelligent'
  Downsides   : 'too attached to her mother', "doesn't like meeting people"
```

Der bedeutende Unterschied liegt also in Zeile 4: Hier wird dem lokalen Symbol `ex` eine Referenz auf eine neue Speicherstelle gegeben (`Expartner()` erzeugt quasi eine neue Speicherstelle). In der Funktion `makePerfect` kann dann wie üblich auf diese *neue* Speicherstelle zugegriffen werden; dies ändert aber nichts daran, dass `exgf1` in der Modulebene immer noch auf den ursprünglichen Datensatz zeigt; an „STEFFIE“ ändert sich nichts[5].

Auch ohne, dass dies in der Definition der Klasse angegeben wurde, können einzelnen Instanzen neue Attribute hinzugefügt werden. Dies betrifft dann aber nur *eine spezielle Instanz*, ohne an der Definition der Klasse etwas zu ändern:

[5] Vielleicht auch besser so – sonst müsste ich mich fragen, warum wir nicht mehr zusammen sind.

Beispiel: Dynamische Erweiterung

```
# Klasse Expartner wie oben

exbf1 = Expartner()
exbf2 = Expartner()

exbf2.eyeColor = "blue"  # neues Attribut nur für exbf2 angelegt

print(exbf2.eyeColor)    # Ausgabe: blue
#print(exbf1.eyeColor)   ! Fehler: Attribut "eyeColor" existiert in exbf1 nicht.
```

Ebenso kann auch die *Klasse als Ganzes* dynamisch verändert werden. Das bedeutet, wir können neue Attribute erstellen, die jede Instanz der Klasse „nachträglich“ hinzugefügt wird. Dies geschieht einfach über

Syntax: Klassen dynamisch erweitern

```
Klasse.Attribut = Wert
```

Hier wird also wirklich die *Klasse* genannt, nicht nur eine einzige Instanz. Entsprechend wirkt sich diese Änderung auch auf alle Instanzen (z. B. `exbf1, exgf2, ...` aus).

7.2. Methods

Bis hierhin haben wir Klassen nur als etwas aufwändigere Variante eines `dict`s benutzt: Wir haben Schlüssel-Wertpaare (z. B. `name: "Sönke"`) gebildet, und diese mit einem gemeinsamen Symbol ansprechbar gemacht. Jetzt aber erweitern wir Klassen um *Methoden*, d. h. Funktionen, die einen speziellen Bezug auf die Instanzen der Klasse haben.

Wir erreichen dies, indem wir die Funktion *in der Klasse* definieren, und als ersten Parameter `self` verlangen:

Syntax: Klassen mit Methoden

```
class Klassenname  :
    def Methode(self, weitereParameter ...) :
        normaler Code
```

Dieser verpflichtende Parameter `self` wird automatisch beim Aufruf übergeben, und enthält eine Referenz auf die Instanz, auf die die Methode angewandt werden soll. Aufgerufen werden Methoden ähnlich dem Zugriff auf die Attribute:

Syntax: Aufruf von Methoden

```
Instanz.Methode(weitereParameter)
```

Dabei müssen wirklich nur `weitereParameter` übergeben werden; der Wert für `self` wird durch die Angabe der Instanz ganz zu Beginn dieses Konstrukts schon ermittelt.

Das folgende Beispiel illustriert Klassen mit Methoden sowie ihre Verwendung:

Beispiel: Klasse mit Methoden

```
class Expartner :
    name      = None
    height    = None
    upsides   = {}
    downsides = {}

    def areThey(self, trait) :
        return (trait in self.upsides) or (trait in self.downsides)

exbf1.name      = "Sebastian"
exbf1.height    = 1.78
exbf1.upsides   = {"intelligent", "handsome", "likes to listen"}
exbf1.downsides = {"obsessed with catching bugs", "always late"}

print("Sebastian is intelligent:", exbf1.areThey("intelligent")) # Ausgabe: True
```

Natürlich hätten wir zu diesem Zweck auch eine Funktion `isExpartner(ex, trait)` schreiben können. Der Vorteil von Methoden liegt darin, dass sie fest einer Klasse zugewiesen sind, und somit nur in einem „sinnvollen Kontext“ verwendet werden können. Der normalen Funktion `isExpartner(ex, trait)` können wir als ersten Parameter auch eine Instanz der Klasse `Partner` übergeben. Natürlich hat die Klasse `Partner` kein Attribut `downsides`; daher wird unser Programm abstürzen, wenn wir dies versuchen. Bei Methoden hingegen kommen wir gar nicht erst in die Gelegenheit, diesen Fehler zu machen. (Leider gilt diese Typensicherheit nicht für die weiteren Parameter. Methoden nehmen also eine Fehlerquelle ab, nicht aber alle).

Daneben gewinnen wir Freiraum bei der Namensgebung: Die Klasse `Partner` kann ihre eigene Methode `areThey(self, trait)` haben. Diese Methode kann auf eine beliebige Art implementiert werden und so der Tatsache Rechnung tragen, dass die Klasse `Partner` eben kein Attribut `downsides` hat. Obwohl die Implementierung sich unterscheidet, sind beide Methoden in der Anwendung („im Interface“) gleich. Wir können sowohl `oneOfMyExGirlfriends.areThey("beautiful")` als auch `myGirlfriend.areThey("beautiful")` setzen, und erhalten ein sinnvolles Ergebnis. Dies ist wichtig, sobald Ihre Codes länger als ein paar Zeilen werden. Sie müssen dann nicht mehr so viele Details über Ihr Programm im Kopf behalten (hieß die Funktion jetzt `isExPartner` oder `isExpartner`?), sondern können darauf vertrauen, dass ähnliche Klassen auch ein ähnliches *Interface* bieten (d. h. dass die Methoden gleich heißen und gleiche Parameter verlangt werden).

Methoden dürfen auch ohne Parameter definiert werden, müssen aber immer noch das obligatorische `self` entgegen nehmen. Damit können wir folgenden Code schreiben, und damit die vergangenen Beziehungen miteinander vergleichen:

Beispiel: Klasse mit Methode ohne Parameter

```
1  class Expartner :
2      traits = {
3          "good taste in music"         :   2,
4          "beautiful"                   :   3,
5          "handsome"                    :   3,
6          "intelligent"                 :   5,
7          "very beautiful"              :   5,
8          "very handsome"               :   5,
9          "very intelligent"            :  10,
10         "doesn't like coding"         : - 4,
11         "doesn't like meeting people" : - 4,
12         "complains a lot"             : - 8,
13         "cheated on me"               : -20,
14         "racist"                      : -30
15     }
16     name      = None
17     height    = None
18     upsides   = {}
19     downsides = {}
20
21     def rating(self) :
22         reVal = 0
23         for trait in self.upsides :
24             if trait in self.traits : reVal += self.traits[trait]
25             else                    : reVal += +1
26         for trait in self.downsides :
27             if trait in self.traits : reVal += self.traits[trait]
28             else                    : reVal += -1
29         return reVal
30
31 exbf2 = Expartner()
32
33 exbf2.name      = "Reginald"
34 exbf2.height    = 1.84
35 exbf2.upsides   = {"very intelligent", "handsome", "plays in a band"}
36 exbf2.downsides = {"flirty with everyone", "complains a lot"}
37
38 print("Rating of time with ", exbf2.name, ": ", exbf2.rating(), sep = "")
```

Ausgabe: Klasse mit Methode ohne Parameter

```
Rating of time with Reginald: 5
```

Das Attribut `traits` ist Teil der Klasse `Expartner`. Damit hat also auch jede Instanz ein eigenes Feld, in dem Bewertungskriterien eingetragen werden können. Wir haben also die Möglichkeit, „nochmal nachzujustieren“. Vielleicht war Arista zwar „nur“ `"beautiful"`, aber doch um ein bisschen hübscher als Steffie. Wir könnten also folgenden – fehlerhaften! – Code schreiben (in den wir aus didaktischen Gründen auch Zeile 23 einfügen und im Anschluss erklären):

Beispiel: Versehentliches Ändern von Referenzen

```
1  # Klasse Expartner wie oben
2
3  exgfs = [Expartner(), Expartner(), Expartner()]
4
5  exgfs[0].name      = "Steffie"
6  exgfs[0].height    = 1.65
7  exgfs[0].upsides   = {"intelligent", "beautiful", "has a dog"}
8  exgfs[0].downsides = {"too attached to her mother",
9                        "doesn't like meeting people"}
10 exgfs[1].name      = "Arista"
11 exgfs[1].height    = 1.60
12 exgfs[1].upsides   = {"very intelligent", "beautiful", "good taste in music"}
13 exgfs[1].downsides = {"doesn't like coding", "doesn't like coffee"}
14 exgfs[2].name      = "Katja"
15 exgfs[2].height    = 1.81
16 exgfs[2].upsides   = {"intelligent", "very beautiful", "musician"}
17 exgfs[2].downsides = {"moody", "cheated on me"}
18
19 print("Evaluation 1:")
20 for ex in exgfs :
21   print(ex.name, ": ", ex.rating(), sep="")
22
23 gfs[1].traits["beautiful"] = 3.5
24 gfs[2].traits = dict()
25
26 print("\nEvaluation 2:")
27 for ex in exgfs :
28   print(ex.name, ": ", ex.rating(), sep="")
```

Ausgabe: Versehentliches Ändern von Referenzen

```
Evaluation 1:
Steffie: 4
Arista: 10
Katja: -10

Evaluation 2:
Steffie: 4.5
Arista: 10.5
Katja: 1
```

Obwohl wir scheinbar nur die `traits` von `exgfs[1]` (also Arista) ändern, sehen wir in der Evaluation 2, dass sich auch die Bewertung von Steffie geändert hat! Dagegen hatte Zeile 23 – das völlige Löschen der Bewertungskriterien – nur Einfluss auf Katja.

Grund ist wie schon öfter, dass Python mit Referenzen arbeitet. Jede Instanz von `Expartner` hat zwar seine eigene *Referenz* `traits`; diese können jedoch auf dieselbe Speicherstelle zeigen. In Zeile 22 wird damit das *gemeinsame* `dict` der Bewertungskriterien von allen drei `Expartner`s geändert. Zeile 24 dagegen legt ein komplett neues (leeres) `dict` an und gibt eine Referenz auf dieses neue `dict` an Katja.

Wir könnten `copy` benutzen, um dieses Problem zu umgehen.

7.3. Magic Methods (Dunders)

Einige Namen für Methoden werden automatisch an bestimmten Stellen aufgerufen. Diese automatischen Aufrufe erlauben es, dem Code eine sehr viel kompaktere, klarere Form zu geben. Allen Methoden ist gemein, dass sie mit einem Doppelten Unterstrich (`__`) beginnen und auch mit diesen beiden Zeichen enden. Wegen diesem *double underscore* werden die hier gezeigten Methoden auch *Dunders* genannt. Daneben ist auch der Name *Magic Methods* gebräuchlich. Alle diese Dunders *können* auch wie ganz normale Methoden aufgerufen werden; vorgesehen sind sie aber zu anderen Zwecken:

7.3.1. Initializer: `__init__`

Mit `__init__` wird ein *Initializer* beschrieben, also eine Funktion, die aufgerufen wird, sobald eine neue Instanz einer Klasse angelegt wird. Üblicherweise werden hier die Werte der einzelnen Attribute gesetzt. Gegebenenfalls können auch Gültigkeitsprüfungen eingeführt werden. Beispielsweise können wir damit unsere Klasse folgendermaßen erweitern:

Beispiel: Klasse mit Initializer

```
1  class Expartner :
2      traits = {
3          "intelligent"                :   5,
4          # ... andere traits ...
5      }
6
7      name = None
8      # ... andere Attribute ...
9
10     def __init__(self, name = None, height = None,
11                  upsides = {}, downsides = {}) :
12         if height < 0 :
13             raise Exception("Your Ex cannot be less than 0.0m tall")
14
15         self.name      = name
16         self.height    = height
17         self.upsides   = upsides
18         self.downsides = downsides
19
20 ex = Expartner("Arista" , 1.60,
21                {"very intelligent", "beautiful", "good taste in music"},
22                {"doesn't like coding", "doesn't like coffee"})
```

Der Befehl `raise` ist ihnen noch neu, und wird im Detail in Kapitel 9 besprochen. An dieser Stelle reicht es zu wissen, dass damit die Programmausführung mit Fehlermeldung (*Your Ex cannot be less than 0.0m tall*) abgebrochen wird.

Entsprechend finden Sie nun folgendes Verhalten:

Beispiel: Ungültiger Parameter im Initializer

```
24 gotye = Expartner("Somebody That I Used To Know", -1)
```

Ausgabe: Ungültiger Parameter im Initializer

```
Traceback (most recent call last):
  File "expartners.py", line 24, in <module>
    Expartner("Somebody That I Used To Know", -1)
  File "expartners.py", line 13, in __init__
    raise Exception("Your Ex cannot be less than 0.0m tall")
Exception: Your Ex cannot be less than 0.0m tall
```

Die Zeilen 20-22 sind der Aufruf des Dunders `__init__`. Sie sehen, wie viel Komfort Sie hiermit gewinnen: Ab jetzt können Sie einfach die Beschreibung der Instanz in einer kompakten Form erledigen, und müssen nicht mehr Attribut für Attribut explizit benennen, bevor Sie die Werte eintragen. Vor allem aber gewinnen Sie so den Luxus einer automatischen Prüfung auf Sinnhaftigkeit. Im obigen Beispiel wird es unmöglich gemacht, eine Instanz von `Expartner` anzulegen, in der das Attribut `height` negativ ist[6].

Auch im Initializer dürfen Attribute definiert werden. Sie erinnern sich, dass wir schon zuvor `ex.eyeColor = "blue"` programmieren konnten. Damit wurde die Instanz `ex` zu einer Abwandlung der Klasse `Expartner` mit dem *zusätzlichen Attribut* `eyeColor`. Wenn wir nun also im Initializer ein neues Attribut definieren, dann hat also jede *Instanz* auch das entsprechende Attribut; *die Klasse selbst* hat dieses Attribut aber nicht.

Wir können hierzu den Befehl `dir` benutzen: Dieser Befehl listet alle Attribute und Methoden eines Objekts auf. Ein Objekt kann dabei sowohl ein Datentyp (d. h. eine Klasse), eine Variable oder auch ein komplexer Ausdruck sein.

Beispiel: `dir`

```
1  class Empty :
2      def __init__(self) :
3          self.attribute = "something"
4
5  myBrain = Empty()
6
7  print("Instanz:\n", dir(myBrain), "\n")
8  print("Klasse :\n", dir( Empty ), "\n")
```

Ausgabe: `dir`

```
Instanz:
 ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
  '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
  '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
  '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
  '__str__', '__subclasshook__', '__weakref__', 'attribute']
```

[6] Natürlich bleibt es möglich, explizit `ex.height = -1` zu programmieren. Zumindest beim Erstellen der Instanz sind Sie so aber auf der sicheren Seite. Gängige Praxis ist es, für jedes Attribut eine Methode `getAttribute(self)` und `setAttribute(self, newValue)` zu schreiben. Wie die Namen es vermuten lassen, sollen diese den Lese- und Schreibzugriff auf die Attribute handeln, und können so zum Beispiel auch Gültigkeitsprüfungen einführen. In Kapitel **??** werden wir hierfür eine besonders bequeme Möglichkeit kennen lernen

```
Klasse :
 ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
  '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
  '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
  '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
  '__str__', '__subclasshook__', '__weakref__']
```

Auf den ersten Blick sehen Sie, dass bereits eine Menge Dunders automatisch angelegt werden, die wir hier leider nur zum Teil besprechen können. Bei genauerer Betrachtung finden Sie, dass die Ausgabe für die Instanz zwar `attribute` aufführt; in der Ausgabe für die Klasse hingegen fehlt dieses Element.

`dir` als Gedächtnisstütze

Sie werden sich oft in der Situation finden, dass Sie zwar wissen *dass* es einen Dunder gibt, der eine bestimmte Aufgabe erledigt, dass sie aber nicht mehr *genau* wissen, wie dieser heißt. Natürlich können Sie in diesem Lehrbuch nachlesen oder schnell im Internet suchen. Häufig ist es aber die schnellste Option, die Ausgabe von `dir` für eine bestehende Klasse (wie `int` oder `list`) zu durchsuchen.

Wenn Sie das tun werden Sie zwangsläufig auch über Methoden stoßen, die Ihnen noch nicht bekannt sind. Viele Methoden haben einen Namen, der auch ohne tiefere Kenntnisse in Programmiertechniken für Sie jetzt gut deutbar ist: Beispielsweise hat die Klasse `float` eine Methode `as_integer_ratio()`. Für andere Methoden werden Sie nach kurzer Webrecherche eine gute (i. d. R. englische) Erklärung finden.

Ich will Sie hiermit dazu ermutigen, die Sprache Python auch selbstständig zu erkunden. Kein Lehrbuch kann den massiven Sprachumfang vollständig erklären; daher gehört es auch zu Ihrer Aufgabe als ProgrammiererIn, sich selbst in Techniken und Konzepte einzuarbeiten.

Attribute in `__init__` anlegen

Wir haben gesehen, dass Attribute der Klasse zunächst von allen Instanzen geteilt werden. Wie Sie gesehen haben, kann dies Probleme bereiten. Wir umgehen dies, indem wir die Attribute zu jeder *Instanz* separat hinzufügen. Das geht am einfachsten, indem wir die Attrubte im Dunder `__init__` definieren. Im folgenden Text behalte ich die Attribute der Klasse `Expartner` in der Definition der Klasse selbst, um interne Mechanismen zu verdeutlichen. Ihnen will ich aber nahelegen, Zeilen der Form `Attribut = Standardwert` in der Methode `__init__` zu schreiben.

7.3.2. Darstellung: `__str__` und `__repr__`

Wie Sie wissen, können wir den Inhalt einer `list` direkt mit `print` ausgeben:

Beispiel: Ausgabe einer `list`

```
1 houseNumbers = [12, 21, 12, 7]  # yes, they really belong to the last four
2 print(houseNumbers)
```

Ausgabe: Ausgabe einer `list`

```
[12, 21, 12, 7]
```

Dagegen erhalten wir für unsere Klasse so nur eine sehr unzufriedenstellende Antwort

Beispiel: Ausgabe einer Instanz von `Expartner` mit Standard-Methode

```
# Klasse Expartner und Instanz ex wie oben
print(ex)
```

Ausgabe: Ausgabe einer Instanz von `Expartner` mit Standard-Methode

```
<__main__.Expartner object at 0x7f00331bc820>
```

Diese Ausgabe sagt uns, dass `ex` ...

- eine Instanz vom Typ `Expartner` ist
- zum Modul `__main__` gehört
- die Speicheradresse 0x7f00331bc820 hat

Die Informationen könnten manchmal zum Debuggen nützlich sein; viel nützlicher wäre es jedoch, wenn wir so Name, Größe, Vor- und Nachzüge erfahren könnten, die sich hinter dem Symbol `ex` verbergen. Doch woher kommt überhaupt die Ausgabe, die wir im Moment sehen?

Der Befehl `print` kann eigentlich nur mit Strings umgehen. Wird ein Parameter übergeben, der nicht zur Klasse `str` gehört, ruft `print` intern automatisch die Funktion `str` auf. Diese Funktion wiederum versucht verschiedene Taktiken, eine Textdarstellung für das Objekt zu erstellen. Die erste Taktik ist dabei immer, die Magic Method `__str__` des Objekts aufzurufen.

Long story short: Wenn wir einen Dunder `__str__` zur Klasse hinzufügen, können wir eine schönere Ausgabe erreichen. Dies kann z.B. so aussehen:

Beispiel: Ausgabe einer Instanz von `Expartner` mit eigener Methode

```
class Expartner :
    # ... Attribute wie oben ...
    # ... __init__ wie oben ...
    # ... rating wie oben ...

    def __str__(self) :
        reVal  = self.name + "\n"
        reVal += "  Height      : " + str(self.height)          + "\n"
        reVal += "  Upsides     : " + str(self.upsides)  [1:-1] + "\n"
        reVal += "  Downsides   : " + str(self.downsides)[1:-1] + "\n"
        reVal += "  Total Rating: " + str(self.rating())
        return reVal

ex = Expartner("Sönke" , 1.81,
               {"intelligent", "quite handsome", "cheerful nature"},
               {"superficial", "racist"})

print(ex)
```

Ausgabe: Ausgabe einer Instanz von `Expartner` mit eigener Methode

```
Sönke
  Height      : 1.81
  Upsides     : 'intelligent', 'quite handsome', 'cheerful nature'
  Downsides   : 'racist', 'superficial'
  Total Rating: -20
```

Halten Sie sich vor Augen: Der Aufruf von `__str__` geschieht durch die Funktion `str`, die wiederum von `print` aufgerufen wird. Das heißt, auch wenn Sie direkt `str` benutzen, erhalten Sie den String, wie er von Ihrer Methode vorbereitet wird.

Beispiel: `str` und Klassen

```
1  # Klasse Expartner und Instanz ex wie oben
2  text = str(ex)
3  lines = text.split("\n")
4  print(lines[0])
```

Ausgabe: `str` und Klassen

```
Sönke
```

Natürlich können Sie den Dunder auch direkt aufrufen. Die Zeile `ex.__str__()` hat exakt dasselbe Ergebnis wie der Aufruf über `str`. Diese etwas eigenartig anmutende Konstruktion wurde gewählt, um die Logik anderer Sprachen nachzuahmen, wo eben *Funktionen* zur Typ-Umwandlung benutzt werden, nicht aber *Methoden*.

Ähnlich wie `__str__` wird `__repr__` über `repr` aufgerufen. Erzeugt wird der oben gezeigte Debug-String mit der Adresse der Instanz. Der Code hierfür wird automatisch mit dem Anlegen einer Klasse erstellt (mehr dazu in Abschnitt 7.4), kann aber leicht überschrieben werden. Die Ausgabe von `repr` soll eine kompakte Darstellung des Datenobjekts enthalten, die nicht zwingend für den Enduser gedacht ist. Prinzipiell aber kann hier ein beliebiger Alternativtext zur Ausgabe von `str` erzeugt werden.

7.3.3. Vergleich: `__eq__`, `__ne__`, `__gt__`, `__ge__`, `__lt__`, `__le__`

Die Instanzen von Klassen können oft miteinander verglichen werden. Das bedeutet sowohl, dass wir die Frage stellen können, ob zwei Instanzen der Klasse denselben Wert beschreiben (`a == b`, z. B. `"Wort" == "Wort"`), also auch, in welcher Reihenfolge man diese Instanzen anordnen würde, (`a < b`, z. B. `"Apfel" < "Zitrone"`).

Am Beispiel von Strings hatten wir schon kennengelernt, dass Python diese *lexikographisch* („alphabetisch mit Regeln für Ziffern und Sonderzeichen“) ordnet. Im Allgemeinen gibt es aber kein „natürliches Kriterium“, nach dem Klasseninstanzen verglichen oder geordnet werden sollen. Im Kontext des Beispiels des Kapitels: Wir könnten die Instanzen von `Expartner` beispielsweise nach Körpergröße, nach dem Namen, nach Bewertung der Beziehung oder jedem beliebigen anderen Kriterium ordnen.

Wir müssen also definieren, was unter `a == b`, `a != b`, `a < b`, `a <= b`, `a > b` und `a >= b` zu verstehen ist. Wie Sie dies aus der Struktur des Kapitels vermutlich erwarten, geschieht das über die Dunders `__eq__` (equal), `__ne__` (not eqal), `__gt__` (greater than), `__ge__` (greater than or equal), `__lt__` (less than) und `__le__` (less than or equal). Die Funktionen werden aufgerufen, wenn einer der Vergleichsoperatoren an einer Instanz unserer Klasse angewandt wird. Die Methode muss neben `self` einen weiteren Parameter entgegennehmen – die Instanz, gegen die verglichen werden soll. Der Rückgabewert soll `True` sein, wenn

die Vergleichsbedingung erfüllt ist (also z. B.: die Methode `__eq__` soll `True` zurückgeben, wenn die beiden Instanzen tatsächlich gleichwertig sind), andernfalls soll das Ergebnis `False` sein.

Beispiel: da wir hier die Vergangenheit mit wissenschaftlichen Methoden aufarbeiten wollen, bietet es sich an, die Beziehungen nach der Bewertung (dem Ergebnis der Methode `rating`) zu ordnen:

Beispiel: Implementation der Vergleichsmethoden

```
class Expartner:
    # ... Attribute wie oben ...
    # ... __init__ wie oben ...
    # ... rating wie oben ...

    def __eq__(self, other) :
        return (self.rating() == other.rating())

    def __ne__(self, other) :
        return (self.rating() != other.rating())

    def __lt__(self, other) :
        return (self.rating() <  other.rating())

    def __le__(self, other) :
        return (self.rating() <= other.rating())

    def __gt__(self, other) :
        return (self.rating() >  other.rating())

    def __ge__(self, other) :
        return (self.rating() >= other.rating())
```

Sobald diese Magic Methods implementiert sind, können alle Operationen durchgeführt werden, die einen Vergleich benötigen:

Beispiel: Anwendung der Vergleichsmethoden

```
# Klasse Expartner wie oben

exbfs = [
    Expartner("Sebastian", 1.65,
              {"intelligent", "handsome", "likes to listen"},
              {"obsessed with catching bugs", "always late"}),
    Expartner("Reginald", 1.84,
              {"very intelligent", "handsome", "plays in a band"},
              {"flirty with everyone", "complains a lot"}),
    Expartner("Sönke", 1.81,
              {"intelligent", "quite handsome", "cheerful nature"},
              {"superficial", "racist"})]
```

```
13  exgfs = [
14      Expartner("Steffie", 1.65,
15                {"intelligent", "beautiful", "has a dog"},
16                {"too attached to her mother", "doesn't like meeting people"}),
17      Expartner("Arista" , 1.60,
18                {"very intelligent", "beautiful", "good taste in music"},
19                {"doesn't like coding", "doesn't like coffee"}),
20      Expartner("Katja"  , 1.81,
21                {"intelligent", "very beautiful", "musician"},
22                {"moody", "cheated on me"})
23      ]
24
25  if exbfs[0] < exbfs[1] :
26      print(f"It was better with {exbfs[1].name} than with {exbfs[0].name}")
27  else :
28      print(f"It was better with {exbfs[0].name} than with {exbfs[1].name}")
29
30  print("All of them, from worst to best")
31  allExes = sorted(exgfs + exbfs)
32  for ex in allExes :
33      print(ex)
```

Beachten Sie zuerst Zeile 25: Wir vergleichen *direkt* Instanzen der Klasse `Expartner`. Natürlich wäre es immer noch möglich, zu fragen ob `exbfs[0].rating() < exbfs[1].rating()`. Diese Bedingung hätte in Zeile 25 exakt denselben Effekt. Diese explizite Angabe können Sie sich von hier weg aber sparen. Durch die Implementation von `__eq__` etc. „weiß“ Python, nach welchen Regeln ein Vergleich stattzufinden hat. Natürlich hätte es wenig Sinn, diese sechs Methoden zu implementieren, wenn diese nur an einer einzigen Stelle verwendet werden (wie in diesem Beispiel). Im Allgemeinen aber werden Sie Klassen und Instanzen in sehr viel größeren Kontexten verwenden, und gewinnen so signifikant Komfort.

Ein weiterer Vorteil der Verwendung dieser Duders liegt darin, dass viele Funktionen „in ihrem Inneren“ Vergleiche durchführen, und damit natürlich nur anwendbar sind, wenn der notwendige Vergleich klar definiert ist. Dies gilt zum Beispiel für die Funktion `sorted`. Wie Sie sich aus Kapitel 4 erinnern (und wie der Name vermuten lässt), wird dieser Befehl benutzt, um Datencontainer zu sortieren. Um diese Aufgabe zu erledigen, führt `sorted` für Elemente `a, b` des Containers jeweils Vergleiche `a < b`, ... durch. Ohne die Dunders, die wir gerade besprechen, wäre das nicht möglich. Natürlich könnten wir immer noch ein Lambda übergeben, das das Ergebnis von `rating` als Sortierkriterium markiert (vgl. Hierzu Kapitel 6). Durch die hier gewählte Methode erhalten wir aber die Wahl des Sortierkriteriums automatisch.

Mit diesen Überlegungen verstehen Sie, weshalb wir die folgende Ausgabe erhalten:

Ausgabe: Anwendung der Vergleichsmethoden

```
It was better with Sebastian than with Reginald
All of them, from worst to best
Sönke
  Height      : 1.81
  Upsides     : 'intelligent', 'cheerful nature', 'quite handsome'
  Downsides   : 'racist', 'superficial'
  Total Rating: -20
Katja
  Height      : 1.81
  Upsides     : 'intelligent', 'very beautiful', 'musician'
  Downsides   : 'moody', 'cheated on me'
  Total Rating: -10
Steffie
  Height      : 1.65
  Upsides     : 'intelligent', 'has a dog', 'beautiful'
  Downsides   : "doesn't like meeting people", 'too attached to her mother'
  Total Rating: 4
Reginald
  Height      : 1.84
  Upsides     : 'very intelligent', 'handsome', 'plays in a band'
  Downsides   : 'flirty with everyone', 'complains a lot'
  Total Rating: 5
Sebastian
  Height      : 1.65
  Upsides     : 'intelligent', 'likes to listen', 'handsome'
  Downsides   : 'obsessed with catching bugs', 'always late'
  Total Rating: 7
Arista
  Height      : 1.6
  Upsides     : 'good taste in music', 'very intelligent', 'beautiful'
  Downsides   : "doesn't like coffee", "doesn't like coding"
  Total Rating: 10
```

Beachten Sie, dass der *Datentyp* der zu vergleichenden Objekte eine wesentliche Rolle spielt: die beiden folgenden Beispiele werden jeweils mit Fehlermeldung abgebrochen:

Beispiel: Inkompatible Datentypen (1)

```
# Expartner, allExes wie oben
print(allExes[0] < 0)
```

Beispiel: Inkompatible Datentypen (2)

```
# Expartner, allExes wie oben
print(0 > allExes[0])
```

In beiden Fällen erhalten Sie die Fehlermeldung:

Fehlermeldung: Inkompatible Datentypen (1)

```
...
AttributeError: 'int' object has no attribute 'rating'
```

Durchdenken wir zuerst das Beispiel: Inkompatible Datentypen (1). Der Ausdruck `allExes[0] < 0` wird also übersetzt in einen Aufruf der Methode `Expartner.__lt__(allExes[0], 0)`, d. h. `allExes[0]` wird als erster Parameter `self` übergeben, und `0` ist der Wert von `other`. Damit ist `other` also vom

Typ `int` und hat folglich auch keine Methode `rating` – diese Methode ist nur für Instanzen der Klasse `Expartner` erklärt. Damit ist auch die Fehlermeldung für diesen Fall verständlich. Sie als ProgrammiererIn müssen also entweder darauf achten, nur „sinnvolle“ Vergleiche anzufordern, oder aber in den Dunders `__eq__`, ... verschiedene Vergleichsmethoden für verschiedene Datentypen anzubieten. Hierzu können Sie beispielsweise den folgenden Ansatz wählen:

Beispiel: Typabhängige Vergleichsmethoden

```
class Expartner:
    # ... Attribute wie oben ...
    # ... rating wie oben ...

    def __eq__(self, other) :
        if type(other) in (int, float) : return self.rating() == other
        if type(other) == Expartner    : return self.rating() == other.rating()
        return NotImplemented

    # andere Vergleichsmethoden ...
```

Wie Sie sich erinnern, erhalten Sie mit der Funktion `type` den Datentypen einer Variablen. Dabei ist es nicht von Belang, ob es sich um einen „eingebauten“ (nativen) Datentypen wie `int` handelt, oder um eine von Ihnen erstellte Klasse. Mit diesem Hilfsmittel können Sie nun also abhängig von der „Beschaffenheit“ von `other` also unterschiedliche Vergleichsmethoden anbieten, die allesamt über die üblichen Vergleichsoperatoren (==, <, ...) genutzt werden können.

Neu ist an dieser Stelle der Rückgabewert `NotImplemented`; dieser führt auch direkt zum Beispiel: Inkompatible Datentypen (2).

Bei den Vergleichen wie `a < b` wird implizit das linke Element (also `a`) als *Basisklasse des Vergleichs* herangezogen. Das heißt, dass die Vergleichsmethode (`__lt__`) in der Klasse gesucht wird, von der `A` eine Instanz ist. Damit wird `a` als `self` übergeben, während `b` als `other` an die Methode weitergeleitet wird.

Der Vergleich `0 > allExes[0]` würde also zuerst die Methode `__lt__` der Klasse `int` aufrufen, und zwar mit `0` als `self` und mit `allExes[0]` als `other`. Wie Sie sich vorstellen können, hatten die Entwickler von Python nicht vorhergesehen, dass ich im Rahmen dieses Kurses eine Vergleichsmethode von `int`s und `Expartner`s benötige. Stattdessen sind nur die Vergleiche zu den „in Python eingebauten“ Datentypen implementiert. Für jeden anderen Datentyp wird zunächst der Fehlerwert `NotImplemented` zurückgegeben.

Die interne Mechanik des Python-Interpreters reagiert genau auf dieses Signal: Wenn eine Vergleichsmethode zu `NotImplemented` ausgewertet wird, so zieht der Interpreter das rechtsseitige Vergleichsobjekt (das `b` in `a < b`) heran, und versucht hier eine geeignete Vergleichsmethode zu finden. Erst wenn dies fehlschlägt (d. h. wenn auch die Klasse von `b` den Wert `NotImplemented` zurück gibt), „gibt der Python-Interpreter auf“. Sie finden die Fehlermeldung

Fehlermeldung: Inkompatible Datentypen (2)

```
TypeError: '<' not supported between instances of 'Expartner' and 'int'
```

Wenn Sie also die Klassen `Apple` und `Orange` schreiben, sollte ein `Apple` wissen, wie er sich mit einem anderen `Apple` vergleicht; auch eine `Orange` muss wissen, wie sie sich mit einer `Orange` vergleicht. Es

reicht aber, in `Orange` den Vergleich mit `Apple` zu implementieren. `Apple` selbst kann sich mit einem `NotImplemented` begnügen[7].

7.3.4. Gebrauch als Funktion: `__call__`

Klasseninstanzen können auch als Funktionen gebraucht werden. Über die Syntax:

Syntax: Aufruf einer Instanz

```
Instanz(Parameterliste)
```

wird der Dunder `__call__(self, Parameterliste)` aufgerufen. Wie bei allen Methoden darf die Parameterliste auch leer sein, Default-Werte annehmen und/oder variadische Elemente enthalten. Konkret kann dies so aussehen:

Beispiel: Aufrufbare Instanzen

```
class Expartner :
    # wie oben

    def __call__(self) :
        print("You shouldn't call your Ex'es. Leave the past behind.")

ex = Expartner("Sandra", 1.70)
ex()
```

Dies kann etwa nützlich werden, wenn eine Funktion sehr viele Parameter bräuchte, wenn die Werte dieser Parameter sich durch den Code der Funktion verändern und im Nachhinein erhalten bleiben sollen, oder eine thematische Einheit bilden. Sie können sich etwa ein kleines Spiel vorstellen: Das Spiel muss zuerst konfiguriert werden (Schwierigkeitsgrad, Größe des Spielfelds, Spielerzahl, ...), bevor es gestartet werden kann. In diesem Fall ist es ein sinnvolles Programmdesign, eine Klasse `Game` zu erstellen, die alle diese Daten als Attribute verwaltet. Von dieser Klasse wird eine Instanz `match` erstellt. Sind (z. B. über ein geeignetes User-Interface) alle Werte sinnvoll festgelegt, so kann die Instanz `match` gestartet, d. h. aufgerufen werden. Nach Ende der Partie steht das Objekt `match` weiter zur Verfügung, und kann z. B. für eine Highscore-Liste ausgewertet werden.

7.3.5. Rechenoperatoren: `__add__`, `__sub__`, `__mul__`, `__truediv__`, `__floordiv__`, `__matmul__`

Mit Instanzen von Klassen kann auch gerechnet werden wie mit „normalen Zahlen“. Natürlich ist dies nicht immer sinnvoll; beispielsweise habe ich keine Idee, welche Bedeutung dem Ausdruck `CharlottesLastEx - MyLastEx` zukommen sollte. Daher möchte ich Sie darum bitten, sich für die folgenden Abschnitte in ein neues Beispiel einzudenken. Wir werden hier *Matrizen* wie aus der Mathematik bekannt behandeln.

Eine Matrix ist eine Sammlung von Werten, die in einer rechteckigen Tabelle angeordnet sind, d. h. eine Sammlung von Zeilen, wobei jede Zeile eine Sammlung von gleich vielen Zahlen ist. Eine Matrix lässt sich also als *Liste von Listen* abbilden. Um eine Matrix als Klasse abzubilden, ist es also sinnvoll, ein Attribut `data` zu definieren, das diese Liste von Listen beinhaltet.

[7] Dies scheint auch die Denkweise des Konzerns zu sein...

Bei der Arbeit mit Matrizen ist es auch wichtig, die Größe der Matrix zu kennen. Beispielsweise ist die Matrixmultiplikation nur dann möglich, wenn die Zahl der Spalten der linken Matrix mit der Zahl der Zeilen der rechten Matrix zusammenfällt. Wir können mit `len` diese Information (Zeilenzahl und Spaltenzahl) in Erfahrung bringen. Wir wollen daher diese Information als Attribut `shape` in einem `tuple` speichern. (Ebenso könnten wir zwei `int`-Attribute `rows, cols` verwenden. Vorlieben variieren.)

Jede Instanz von `Matrix` braucht seine eigenen Attribute `data, shape`. Es gibt keine Zahlen, die sich alle Matrizen teilen. Daher werden wir die genannten Attribute nicht direkt nach dem Header `class Matrix` anlegen, sondern erst in der Methode `__init__`. Sie erinnern sich: Diese Methode wird aufgerufen, sobald eine neue Instanz angelegt wird. Wenn wir hier Attribute einpfelgen, sorgen wir auch dafür, dass sie rein an die Instanz gebunden sind. Komplikationen wie im *Beispiel: Versehentliches Verändern von Referenzen* werden so vermieden.

Wenn wir eine neue Matrix anlegen, müssen wir ihre Größe und ihren Inhalt festlegen. Es gibt aber häufig Situationen, in denen wir die Matrixwerte erst Stück für Stück aufbauen können. Daher soll es auch möglich sein, eine „leere Matrix“ mit fester Größe zu erstellen. Diese „leere Matrix“ füllen wir mit Nullen auf. Wenn dagegen alle Werte schon bekannt sind, so wollen wir eine Liste von Listen als Parameter entgegen nehmen können und diese in unsere Matrix-Struktur einpflegen können.

Mit diesen Überlegungen können wir den folgenden Code schreiben:

Beispiel: Klasse Matrix

```
1  import copy
2
3  class Matrix :
4      def __init__(self, shape, data = None) :
5          # check for validity of shape parameter
6          if type(shape) != tuple         :
7              raise Exception("Expected a tuple!")
8          if len (shape) != 2             :
9              raise Exception("Expected a 2-tuple!")
10         if shape[0] < 1 or shape[1] < 1 :
11             raise Exception("Expected positive values")
12
13         self.shape = shape
14
15         if data == None :
16             # fill with zeros
17             self.data  = []
18             for     row in range(shape[0]) :
19                 self.data.append([])
20                 for col in range(shape[1]) :
21                     self.data[row].append(0)
22         else :
23             self.setData(data)
```

```
24      def setData(self, data) :
25          if type(data)    != list : raise Exception("Expected a list of lists")
26          if type(data[0]) != list : raise Exception("Expected a list of lists")
27
28          rows = len(data)
29          cols = len(data[0])
30
31          # check if all provided rows have same length
32          for row in data :
33              if len(row) != cols : raise Exception("Ill-formed matrix")
34
35          self.shape = (rows, cols)
36          self.data  = copy.deepcopy(data)
```

Zuerst testen wir, ob die angegebene `shape` in unserem Kontext sinnvoll ist. Falls nein, brechen wir mit Fehlermeldung ab (Zeilen 6-11).

In Zeile 14 übertragen wir die Daten des Parameters `shape` in das Attribut `self.shape`. Lassen Sie sich nicht von den gleichen Namen verwirren: `shape` kommt „von außen“ und ist nicht dasselbe wie das Attribut `self.shape`, das zur Matrix gehört. Da wir hier mit `tuple`s arbeiten (die immutable sind), ist eine versehentliche Veränderung der Werte ausgeschlossen. Wir können ohne weiteres die Referenz übernehmen.

In den Zeilen 17-21 erzeugen wir eine Null-Matrix der angeforderten Größe. In eine leere Liste fügen wir zunächst leere Listen (Zeilen) ein, die wir im Anschluss mit Nullen auffüllen.

Der Parameter `data` enthält gegebenenfalls Werte, die in die Matrix eingetragen werden sollen. Falls wir beim Anlegen der Instanz keine Daten bereit haben, können wir dies durch `data = None` signalisieren. Es könnte auch sein, dass wir die Werte einer bestehenden Matrix später updaten wollen, ohne die Matrix selbst komplett neu anzulegen. Daher lagern wir das Befüllen mit Werten in eine eigene Methode aus, und rufen von `__init__` diese Methode `setData` auf (Zeile 23).

In `setData` prüfen wir wieder zunächst den Datentypen (Liste von Listen; Zeile 25-26), und dann, ob die übergebenen Werte auf tatsächlich eine rechteckige Matrix beschreiben (Zeile 31-35). Um nicht versehentlich Referenzen zu übernehmen, benutzen wir `copy.deepcopy`, um die Werte in unsere Matrix zu übernehmen.

Schon um zu testen, ob unsere Methoden so funktionieren, wie wir das wollen, fügen wir noch die Dunders `__str__` und `__repr__` hinzu:

```
37      def __repr__(self) :
38          return f"<matrix of dimension {self.shape[0]}x{self.shape[1]}>"
39
40      def __str__(self) :
41          reVal = ""
42          for row in self.data :
43              reVal += str(row) + "\n"
44          return reVal[:-1]               # remove superfluous line break
```

Nachdem wir so kurz wiederholt haben, wie wir eine Klasse konzipieren und ihr vier Methoden hinzugefügt haben, können wir uns nun der Aufgabe stellen, das Rechnen mit Matrizen zu implementieren. Wenn `A` und `B` Instanzen von `Matrix` sind, so wollen wir, dass Ausdrücke wie `A + B` korrekt ausgewertet werden.

Zu diesem Zweck schreiben wir für jeden Operator einen speziellen Dunder. Im Falle der Addition ist das `__add__`:

Beispiel: Methode `__add__`

```
45      def __add__(self, other) :
46          if type(other) != Matrix      : return NotImplemented
47          if self.shape  != other.shape : raise Exception("Incompatible matrices")
48
49          reVal = Matrix(self.shape)
50
51          for     row in range(self.shape[0]) :
52              for col in range(self.shape[1]) :
53                  reVal.data[row][col] = self.data[row][col] + other.data[row][col]
54          return reVal
```

Wie bei den Vergleichsoperatoren (`__eq__`, ...) bezieht sich `self` auf das Symbol links des Operators (also auf `A` in `A + B`), während `other` eine Referenz auf das rechte Symbol (`B`) erhält.

Wir prüfen wieder, ob der Datentyp des rechten Operanden (`B`) geeignet ist (Zeile 46). Falls nein geben wir den Fehlerwert `NotImplemented` zurück, der Python die Chance gibt, verschiedene Fehlerbehandlungsmethoden anzuwenden. (Wir werden dies im nächsten Punkt besprechen). Falls die Größen der Matrizen nicht aufeinander passen, brechen wir das Programm mit einer Fehlermeldung ab (Zeile 47).

Für die eigentliche Rechnung müssen wir eine neue Matrix bereit stellen, die das Ergebnis aufnimmt (Zeile 49). In diese können wir jetzt jeweils die komponentenweise Summe eintragen (Zeilen 51-53). Schließlich muss dieses so gefundene Ergebnis auch der aufrufenden Stelle mitgeteilt werden – wir schließen also mit einer **return**-Zeile ab (Zeile 54).

Das Zeichen $*$ wird durch einen Aufruf der Methode `__mul__` aufgelöst. Hier wollen wir die *komponentenweise* Matrixmultiplikation sowie die Multiplikation mit einem Skalar umsetzen:

Beispiel: Methode `__mul__`

```
55      def __mul__(self, other) :
56          reVal = Matrix(self.shape)
57
58          if type(other) == Matrix :
59              if self.shape  != other.shape :
60                  raise Exception("Incompatible matrices")
61
62              for     row in range(self.shape[0]) :
63                  for col in range(self.shape[1]) :
64                      reVal.data[row][col] = self  .data[row][col] * \
65                                             other.data[row][col]
66
67              return reVal
```

```
68          elif type(other) in (int, float, complex) :
69              for     row in range(self.shape[0]) :
70                  for col in range(self.shape[1]) :
71                      reVal.data[row][col] = self.data[row][col] * other
72
73              return reVal
74
75          else :
76              return NotImplemented
```

Hier finden wir dieselbe Grundstruktur wie schon bei `__add__` wieder: Vorbereiten einer Ergebnis-Variable `reVal`, Typenprüfung, Schrittweiser Aufbau der Lösung, und „Abschicken“ der Lösung mit `return` bzw. Zurückgeben des Fehlerwertes `NotImplemented`, wenn bei der Typenprüfung ein Fehler aufgetreten ist.

Da wir durch die Multiplikation mit dem Skalar `-1` jetzt auch das Vorzeichen jeder Matrix-Komponente umdrehen können, lassen sich die Subraktion (`__sub__`) und die *exakte Division* (`__truediv__`) mit sehr geringem Aufwand schreiben:

Beispiel: Methoden `__sub__` und `__truediv__`

```
77      def __sub__(self, other) :
78          return self + (other * -1)
79
80      def __truediv__(self, other) :
81          return self * (1 / other)
```

Für die *Integerdivision* (Division mit Abrunden, `__floordiv__`) dagegen müssen wir unser übliches Schema anwenden:

Beispiel: Methode `__floordiv__`

```
82      def __floordiv__(self, other) :
83          if type(other) in (int, float, complex) :
84              for     row in range(self.shape[0]) :
85                  for col in range(self.shape[1]) :
86                      reVal.data[row][col] = self.data[row][col] // other
87
88              return reVal
89
90          else :
91              return NotImplemented
```

Vielleicht haben Sie sich schon zuvor gewundert, dass wir die Methode `__mul__` *komponentenweise* implementiert haben. Für die übliche Matrix-Multiplikation

$$(AB)_{ij} = \sum_k A_{ik} B_{kj}$$

ist nämlich der Dunder `__matmul__` vorgesehen. Dieser wird über den Operator `@` aufgerufen:

Beispiel: Methode `__floordiv__`

```
92      def __matmul__(self, other) :
93          if type(other) == Matrix :
94              if other.shape[0] != self.shape[1] :
95                  raise Exception("Incompatible Shapes")
96              resultShape = (self.shape[0], other.shape[1])
97              result = Matrix(resultShape)
98
99              for         i in range(resultShape[0]) :
100                 for     j in range(resultShape[1]) :
101                     for k in range(other.shape[0]) :
102                         result.data[i][j] += self.data[i][k] * other.data[k][j]
103             return result
104         return NotImplemented
```

Natürlich sollten wir unsere Methoden auch testen:

Beispiel: Tests einiger Methoden

```
105 A = Matrix((2,2), [[1,2], [3,4]])
106 B = Matrix((2,3), [[9,8,7], [6,5,4]])
107
108 print("A")
109 print( A )
110 print("B")
111 print( B )
112
113 print("A + A")
114 print( A + A )
115
116 print("A * A")
117 print( A * A )
118
119 print("A @ B")
120 print( A @ B )
```

Ausgabe: Tests einiger Methoden

```
A
[1, 2]
[3, 4]
B
[9, 8, 7]
[6, 5, 4]
A + A
[2, 4]
[6, 8]
A * A
[1, 4]
[9, 16]
```

```
A @ B
[21, 18, 15]
[51, 44, 37]
```

7.3.6. Rechtsseitige Rechenoperatoren: `__radd__`, `__rsub__`, `__rmul__`, `__rtruediv__`, `__rfloordiv__`

Im Moment können wir zwar für eine `Matrix` `A` den Ausdruck `A * 2` berechnen; für `2 * A` dagegen erhalten wir noch die Fehlermeldung

Fehlermeldung: Multiplikation mit Skalar von links

```
Traceback (most recent call last):
  File "matrix.py", line 139, in <module>
    print( 2 * A )
TypeError: unsupported operand type(s) for *: 'int' and 'Matrix'
```

Um dies zu verstehen, erinnern wir uns, dass durch den Operator `*` die Methode `__mul__` des *linken* Objekts aufgerufen wird, also hier der `int`-Instanz `2`. Natürlich „weiß `int` nicht“, wie es mit Matrizen umzugehen hat, und gibt damit den Fehlerwert `NotImplemented` zurück, der hier zu dieser Fehlermeldung führt. Bevor Python aber „aufgibt“, versucht der Interpreter, die Methode `__rmul__` des *rechten* Operanden (also `A`) aufzurufen.

Fügen wir also in unserer Klassen-Definition von `Matrix` noch folgende Zeilen ein:

Beispiel: Tests einiger Methoden

```
105     def __rmul__(self, other) :
106         return self * other
```

so kann Python auch `2 * A` berechnen: für die Methode `__rmul__` verweist der Parameter `other` auf den *linken* Operanden (also die `2`). Wir haben hier die Reihenfolge des Ausdrucks umgedreht, und können so den schon bestehenden Code von `__mul__` ausnutzen.

Ähnlich wie für die Multiplikation existieren auch rechtsseitige Methoden für die Addition, Subtraktion, Fließkomma- und Ganzzahldivision (`__radd__`, `__rsub__`, `__rtruediv__`, und `__rfloordiv__`). Alle diese Methoden haben dasselbe Verhalten, d. h. sie werden aufgerufen, nachdem ihr linksseitiges Pendant `NotImplemented` zurückgegeben hat.

7.3.7. Index-Zugriff: `__getitem__`, `__setitem__`

Wir wollen auch einzelne Matrixelemente lesen und schreiben können, so wie wir auch bei `list`s etc. Elemente aus der Liste herausgereifen können. Zum *Lesen* von einzelnen Werten dient der Dunder `__getitem__`. Neben dem obligatorischen `self` muss die Methode einen Index annehmen, und soll das durch diesen Index bezeichnete Objekt zurück geben. Wie wir das von `list`s kennen, geschieht der lesende Index-Zugriff über die Syntax `Instanz[Index]`.

Klassischerweise versteht man unter einem Index eine einzige Ganzzahl; es spricht aber nichts dagegen, hier andere Datentypen auch zu verwerten. Da eine Matrix ein *zweidimensionales* Objekt ist, könnten wir auch einen Tupel aus zwei Werten akzeptieren. Wenn `A` also eine Instanz von `Matrix` ist, so soll

`A[1,2]` also das Matrixelement in der *zweiten* Zeile und der *dritten* Spalte ansprechen. (Die Zählung beginnt wie immer bei der Null).

Falls tatsächlich nur eine einzelne Zahl übergeben wird, so ist es sinnvoll, diese als *Zeilenindex* aufzufassen, und die ganze Zeile der Matrix zurück zu geben. Dies kann aus unserer gewählten Struktur heraus leicht implementiert werden, und hat den weiteren Vorteil, dass Konstrukte wie `A[row][column]` immer noch funktionieren. Hierbei wird zuerst `A[row]` ausgewertet und gibt dann eine `list` zurück; auf diese Liste wird dann wieder der Indexzugriff mit index `column` ausgewertet.

Eine Implementierung dieser Gedanken kann so aussehen:

Beispiel: Lesender Indexzugriff

```
107       def __getitem__(self, index) :
108         if type(index) == int :
109           if 0 <= index < self.shape[0] :
110             return self.data[index]
111           else :
112             raise Exception("Invalid Index")
113
114         elif type(index) == tuple :
115           if len(index) != 2 : raise Exception("Invalid Dimension")
116
117           row = index[0]
118           col = index[0]
119
120           if (0 <= row < self.shape[0]) and (0 <= col < self.shape[1]) :
121             return self.data[row][col]
122           else :
123             raise Exception("Invalid Index")
124
125         else :
126           raise Exception("Invalid Index")
```

Ähnlich Funktioniert der Dunder `__setitem__`, der aufgerufen wird, wenn über die Syntax `Instanz[Index] = Ausdruck` ein Wert gesetzt werden soll. Neben den zuvor schon genannten Parametern `self` und `index` ist hier natürlich ein dritter Parameter `value` vonnöten. Einen Rückgabewert gibt es bei der Wertzuweisung dagegen nicht.

Wir wollen auch hier der Logik folgen, dass sowohl `int`-Werte als auch `tuple`s akzeptiert werden sollen. Entsprechend implementieren wir die Methode folgendermaßen:

Beispiel: Schreibender Indexzugriff

```
127       def __setitem__(self, index, value) :
128         if type(index) == int :
129           if type(value) != list          : raise Exception("Invalid Data Type")
130           if len (value) != self.shape[1] : raise Exception("Invalid Dimension")
131
132           if 0 <= index < self.shape[0] :
133             self.data[index] = value
134           else :
135             raise Exception("Invalid Index")
136
137         elif type(index) == tuple :
138           if len(index) != 2 : raise Exception("Invalid Dimensionality")
139           row = index[0]
140           col = index[0]
141
142           if (0 <= row < self.shape[0]) and (0 <= col < self.shape[1]) :
143             self.data[row][col] = value
144           else :
145             raise Exception("Invalid Index")
146
147         else :
148           raise Exception("Invalid Index")
```

Mit diesen Ergänzungen zu unserer Klasse sind uns nun folgende Zugriffe möglich:

Beispiel: Tests – Indexzugriff

```
149  A = Matrix( (2,2) )
150  A[0] = [1, 2]
151  A[1, 1] = 4
152  print(A)
153  print(A[0, 1])
154  print(A[1])
```

Ausgabe: Tests – Indexzugriff

```
[1, 2]
[0, 4]
1
[0, 4]
```

Slices

Aus Kapitel 4 kennen Sie das Konzept Slicing. Sie können Ihren Klassen ähnliche Fähigkeiten beibringen, wenn Sie den Datentyp `slice` als Parameter akzeptieren. Es handelt sich also um Instanzen der Klasse `slice`, die unter anderem die Attribute `start`, `step` und `step` haben.

Experimentieren Sie damit herum!

7.3.8. Zahl der Elemente: `__len__`

Ähnlich wie `str` leitet auch der Befehl `len` nur auf den Dunder `__len__` um, der in jeder Klasse implementiert werden kann. Der Rückgabewert sollte ein `int`-Wert sein, der sich als Anzahl von Elementen interpretieren lässt. In unserem Fall ist also eine sinnvolle Implementierung:

Beispiel: `__len__`

```
149       def __len__(self) :
150         return self.shape[0] * self.shape[1]
```

7.3.9. Absolutbetrag: `__abs__`

Die Funktion `abs` berechnet den *Absolutbetrag* eines Objekts, indem sie das Ergebnis des Dunders `__abs__` weiterleitet. Was der Absolutbetrag einer Klasseninstanz ist, hängt sehr stark vom Kontext ab (und oft genug ist das Konzept überhaupt nicht anwendbar). Im Fall von Matrizen kann Beispielsweise die *Maximumsnorm* berechnet werden, also das größte Element der Matrix selbst:

Beispiel: `__abs__`

```
151       def __abs__(self) :
152         return max( [max(line) for line in self.data] )
```

7.4. Vererbung

In größeren Projekten werden wir in die Situation kommen, dass sich Klassen stark ähneln, aber nicht in allen Details gleich sind. In dem Beispiel der Exfreundinnen: nur von manchen kann ich mich noch an die Augenfarbe erinnern. Wir wissen, dass wir zu jeder Instanz dynamisch Attribute hinzufügen können. Dies ändert natürlich nichts daran, wie die Methoden ausgeführt werden (und soll auch nichts daran ändern). Jedoch wäre es schön, wenn die Methode `__str__` diese zusätzliche Information Augenfarbe mit ausgegeben würde.

Natürlich können wir eine zweite Klasse `ExpartnerWithEyeColor` anlegen, und allen Code der Klasse `Expartner` kopieren. Dies hätte aber auch den Effekt, dass Weiterentwicklungen an der Klasse `Expartner` doppelt gemacht werden müssen, also auch in `ExpartnerWithEyeColor` manuell eingepflegt werden müssen.

Die Technik *Vererbung* erlaubt uns, dies zu umgehen oder zu automatisieren. Von einer *Basisklasse* wird eine Kopie angelegt, die wir *abgeleitete Klasse* nennen. Diese Klasse hat zunächst alle Eigenschaften (d. h. Attribute und Methoden) der Basisklasse. Definitionen innerhalb der abgeleiteten Klasse erweitern diese gegenüber der Basisklasse, oder überschreibt die Eigenschaften, die von der Basisklasse schon gegeben waren.

Wir erreichen dies mit einem neuen Syntaxelement bei der Definition einer Klasse:

Syntax: Abgeleitete Klassen

```
class Abgeleitete_Klasse(Basisklasse)  :
    neues_Attribut = Standardwert
    ...
    def neue_Methode(self, ...) :
        ...
```

In unserem Beispiel wollten wir der abgeleiteten Klasse `ExpartnerWithEyeColor` ein Attribut `eyeColor` hinzufügen, und die Methode `__str__` überarbeiten. Wir erreichen dies mit den folgenden Zeilen:

Beispiel: Abeleitete Klasse

```
1  # Klasse Expartner wie oben
2
3  class ExpartnerWithEyeColor(Expartner) :
4      eyeColor = None
5
6      def __str__(self) :
7          reVal  = self.name + "\n"
8          reVal += "  Height      : " + str(self.height)          + "\n"
9          reVal += "  Upsides     : " + str(self.upsides)  [1:-1] + "\n"
10         reVal += "  Downsides   : " + str(self.downsides)[1:-1] + "\n"
11         reVal += "  Total Rating: " + str(self.rating())        + "\n"
12         reVal += "  Eye Color   : " + str(self.eyeColor)
13         return reVal
14
15 ex = ExpartnerWithEyeColor("Reginald", 1.84,
16                           {"very intelligent", "handsome", "plays in a band"},
17                           {"flirty with everyone", "complains a lot"})
18 ex.eyeColor = "blue"
19 print(ex)
```

Ausgabe: Abeleitete Klasse

```
Reginald
  Height      : 1.84
  Upsides     : 'very intelligent', 'handsome', 'plays in a band'
  Downsides   : 'flirty with everyone', 'complains a lot'
  Total Rating: 5
  Eye Color   : blue
```

Durch das `(Expartner)` in Zeile 3 werden alle Attribute und Methoden von `Expartner` kopiert. Wir können also auch neue Instanzen mit `ex = ExpartnerWithEyeColor(name, height)` anlegen, und haben die Methode `rating` zur Verfügung. Vergleiche zwischen Instanzen von `ExpartnerWithEyeColor` oder sogar zwischen Instanzen von `ExpartnerWithEyeColor` und `Expartner` sind direkt möglich (denn unsere Vergleichsmethoden prüfen nicht den Datentyp ab, sondern verlangen nur, dass eine Methode `rating` existiert). Die Methode `__str__` dagegen wurde überschrieben, und listet nun auch die Augenfarbe mit auf.

7.4.1. Methoden der Basisklasse – super()

Unsere Implementierung oben hat noch zwei kleine Schönheitsfehler:

- In der Methode `__init__` kann die Augenfarbe noch nicht mit angegeben werden.
- Wenn wir die Methode `__str__` von `Expartner` ändern, müssen wir immer noch manuell diese Änderungen auch in `ExpartnerWithEyeColor` vornehmen.

Beide Methoden – `__init__` und `__str__` – sind sich in `Expartner` und `ExpartnerWithEyeColor` sehr ähnlich, aber nicht exakt identisch. Es wäre angenehm, wenn wir die Methoden der Basisklasse zur Konstruktion der abgeleiteten Klasse benutzen könnten – und dies ist tatsächlich möglich!

In Klassenmethoden kann die Funktion `super()` aufgerufen werden. Diese gibt eine Instanz von `self` zurück, das aber als Instanz der Basisklasse interpretiert wird. Damit können wir unsere Klasse `ExpartnerWithEyeColor` auch schreiben als:

Beispiel: Abeleitete Klasse mit Methoden der Basisklasse

```
1  # Klasse Expartner wie oben
2
3  class ExpartnerWithEyeColor(Expartner) :
4      eyeColor = None
5
6      def __init__(self, name = None, height = None, eyeColor = None
7                   upsides = {}, downsides = {}) :
8          super().__init__(name, height, upsides, downsides)
9          self.eyeColor = eyeColor
10
11     def __str__(self) :
12         return super().__str__() + "\n  Eye Color    : " + str(self.eyeColor)
13
14 ex = ExpartnerWithEyeColor("Reginald", 1.84, "blue"
15                           {"very intelligent", "handsome", "plays in a band"},
16                           {"flirty with everyone", "complains a lot"})
17 print(ex)
```

(Die Ausgabe bleibt hierbei unverändert.)

Betrachten wir zuerst Zeile 12: Hier behandeln wir also die aktuelle Klasseninstanz `self` als Instanz der *Basisklasse* `ExpartnerWithEyeColor` (durch das Element `super()`). Von dieser Instanz rufen wir die Methode `__str__` auf, und erhalten so bereits die ersten Zeilen unserer Zusammenfassung. An diese wird nun noch eine weitere Zeile angehängt, und ergänzt so die Textdarstellung von `ExpartnerWithEyeColor`.

Ähnlich lassen wir in Zeile 8 bereits alle Attribute ausfüllen, die schon in `Expartner` existiert haben. Insbesondere nutzen wir hier auch dieselbe Plausiblitätsprüfung: ExpartnerInnen mit negativer Körpergröße führen zu einer Fehlermeldung. Was danach noch übrig bleibt ist das Eintragen der Augenfarbe, was in Zeile 9 erledigt wird.

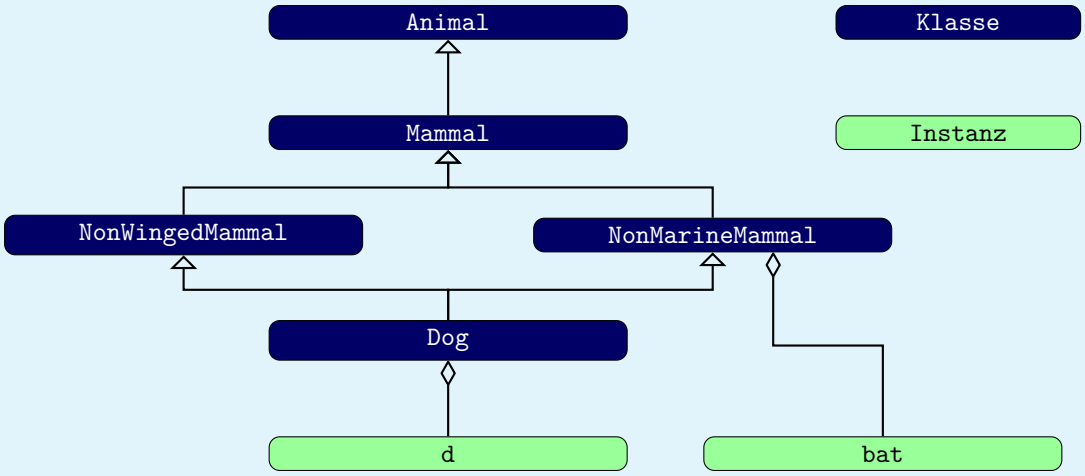
7.4.2. Mehrfach-Vererbung

Es ist auch möglich, dass eine Klasse von *mehreren* Basisklassen erbt. Das Ergebnis ist eine Klasse, die die *Summe* aller Methoden und Attribute der Basisklasse implementiert. Dabei darf es nicht zu Namenskollisionen kommen: Sowohl die Basisklasse `A` als auch die Basisklasse `B` eine Methode `M` besitzen, wird die abgeleitete Klasse `class X(A, B)` nur die Methode `M` von `A` übernehmen. Einzige

Ausnahme: Die Methode `__init__` wird von allen Basisklassen übernommen; `super().__init__()` ruft die Konstruktoren von *allen* Basisklassen in der Reihenfolge auf, in der sie bei der Definition der abgeleiteten Klasse genannt wurden.

Betrachten wir das folgende Klassendiagramm:

Klassendiagramm



(Quelle: `https://www.programiz.com/python-programming/methods/built-in/super`)

dies lässt sich durch den folgenden Code umsetzen:

Beispiel: Multi-Vererbung

```
1  class Animal :
2    def __init__(self, Animal) :
3      print(Animal, 'is an animal.')
4
5  class Mammal(Animal) :
6    def __init__(self, mammalName) :
7      print(mammalName,
8            'is a warm-blooded animal.')
9      super().__init__(mammalName)
10
11 class NonWingedMammal(Mammal) :
12   def __init__(self, NonWingedMammal) :
13     print(NonWingedMammal, "can't fly.")
14     super().__init__(NonWingedMammal)
15
16 class NonMarineMammal(Mammal) :
17   def __init__(self, NonMarineMammal) :
18     print(NonMarineMammal,
19           "can't swim.")
20     super().__init__(NonMarineMammal)
```

```
21  class Dog(NonMarineMammal,
22            NonWingedMammal) :
23    def __init__(self) :
24      print('Dog has 4 legs.')
25      super().__init__('Dog')
26
27  d = Dog()
28  print('')
29  bat = NonMarineMammal('Bat')
```

und erzeugt die Ausgabe:

Ausgabe: Multi-Vererbung

```
Dog has 4 legs.
Dog can't swim.
Dog can't fly.
Dog is a warm-blooded animal.
Dog is an animal.

Bat can't swim.
Bat is a warm-blooded animal.
Bat is an animal.
```

Klassenbeziehungen einfach halten

Die Technik *Vererbung* ist zweifellos sehr nützlich, kann aber schnell unerwartete Nebeneffekte ansammeln. Über mehrere Ebenen hinweg geht der Überblick leicht verloren. Versuchen Sie daher nach Möglichkeit, abgeleitete Klassen nicht als Basisklasse anderer Klassen zu benutzen.

Auch von Mehrfach-Vererbung wird im Allgemeinen abgeraten. Zu bevorzugen sind einfachere Code-Strukturen mit weniger verschränkten Abhängigkeiten.

8. Dateien

> The only superstition I have is that I must start a new book on the same day that I finish the last one, even if it's just a few notes in a file. I dread not having work in progress.
>
> Terry Pratchett

Ausgaben auf dem Bildschirm sind temporär – Daten auf der Festplatte sind für immer[1]. Hier werden wir uns der Aufgabe stellen, Daten vom Arbeitsspeicher auf dauerhafte Speichermedien[2] zu schreiben und von dort wieder in den Arbeitsspeicher zu lesen.

8.1. Binäre und Text-Dateien

Zuerst wollen wir aber einige Gedanken zur internen Darstellung von Daten aufwenden.

Wie Sie wissen, sind im Speicher alle Daten *binär* abgelegt, d. h. als Pattern von Einsen und Nullen. Je nach Kontext werden diese Pattern auf unterschiedliche Weise interpretiert.

Eine Folge von Bits b_i kann zum Beispiel nach dieser Formel als Ganzzahl[3] n interpretiert werden:

$$n = \sum_i 2^n \, b_i$$

Dasselbe Pattern kann aber auch für ein Schriftzeichen stehen. In diesem Fall wird mittels einer Tabelle übersetzt. Jedes Schriftzeichen hat eine Nummer, die leicht als Binärzahl darstellbar ist und in dieser Form das Zeichen im Arbeitsspeicher repräsentiert. Die ersten 128 Schriftzeichen sind in Abbildung 8.1 gezeigt[4].

Wenn Sie sich diese Tabelle durchsehen, wird Ihnen auffallen, dass darin auch wieder Ziffern vorkommen. Das bedeutet, dass eine Zahl sowohl durch ihr Bitmuster dargestellt werden kann, als auch durch die Bitmuster ihrer Textdarstellung. Die Zahl `42` hat etwa das Bitmuster `00101010`. Als *Text* `"42"` finden wir dagegen die Schriftzeichen `"4"` und `"2"` mit ASCII-Codes 52 und 50 und daher der das Bitmuster `00110110 00110100`. Genauso kann auch das Bitmuster `00101010` als das *Schriftzeichen* mit der Nummer 42 gelesen werden, also als `"*"`.

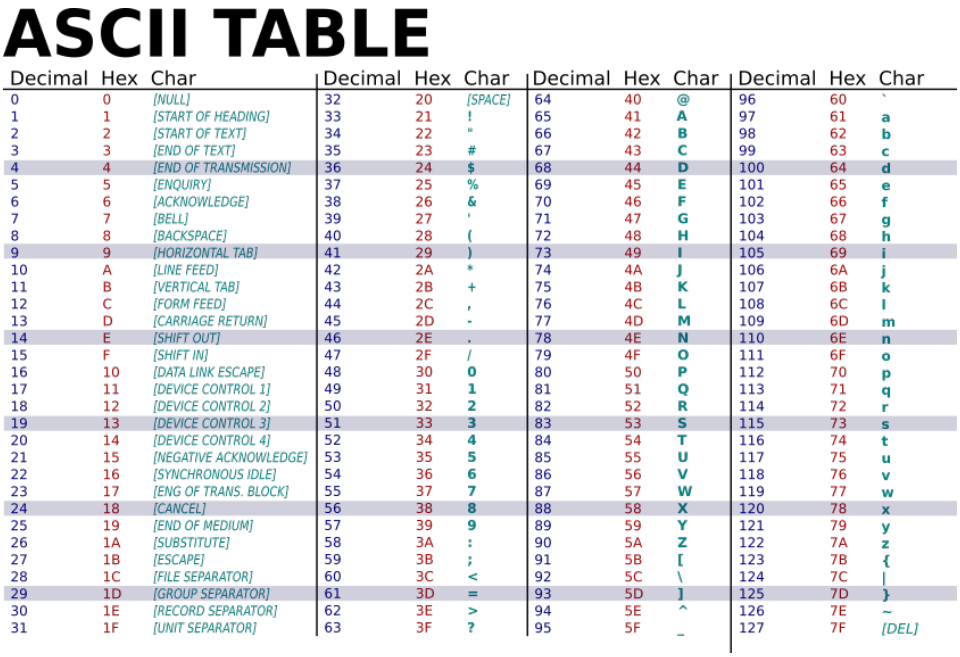
Bisher mussten wir uns mit der Frage der Interpretation von Daten kaum beschäftigen, da jeder Ausdruck in Python einen zugeordneten *Datentypen* hat; die Regeln zur Interpretation werden also „kostenlos“

[1] Eigentlich nur für die nächsten Jahre, abhängig vom Speichermedium. In jedem Fall aber lang genug...

[2] In der Regel auf die Festplatte

[3] Eine ähnliche, jedoch schwerer zu verstehende Formel existiert auch für Fließkommazahlen. Interessierte KursteilnehmerInnen können sich über die Norm IEEE 754 informieren (siehe hierzu beispielsweise `https://de.wikipedia.org/wiki/IEEE_754`). Für hier reicht es vollkommen, zu verstehen, dass Kommazahlen und Ganzzahlen nach unterschiedlichen Regeln in den Speicher geschrieben werden

[4] ASCII – American Standard Code for Information Interchange – stellt einen der ältesten Standards dar, der in der Westlichen Welt lange Zeit genutzt wurde. Darauf aufbauend existieren ANSI, Unicode in diversen Implementationen, ... – *It's a mess.* Allen diesen Standards ist gemeinsam, dass einem Schriftzeichen ein *Codepoint* zugeordnet ist, dass es also mit einer Ganzzahl identifiziert wird.



ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Abbildung 8.1.: ASCII-Tabelle: Lookup-Tabelle zur Interpretation von Zahlen als Schriftzeichen
Quelle: `https://en.wikipedia.org/wiki/File:ASCII-Table-wide.svg`

mitgeliefert. In Dateien dagegen können wir dagegen nur rohe Daten speichern. Als ProgrammiererInnen müssen wir uns also überlegen, wie und von wem diese Daten interpretiert werden sollen.

Wir unterscheiden im Wesentlichen zwischen *Binärdaten* und *Textdaten*. Binärdaten werden so auf die Festplatte geschrieben, wie sie auch im Speicher vorliegen, und auch so von dort gelesen. Man könnte sagen, Binärdaten seien die Muttersprache der Computer. Textdaten dagegen werden zuerst so übersetzt, dass sie ein menschlicher Leser leicht übersetzen kann.

In einem Beispiel: Sie haben im Speicher die `int`-Zahl `42`. Im Binärmodus wird daher das zugehörige Bitmuster `00101010` auf die Festplatte geschrieben. Öffnen wir diese Datei mit einem *Text*editor, sehen wir das Zeichen `*`.
Wird dieselbe Zahl `42` dagegen im Textmodus geschrieben, so übersetzt Python diese zuerst in den *Text* `"42"`, und schreibt dann diesen auf die Festplatte. Wir finden also das Bitmuster `00110110 00110100`. In einem Texteditor lesen wir auch wieder `42`.

Python erledigt all dieses Übersetzen für uns im Hintergrund. Wir als ProgrammiererInnen müssen aber zumindest vorgeben, ob eine Datei im Binärmodus oder im Textmodus gelesen/geschrieben werden soll.

8.2. Einfacher Dateizugriff

Python stellt „einfache Hausmittel“ zur Arbeit mit Dateien zur Verfügung. Darauf aufbauend existieren weitere Module, die uns die Arbeit erleichtern, für die wir aber auch die „Hausmittel“ im Prinzip verstanden haben müssen.

8.2.1. Dateien Öffnen und Schließen

Um mit Dateien umzugehen, brauchen wir zunächst ein *Handle auf die Datei*. Das ist eine Variable, in der alle nötigen Hintergrund-Informationen (Speicherort, Dateigröße, Netzwerkdatei oder lokale Datei, ...) zusammengefasst sind. Es handelt sich also um eine Klasseninstanz, deren Eigenschaften wir hier oberflächlich kennen lernen wollen. Stellen Sie sich das Handle tatsächlich als einen „Griff“ vor, der an eine Datei angebracht wird: mit diesem Griff können Sie die Datei anpacken und darin Operationen (Lesen, Schreiben, Analysieren) durchführen.

Wir erhalten ein solches Handle über den Befehl `open`:

Syntax: `open`

```
handle = open(Dateiname, Modus)
```

Wie zu erwarten ist `Dateiname` ein String, der den Namen der Datei enthält, die geöffnet werden soll. Je nach Betriebssystem wird Groß- und Kleinschreibung unterschieden[5]. Die Datei wird im *aktuellen Arbeitsverzeichnis* erwartet, d. h. in dem Ordner, von dem aus Python auch gestartet wurde. Üblicherweise ist das derselbe Pfad, in dem auch Ihre Script-Datei liegt. Sollen Dateien in anderen Ordnern betrachtet werden, können aber auch *absolute* und *relative* Pfadangaben gemacht werden:

Absolute Pfadangaben enthalten die volle Information, wo im Dateisystem eine Datei abgelegt ist. Sie beginnt mit einem Betriebssystem-spezifischen Symbol für die *Wurzel* des Dateisystems („root“), und durch *Forward Slashes /*[6] voneinander abgetrennte Ordnernamen. Unter Windows ist dieses Wurzel-Zeichen der *Laufwerks-Buchstabe* gefolgt von einem Doppelpunkt. Ein gültiger Dateiname mit absoluter Pfadangabe unter Windows könnte also lauten:

```
C:/User/some_folder/myFile.txt
```

Linux und Mac verwalten Laufwerke anders; die Wurzel des Dateisystems ist daher durch einen einfachen Forward Slash gekennzeichnet:

```
/home/user/some_folder/myFile.txt
```

Relative Pfadangaben gehen vom aktuellen Arbeitsverzeichnis aus. Von diesem Startpunkt aus wird die Orderstruktur navigiert. Dass eine Pfadangabe relativ sein soll, wird erklärt, indem man als erstes Zeichen einen Punkt angibt.

Beispiel: Ihre Code-Datei liegt unter `/home/user/Codes/`. Die Datei, die Sie öffnen wollen, befindet sich in `/home/user/Codes/files`. In dem Fall können Sie auf diese Datei zugreifen, indem Sie als `Dateiname` angeben:

```
./files/myFile.txt
```

Dies gilt so gleichermaßen für Windows als auch für Linux und Mac.

Ist die Datei in einem *Überordner*, so kann dies mit *zwei Punkten* angezeigt werden. Wir können auch mehrere ebenen zurück gehen, indem wir Punkte und Slashes aneinander reihen: `../../` bezeichnet den Ordner *zwei Ebenen* über dem Arbeitsverzeichnis. Weiter können wir diese Schreibweise auch mit relativen Pfadangaben kombinieren. Stellen wir uns wieder vor, das aktuelle Arbeitsverzeichnis wäre

[5] Unter Windows wird nicht unterschieden. Linux und Mac dagegen erkennen `file` und `File` als unterschiedliche Dateien an.

[6] Unter Windows sind auch *Backslashes* \erlaubt und z. T. noch üblich. Dies bereitet aber oft Probleme mit Escape-Sequenzen und sollte daher vermieden werden

`/home/user/Codes/`. Die Datei, die Sie öffnen wollen, efindet sich nun aber im Order `/home/user/files`. In diesem Fall geben Sie an:

```
../files/myFile.txt
```

Unter *Modus* verstehen wir, für welche Art Zugriff die Datei geöffnet werden soll. Wollen wir aus der Datei Lesen oder Schreiben? Soll der Zugriff im Text- oder Binärmodus stattfinden? Der Modus wird durch einen String beschrieben. Die wichtigsten Modi finden Sie in Tabelle 8.1:

String	Bedeutung	Kommentar
`r`	Lesen – Textmodus	Dateicursor am Anfang der Datei. Neuen Dateien werden *nicht* angelegt.
`rb`	Lesen – Binärmodus	Dateicursor am Anfang der Datei. Neuen Dateien werden *nicht* angelegt.
`w`	Schreiben – Textmodus	Dateicursor am Anfang der Datei. Neuen Dateien werden angelegt, alte Dateien überschrieben.
`wb`	Schreiben – Binärmodus	Dateicursor am Anfang der Datei. Neuen Dateien werden angelegt, alte Dateien überschrieben.
`x`	Schreiben – Textmodus	Dateicursor am Anfang der Datei. Neuen Dateien werden angelegt, alte Dateien *nicht* überschrieben.
`xb`	Schreiben – Binärmodus	Dateicursor am Anfang der Datei. Neuen Dateien werden angelegt, alte Dateien *nicht* überschrieben.
`a`	Anhängen – Textmodus	Dateicursor am *Ende* der Datei. Neuen Dateien werden angelegt, alte Dateien *nicht* überschrieben.
`ab`	Anhängen – Binärmodus	Dateicursor am *Ende* der Datei. Neuen Dateien werden angelegt, alte Dateien *nicht* überschrieben.

Tabelle 8.1.: Dateimodi in python3

Relative Pfadangaben bevorzugen

Absolute Pfadangaben sind von Natur aus unflexibel. Wenn Sie Ihr Programm an Kollegen verschicken, wird es vermutlich nicht mehr funktionieren. Bevorzugen Sie daher relative Pfadangaben, und konzipieren Sie nach Möglichkeit Ihre Programme so, dass notwendige Dateien in *Unterordnern* liegen.

Jede Datei, die geöffnet wurde, sollte auch wieder geschlossen werden. Dies geschieht mit der Methode `close`:

Syntax: `open`

```
handle.close()
```

Ob eine Datei noch geöffnet ist, können wir mit dem Attribut `closed` in Erfahrung bringen: `handle.closed` ist entweder `True` oder `False`.

8.2.2. Dateien Schreiben

Textmodus

Sobald eine Datei in einem Text-Schreibmodus geöffnet wurde (`w`, `x` oder `a`), und solange sie noch nicht wieder geschlossen wurde, können wir mit der Methode `write` Text in eine Datei schreiben. Die Syntax ist denkbar einfach:

Syntax: `write`

```
handle.write(Text)
```

Dabei ist `Text` ein beliebiger Ausdruck, der zu einem String ausgewertet werden kann.

Beispiel: Text-Dateien Schreiben

```
1  handle = open("output.txt", "w")     # Text/Schreibmodus
2
3  handle.write("Some text\n")
4  handle.write("another line " * 2 + "\n")
5  handle.write( str(handle) )
6
7  handle.close()
```

Dateinhalt von `output.txt`

```
Some text
another line another line
<_io.TextIOWrapper name='output.txt' mode='w' encoding='UTF-8'>
```

Beachten Sie, dass, anders als bei `print` Zeilenumbrüche *nicht* automatisch angehängt werden. Diese müssen explizit als `"\n "` teil des zu schreibenden Strings sein.

Im obigen Beispiel wird der *Rückgabewert* der Methode ignoriert. `write` gibt die Zahl der in die Datei geschriebenen Bytes zurück. Dieses Feature wird selten gebraucht, kann aber unter Umständen nützlich sein.

Öffnen wir nach Zeile dieselbe Datei `output.txt` nochmals in den Modi `w`, `wb`, `x` oder `xb`, so wird der Dateiinhalt sofort gelöscht. Bei den Modi `a` und `ab` dagegen bleibt der alte Inhalt bestehen; neu geschriebener Text wird an das Ende der Datei angehängt.

Binärmodus

Schreiben im Binärmodus funktioniert prinzipiell genauso, wie wir es vom Textmodus schon kennen: Öffnen mit `open` in einem geeigneten Modus (`wb`, `xb` oder `ab`), Schreiben mit `handle.write(data)` und Schließen mit `handle.close()`. An das Objekt `data` muss *eindeutig* in eine Byte-Sequenz übersetzt werden können. (Eingangs wurde gesagt, dass z. B. der `int`-Wert `42` in das Bitpattern `00101010` übersetzt wird; gleichermaßen ist aber auch das Bitpattern `00000000 00101010` denkbar, und eine beliebige andere Zahl führender Nullen. Die Gründe hierfür sind leider etwas technisch, und können an dieser Stelle nicht erschöpfend erklärt werden).

Eine Klasse von Objekten, das eine solche absolut unmissverständliche Darstellung haben, sind Instanzen der Klasse `bytes`. Diese können beispielsweise aus `list`s erstellt werden, wenn deren einzelne Elemente nur Werte zwischen `0` und `255` annehmen.

Beispiel: Binär-Dateien Schreiben

```
handle = open("output.dat", "wb")     # Binär/Schreibmodus

data = bytes([65, 66, 67])
handle.write( data )

handle.close()
```

Dateinhalt von `output.dat`

```
ABC
```

8.2.3. Dateien Lesen

Das Lesen aus Dateien gestaltet sich ähnlich einfach wie das Schreiben: nachdem die Datei in einem geeigneten Modus geöffnet wurde (`r` oder `rb`) kann mit den Methoden `read` und `readline` gearbeitet werden.

Beim Lesen können wir uns vorstellen, dass in der Datei ein „Cursor“ platziert wird. Zu Beginn ist dieser ganz am Anfang der Datei. Mit jedem Lese-Befehl wandert dieser Cursor dann vorwärts. Gelesen wird immer jeweils von der aktuellen Cursorposition.

Die Methode `read` im einfachsten Fall *die gesamte Datei* in den Arbeitsspeicher:

Syntax: `read`

```
StringVariable = handle.read()
```

Optional kann auch ein `int`-Parameter `size` mit übergeben werden. Dieser gibt eine Maximalmenge an Daten an, die aus der geöffneten Datei `handle` gelesen werden dürfen.

Syntax: `read` (erweitert)

```
StringVariable = handle.read(size)
```

Dabei bezeichnet `size` die Zahl der Bytes, die maximal gelesen werden dürfen.

Die online-Dokumentation (`https://docs.python.org/3/tutorial/inputoutput.html`) kommentiert hierzu lakonisch:

> *It's your problem if the file is twice as large as your machine's memory.*

Beispiel: Wir wollen die im vorigen Abschnitt vorbereitete Datei `output.txt` zurück in den Arbeitsspeicher lesen. Dies erreichen wir mit dem folgenden Code:

Beispiel: Text-Dateien Lesen

```
handle = open("output.txt", "r")    # Text/Lesemodus

firstWord = handle.read(4)
rest      = handle.read()

handle.close()

print(firstWord)
print(rest)
```

Ausgabe: Text-Dateien Lesen

```
Some
 text
another line another line
<_io.TextIOWrapper name='output.txt' mode='w' encoding='UTF-8'>
```

Der Dateiinhalt wird als String in den Speicher geladen. Soll dieser wieder als Zahl behandelt werden, so muss der String mit den entsprechenden Funktionen (`int()`, `float()`, ...) umgewandelt werden.

Wird „nach dem Ende der Datei“ versucht, weiter zu lesen, so ist das Ergebnis einfach ein leerer String.

Textdateien sollen besonders häufig Zeile für Zeile bearbeitet werden. Zu diesem Zweck existiert die Methode `readline`:

Syntax: `readline`

```
StringVariable = handle.readline(size)
```

`readline` liest von der aktuellen Cursorposition bis zum nächsten Zeilenumbruch. Wieder begrenzt der Parameter `size` die Datenmenge, die maximal gelesen wird. Wird dieser ausgelassen oder wird hier ein negativer Wert übergeben, so liest Python *die gesamte Zeile*.

Verwandt mit der Methode `readline` ist die Methode `readlines`: Diese liest *die gesamte Datei*, zerlegt sie dabei aber bereits an den Zeilenumbrüchen, und packt die Teilde der Datei in eine `list`:

Beispiel: `readlines`

```
handle = open("output.txt", "r")    # Text/Lesemodus

lines = handle.readlines()

handle.close()

for line in lines :
    print(line)
```

Beachten Sie, dass die Zeilenumbrüche selbst beim Lesen nicht entfernt werden. Entsprechend sehen wir bei diesem Beispiel zusätzliche, leere Zeilen:

Ausgabe: `readlines`

```
Some text

another line another line

<_io.TextIOWrapper name='output.txt' mode='w' encoding='UTF-8'>
```

8.2.4. Dateien und Blockstrukturen

Im Kontext von Dateien können Sie das Schlüsselwort `with .. as` benutzen. Damit wird eine neue Umgebung (Einrückungsebene) geschaffen, innerhalb derer eine Datei geöffnet ist. Sehen Sie also `with` als eine Kurzform an, die automatisch eine geöffnete Datei schließt, wenn die Einrückungsebene von `with` geschlossen wird.

Syntax: `with .. as`

```
1  with open(Dateiname, Modus) as Handle :
2      Datei_Anweisungen
```

Das oben gezeigte *Beispiel: Text-Dateien Schreiben* lässt sich also auch so schreiben:

Beispiel: Text-Dateien Schreiben

```
1  with open("output.txt", "w") as handle :
2      handle.write("Some text\n")
3      handle.write("another line " * 2 + "\n")
4      handle.write( str(handle) )
```

Wie schon zuvor wird die Datei „`output`" im Schreibmodus geöffnet und mit drei Zeilen gefüllt. Mit dem Ende von Zeile 4 wird die Datei automatisch geschlossen.

`with`-Blocks für eigene Klassen: `__enter__` und `__exit__`

Wie Sie sich denken können, sind Handles nur Instanzen einer Klasse, nämlich `_io.TextIOWrapper`. Die gezeigten Methoden `write`, `read` und `close` sind im Prinzip nichts anderes, als die Methoden, die Sie im letzten Kapitel bereits kennengelernt haben.

Das Verhalten, das über `with` erreicht wird, realisiert der Python-Interpreter durch aufruf zweier Dunders erreicht:

Die Methode `__enter__(self)` wird mit Beginn des `with`-Blocks aufgerufen. Der Parameter `self` ist dabei gleich dem Objekt, das hinter `with` steht. In der Zeile
`with open("output.txt", "w") as handle`
erhält `self` also den Rückgabewert von `open`. Der Rückgabewert sollte das Objekt `self` einfach durchreichen.

Die Methode `__exit__(self, exc_type, exc_value, exc_traceback)` wird aufgerufen, sobald der `with`-Block verlassen werden soll. Dies kann entweder sein, weil sein Ende erreicht wurde, oder weil im Code des `with`-Blocks ein Fehler aufgetreten ist. Die Fehlerbeschreibung ist dann in den Variablen `exc_...` enthalten. Hierauf soll in Kapitel 9 näher eingegangen werden.

Die weit häufigste Aufgabe im Kontext von Dateien ist, diese Zeile für Zeile zu lesen und mit den geladenen Daten weitere Operationen zu bewerkstelligen (z. B. aufsummieren, plotten, ...). Zu diesem Zweck können `for`-Schleifen verwendet werden:

Syntax: Dateihandles mit `for`

```
for Zeile in handle :
    Anweisungen
```

Diese Syntax enthält also bereits die Mechanik, die Sie sonst mit `handle.readline` umsetzten würden. Hier wird auch automatisch detektiert, ob das Dateiende bereits erreicht wurde.

Wenn Sie tatsächlich den gesamten Dateiinhalt nach Zeilen aufgetrennt im Speicher brauchen, können Sie das Datei-Handle auch einfach in eine `list` umwandeln:

Beispiel: Dateihandles als `list`

```
with open("output.txt", "r") as handle :
    allLines = list(handle)
    print("Zeile 1:",  allLines[0])
```

Dies ist vollkommen gleichwertig zu `allLines = handle.readlines()`

8.2.5. Dateicursor Bewegen und Bestimmen

Manchmal (selten) ist es nötig, die aktuelle Cursorposition in der Datei zu kennen oder zu verändern. Dazu dienen die Methoden `tell` und `seek`.

`handle.tell()` gibt die aktuelle Position des Cursors ab Dateianfang zurück. Mit anderen Worten, der nächste `read`- oder `write`-Befehl wirkt auf die Bytes ab Position `handle.tell()`.

Mit `seek` kann der Dateicursor verschoben werden. Die hat einen verpflichtenden Parameter `offset` und einen optionalen Parameter `from`. Offset gibt an, um wie viele Bytes der Dateicursor gegen einen bestimmten Referenzpunkt verschoben werden soll. Welcher Referenzpunkt das ist, wird über den Parameter `from` angegeben:

- Ist `from == 0` oder wird `from` ausgelassen, so bezieht sich `offset` auf den Dateianfang. Mit anderen Worten, `offset` ist die neue Position des Dateicursors.
- Ist `from == 1`, so ist der Referenzpunkt die aktuelle Position. `handle.seek(-10, 1)` verschiebt also den Dateicursor um 10 Bytes *nach vorne*.
- Schließlich bedeutet `from == 2` einen Bezug auf das Dateiende. `handle.seek(0, 2)` verschiebt also den Dateicursor *an das Dateiende*.

Beispiel: Funktion zum Ermitteln der Länge einer Datei

```
def length_of_file(handle) :
    if handle.closed :
        raise Exception("Datei muss geöffnet sein!")

    oldPos = handle.tell()

    handle.seek(0, 2)         # an das Dateiende springen
    length = handle.tell()    # Cursorposition Dateiende = Länge!

    handle.seek(oldPos)       # Ausgangszustand wiederherstellen
    return length


with open("output.txt", "r") as handle :
    print(handle.name, "ist", length_of_file(handle), "Bytes lang.")
```

8.3. Pickle und JSON

Die Berechnungen, die Sie in Python umsetzen, können sehr Zeit- und Ressourcen-Aufwändig sein. Wenn dieselben Ausgangsdaten für mehrere Teilprojekte gebraucht werden können, bietet es sich an, diese einmalig auf der Festplatte zu speichern, anstatt sie in jedem Teilprojekt neu zu berechnen. In diesem Fall sind die Daten also nicht für einen menschlichen Betrachter vorgesehen, sondern sollten von einem Computer möglichst effizient geschrieben und wieder entpackt werden können. Die Module `pickle` (englisch: Gewürzgurke) und JSON (*JavaScript Object Notation* bieten hierzu einige Werkzeuge, die genau diese Aufgaben erledigen.

8.3.1. Pickle

Bei Pickle handelt es sich um ein Python-eigenes Format, das einige spezifische Eigenschaften der Sprache und seines Unterbaus ausnutzt, um sehr verschiedene Objekte bequem für die Archivierung in Dateien vorzubereiten. Leider kann hier nicht auf alle Details eingegangen werden; als Grundsatz können Sie sich jedoch merken:

Wenn Sie es in Python programmiert haben, können Sie es mit Pickle auf der Festplatte abspeichern.

Sicherheitslücke Pickle

Der oben genannte Satz gilt so weit, dass Sie mit Pickle sogar Programmroutinen abspeichern können. Dies erlaubt leider auch, schadhaften Code in Pickle-Archiven unterzubringen. Laden Sie daher nur Archive aus vertrauenswürdigen Quellen (z. B. selbst erstellte Archive oder Ergebnisse von ArbeitskollegInnen am selben Projekt). Vertrauen Sie hingegen niemals Archiven, die über das Internet bereitgestellt werden.

Das Archivieren mit Pickle läuft in drei Schritten ab:

- Laden des Moduls: `import pickle`
- Öffnen einer Datei im Binär-Schreibmodus: `handle = open(Dateiname, "wb")`

- Schreiben in eine Datei mit der Methode `dump`: `pickle.dump(Objekt, handle)`

In ähnlicher Manier kann mit `Objekt = pickle.load(handle)` ein Objekt aus der Archiv-Datei entpackt werden, und im Anschluss wie eine normale Variable weiter verwendet werden. Unter Objekt ist hierbei tatsächlich eine Variable beliebigen Typs zu verstehen (also z. B. `int`, `complex`, Klasseninstanzen, Funktionen, ...).

Beispiel: Zahl und Klassenobjekt in Datei zwischenspeichern und wieder laden

```
1  import pickle
2
3  class Foo :
4      def __init__(self) :
5          self.bar = 1 + 1j
6
7      def __str__(self) :
8          return "Foo object"
9
10 complexNumber = -0.3 + 4.1j
11 classObject   = Foo()
12
13 with open("archive.pkl", "wb") as handle :
14     pickle.dump(complexNumber, handle)
15     pickle.dump(classObject  , handle)
16
17 with open("archive.pkl", "rb") as handle :
18     readComplex     = pickle.load(handle)
19     readClassObject = pickle.load(handle)
20
21 print(readComplex)
22 print(readClassObject, readClassObject.bar)
```

Ausgabe: Zahl und Klassenobjekt in Datei zwischenspeichern und wieder laden

```
(-0.3+4.1j)
Foo object (1+1j)
```

In einer Datei können also beliebig viele Objekte gespeichert werden. Diese Objekte können dann in derselben Reihenfolge zurück gelesen werden, in der sie auch geschrieben wurden.

Der Inhalt der Datei `archive.pkl` ist nicht für menschliche Betrachter vorgesehen. Je nach Programm, das Sie zum Lesen benutzen, werden Sie eine Kette für Sie sinnloser Zeichen sehen oder auch nur eine Fehlermeldung erhalten.

Dateinamen und Module

Sie haben den Befehl `import` bereits kennengelernt, um Programmkomponenten zu laden, die Sie dann im Anschluss benutzen können. Was dabei geladen wird, ist in der Regel normaler Python-Code. Die Zeile `import pickle` veranlasst also den Interpreter dazu, die Datei `pickle.py` zu suchen und die darin definierten Klassen und Methoden bereitzustellen. Diese Datei `pickle.py` wird dabei *zuerst* im aktuellen Arbeitsverzeichnis gesucht, und dann in einer Reihe von Standard-Ordnern, die bei der Installation von Python festgelegt wurden.

Wenn Sie das obige Beispiel *Beispiel: Zahl und Klassenobjekt in Datei zwischenspeichern und wieder laden* als `pickle.py` abspeichern, werden Sie daher eine Fehlermeldung erhalten. Achten Sie also darauf, ihre Code-Dateien nicht nach den Modulen zu benennen, die Sie benutzen wollen.

8.3.2. JSON

JSON bezeichnet eine Syntax, mit der verschiedene Datenstrukturen als Text in Dateien abgelegt werden können. Der Fokus liegt hierbei auf Portierbarkeit: JSON-Dateien sollen von möglichst für Programme in möglichst vielen Sprachen verwertbar sein. Dies schränkt die Verwendbarkeit in Python leider leicht ein: nicht jedes beliebige Objekt kann direkt mit JSON archiviert werden. Stattdessen ist es oft nötig, komplexe Datenobjekte (Klasseninstanzen) in kleinere Objekte zu zerlegen (`list`s, `dict`s, ...), und diese dann zu schreiben. (Für fortgeschrittene ProgrammiererInnen besteht auch die Möglichkeit, JSON um eigene Encoder/Decoder zu erweitern und so auch komplexere Objekte ohne weitere Vorarbeit in Dateien abzulegen. Darauf kann hier leider nicht eingegangen werden.)

JSON-Dateien sind Text-Dateien, können also von einem menschlichen Betrachter gelesen und verstanden werden. Darin schadhaften Code unterzubringen ist bedeutend schwieriger (wenn auch nicht ganz unmöglich). Das JSON-Format ist also die richtige Wahl für Sie, wenn Sie in Python Daten zu Zwischenergebnissen verarbeiten, die von anderen Programmen gelesen werden sollen, welche nicht in Python geschrieben sind, oder wenn das Lesen der Ergebnisse für einen Menschen ebenfalls möglich sein soll.

Die folgenden Python-Datentypen können in JSON direkt codiert werden:

- `int`
- `float`
- `str`
- `bool`
- `list`
- `tuple`
- `dict`
- `None`

Diese Datentypen können auch ineinander verschachtelt sein, d. h. man kann auch ein `dict` mit mehreren `list`s ablegen, die ihrerseits wieder komplexere Datenstrukturen aus der oben genannten Liste enthalten.

Der Datentyp `complex` dagegen wird *nicht* unterstützt. Komplexe Zahlen müssen also entweder in Real- und Imaginärteil zerlegt werden und so als zwei `float`s gespeichert oder zu einem `str` umgewandelt werden.

Ähnlich wie bei pickle sind für das Archivieren mit JSON drei Schritte notwendig:

- Laden des Moduls: `import json`
- Öffnen einer Datei im Text-Schreibmodus: `handle = open(Dateiname, "w")`
- Schreiben in eine Datei mit der Methode `dump`: `json.dump(Objekt, handle)`

Zum Lesen ihres Archivs öffnen Sie die Datei im Text-Lesemodus (`"r"`), und benutzen die Methode `json.load`.

Damit die Datei später auch wieder gelesen werden kann, darf nur ein einziges Objekt geschrieben werden. Wollen Sie mehrere Objekte in derselben Datei ablegen, so können Sie diese zu einem `dict` zusammenfassen.

Beispiel: Archive schreiben und lesen mit JSON

```
1  import json
2
3  complexNumber = -0.3 + 4.1j
4
5  realPart      = complexNumber.real
6  imaginaryPart = complexNumber.imag
7
8  dictionary  = {1 : "one", 2 : "two", 3 : "three"}
9
10 with open("archive.json", "w") as handle :
11     json.dump(
12         {"real" : realPart,
13          "imag" : imaginaryPart,
14          "dict" : dictionary},
15         handle
16     )
17
18 with open("archive.json", "r") as handle :
19     jsonObjects = json.load(handle)
20
21
22 readComplex = complex(jsonObjects["real"], jsonObjects["imag"])
23
24 print(readComplex)
25 print(jsonObjects["dict"])*
```

Ausgabe: Archive schreiben und lesen mit JSON

```
(-0.3+4.1j)
{'1': 'one', '2': 'two', '3': 'three'}
```

Dateiinhalt: `archive.json`

```
{"real": -0.3, "imag": 4.1, "dict": {"1": "one", "2": "two", "3": "three"}}
```

Unter `https://www.w3schools.com/python/python_json.asp` finden Sie noch einige weitere Erläuterungen sowie zusätzliche Features, die hier nicht besprochen werden sollen.

8.4. CSV-Dateien

CSV steht für *comma separated values* und ist ein Datenformat, das sich besonders für Tabellen eignet. Wie es der Name suggeriert, werden die einzelnen Spalten der Tabelle durch Trennzeichen voneinander

abgesetzt. In der Regel sind diese Trennzeichen Kommata; jedes beliebige andere Zeichen kann jedoch auch als Trennzeichen verwendet werden. CSV-Dateien werden von vielen Programmen als Ausgabeformat unterstützt; unter anderem erlauben es Tabellenkalkulationsprogramme wie Microsoft Excel, Daten im CSV-Format auszugeben.

Beispiel CSV-Dateien

Die folgende Tabelle:

Name	Superkraft	Codenummer
Victoria	hat einen Eigenwert	80085
Lea	verwandelt sich in ein Chamäleon	1337
Martina	ist eigentlich Legolas	8453

kann als CSV-Datei abgebildet werden. Das Datei-Abbild könnte dann folgendermaßen aussehen:

Datei: `superheroines.txt`

```
"Name","Superkraft","Codenummer"
"Victoria","hat einen Eigenwert",80085
"Lea","verwandelt sich in ein Chamäleon",1337
"Martina","ist eigentlich Legolas",8453
```

Wie Sie sehen, werden Texte auch in Anführungszeichen eingerahmt; Zahlen bleiben ohne Sonderzeichen. Während dies eine häufige Wahl bei CSV-Dateien ist, können auch andere Formatierungen gewählt werden.
Im weiteren wollen wir davon ausgehen, dass die gezeigte Datei unter dem Dateinamen `superheroines.txt` im aktuellen Arbeitsverzeichnis liegt.

Das Python-Modul `csv` erlaubt es, genau solche Date-n strukturiert einzuladen. Zentrales Element ist die Klasse `csv.reader`, die auf einen bestimmten *Dialekt* von CSV-Formaten eingestellt werden kann (z. B. Trennzeichen zwischen Spalten, Anführungszeichen bei Texten, etc.), und die anhand dieser Einstellungen eine Datei in den Arbeitsspeicher liest und in Spalten zerteilt.

Um eine Instanz der Klasse `csv.reader` zu erhalten, wird ihr Konstruktor aufgerufen. Diesem muss ein Dateihandle im Text-Lesemodus mitgegeben werden. Das Trennzeichen für Spalten über das optionale Schlüsselwort `delimiter` festgelegt werden:

Beispiel: Datei mit CSV-Reader lesen

```
1  import csv
2  with open("superheroines.txt", "r") as file :
3      rdr = csv.reader(file, delimiter=",")
```

Dieser Reader kann nun benutzt werden, um die Datei Zeilenweise einzulesen. Am leichtesten geht dies über eine `for`-Schleife:

Beispiel: Datei mit CSV-Reader lesen (Fortsetzung)

```
4      for line in rdr :
5          print(line)
```

Ausgabe: Datei mit CSV-Reader lesen

```
['Name', 'Superkraft', 'Codenummer']
['Victoria', 'hat einen Eigenwert', '80085']
['Lea', 'verwandelt sich in ein Chamäleon', '1337']
['Martina', 'ist eigentlich Legolas', '8453']
```

Die Datei wird also Zeile für Zeile eingelesen und in der `list line` gespeichert. Alle Werte in der zweiten Spalte sind somit z. B. über `line[1]` zugänglich. Wie Sie auch sehen, werden *alle* Werte als Strings eingelesen – selbst die, die als Zahl in der CSV-Datei gespeichert waren.

Oft – wie auch im gezeigten Beispiel – enthält die erste Zeile einer Datei die Spaltenüberschriften. Um eine einzelne Zeile einzulesen, ohne über die gesamte Datei zu iterieren, kann das Schlüsselwort `next` benutzt werden. `next` erwartet ein *Iterable*, d. h. eine Variable, die über die mit `for` iteriert werden kann. Mit `next(Iterable)` wird ein einzelner `for`-Durchlauf ausgeführt. Der Wert aus `Iterable`, der hierbei normal erhalten würde, ist nun der Rückgabewert von `next`. In Kapitel **??** wird dies noch genauer besprochen.

Mit `next` kann die Datei auch wie folgt ausgegeben werden:

Beispiel: Datei Spaltenweise ausgeben

```
1  import csv
2  with open("superheroines.txt", "r") as file :
3      rdr = csv.reader(file, delimiter=",")
4      heads = next(rdr)
5
6      for line in rdr :
7          for i in range(len(heads)) :
8              print(heads[i] + ":", line[i])
9          print()
```

Ausgabe: Datei Spaltenweise ausgeben

```
Name: Victoria
Superkraft: hat einen Eigenwert
Codenummer: 80085

Name: Lea
Superkraft: verwandelt sich in ein Chamäleon
Codenummer: 1337

Name: Martina
Superkraft: ist eigentlich Legolas
Codenummer: 8453
```

Für weitere Details siehe auch die offizielle Dokumentation des Moduls unter `https://docs.python.org/3/library/csv.html`.

9. Exceptions

> With few exceptions, one ought always do what one is afraid of.
>
> Viggo Mortensen

Wie wir gesehen haben, bricht der Python-Interpreter die Ausführung unseres Codes ab, wenn sich die angeforderten Operationen als nicht Durchführbar herausstellen. Der Ausdruck `x / y` ist grundsätzlich ein sinnvolles Python-Codeelement, kann aber nicht ausgewertet werden, wenn `y` den Wert 0 hat. In diesem Fall wird eine sogenannte *Ausnahme* (*Exception*) ausgelöst. Unser Ziel als ProgrammiererIn sollte es sein, solche Ausnahmen zu vermeiden. Manchmal ist es aber bequemer, im Nachgang auf Fehler zu reagieren, als diese bereits im Vorfeld abzufangen. Dieses Kapitel soll Ihnen zeigen, wie Sie in Ihre Codes Fehlerbehandlung einbauen.

9.1. `try-except`-Blocks

9.1.1. Grundlegendes

Eine *Ausnahme* ist eine Instanz einer Klasse, die Informationen zum letzten Laufzeitfehler enthält. Tritt ein Fehler in der Codeausführung auf, so wird eine Instanz dieser Klasse angelegt, und durch verschiedene Routinen durchgereicht. Ultimativ wird sie in einem Kernmodul des Python-Interpreters landen, der auf dieses Signal hin die Ausführung unterbricht – es sei denn, wir schreiben Code, der diese Instanz abfängt und nicht mehr weiterreicht. Genau zu diesem Zweck existieren `try-except`-Blocks.

`try-except`-Blocks bestehen aus einem `try`-Block, in dem Anweisungen stehen, die potentiell einen Fehler erzeugen könnten, sowie (mindestens) einem `except`-Block, in den beschrieben wird, was geschehen soll, wenn dieser Fehler auftritt:

Syntax: `try-except`

```
try :
    Code der eine Ausnahme auslösen könnte
except ExceptionClass as varible :
    Code zur Fehlerbehandlung
```

Mit `ExceptionClass` ist dabei die „Art der Ausnahme“ gemeint, also die Fehlerklasse, die Sie sonst bei Programmabbruch angezeigt bekämen:

Auslösen einer Exception und Anzeige in der Python-Konsole

```
>>> print(1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Die letzte Zeile der Fehlermeldung zeichnet die aufgetretene Ausnahme also als `ZeroDivisionError` aus. Eben dies sollte auch anstelle von `ExceptionClass` in einer entsprechenden `except`-Zeile stehen.

`variable` ist eine neue Variable, mit der wir auf die Daten der Instanz von `ExceptionClass` zugreifen können.

Tritt während der Ausführung des `try`-Blocks eine Ausnahme der Klasse `ExceptionClass` auf, so wird der Code des entsprechenden `except`-Blocks ausgeführt; danach setzt Python die Ausführung mit *der ersten Zeile nach dem `try`-`except`-Block fort.*

Beispiel: Inverse

```
for x in range(-3, 3) :
    try :
        print(1/x)
    except ZeroDivisionError as e :
        print("An exception was caught:", e)
```

Ausgabe: Inverse

```
-0.3333333333333333
-0.5
-1.0
An exception was caught: division by zero
1.0
0.5
```

Beachten Sie auch, dass es durchaus von Bedeutung ist, welche Strukturen Sie wie ineinander schachteln: Vertauschen Sie im obigen Beispiel den `for`- mit dem `try`-Block, so werden die Inversen der positiven Zahlen nicht mehr berechnet:

Beispiel: Inverse II

```
try :
    for x in range(-3, 3) :
        print(1/x)
except ZeroDivisionError as e :
    print("An exception was caught:", e)
```

Ausgabe: Inverse II

```
-0.3333333333333333
-0.5
-1.0
An exception was caught: division by zero
```

Ist der aufgetretene Fehler nicht in der Klasse, die im `except`-Block genannt wurde, so ignoriert Python die bereitgestellte Fehlerbehandlung und bricht wie üblich die Fehlerbehandlung ab:

Beispiel: Fehlerklassen

```
1  import math
2  try :
3      print( math.log(-1) )
4  except ZeroDivisionError as e :
5      print("An exception was caught:", e)
```

Ausgabe: Fehlerklassen

```
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
ValueError: math domain error
```

Ausnahmen haben eine hierarchische Struktur, d. h. manche Ausnahmen sind Spezialisierungen von anderen. So ist beispielsweise `ZeroDivisionError` eine Abwandlung von `ArithmeticError`. In Beispiel: Inverse und Inverse II würden also genauso funktionieren, wenn Zeile 4 jeweils durch `except ArithmeticError as e :` ausgetauscht würde. Die allgemeinste (sinnvoll verwendbare) Fehlerklasse ist `Exception`. Im Kasten *Ausgewählte Fehlerklassen* finden Sie eine Aufzählung häufig ausgelöster Ausnahmen.

9.1.2. Mehrteilige `except`-Blöcke

Hinter `except` darf auch ein `tuple` von Fehlerklassen stehen, die dann jeweils mit demselben Code behandelt werden. Wir können also beispielsweise schreiben:

Syntax: `tuple` von Fehlerklassen

```
except (ZeroDivisionError, ValueError) as e :
```

Wenn ein Fehler auftritt, der auf wenigstens eine der Klassen im `tuple` passt (im Beispiel also entweder ein `ZeroDivisionError` oder ein `ValueError`), betritt der Python-Interpreter die `except`-Umgebung und setzt nach dieser die Ausführung am Ende des `try-except`-Blocks fort.

Daneben ist es auch möglich, mehrere `except`-Blöcke hintereinander zu setzen, und für jede Fehlerklasse einen eigenen Code zur Fehlerbehandlung zu schreiben. Dabei wird nur der *erste `except`-Block* abgearbeitet, der zur ausgelösten Ausnahme passt.

Beispiel: Mehrere `except`-Blöcke

```
1  try :
2      print(1/0)
3  except ArithmeticError as e :
4      print("Arithmetic Unit Error")
5  except ZeroDivisionError as e :
6      print("Division by Zero.")
```

Ausgabe: Mehrere `except`-Blöcke

```
Arithmetic Unit Error
```

Ausgewählte Fehlerklassen

- `Exception` (Generelle Fehlermeldung)
  - `StopIteration` (Wird von *iterierbaren* Objekten ausgelöst, um `for`-Schleifen mitzuteilen, dass sie fertig durchlaufen sind.)
  - `ArithmeticError` (Fehler bei Berechnungen)
    - `FloatingPointError` (Fehler spezifisch für Fließkommaberechnungen)
    - `OverflowError` (Ergebnis zu groß für den bereitgestellten Speicher)
    - `ZeroDivisionError` (Division durch Null)
  - `EOFError` (Lesezugriff nach Dateiende) (EOF: End of File)
  - `ImportError` (Fehler bei Bearbeitung einer `import`-Zeile)
    - `ModuleNotFoundError` (Angegebenes Modul wurde nicht gefunden)
  - `LookupError` (Fehler bei Zugriff auf ein Container-Element)
    - `IndexError` (Ungültiger Index)
    - `KeyError` (Ungültiger Schlüssel bei `dict`s)
  - `NameError` (Symbol mit diesem Namen existiert nicht)
  - `OSError` (Fehler bei Nutzung von Funktionen des Betriebssystems)
    - `FileExistsError` (Datei existiert bereits)
    - `FileNotFoundError` (Datei kann nicht gefunden werden)
  - `RuntimeError`
    - `NotImplementedError` (Methode/Variante einer Methode einer Klasse wurde nicht implementiert)
    - `RecursionError`
  - `SyntaxError`
    - `IndentationError`
  - `TypeError` (Datentyp kann nicht verarbeitet werden)
  - `ValueError` (Unzulässiger Wert bei Funktionsaufruf)

Siehe `https://docs.python.org/3/library/exceptions.html` für weitere Details.

Der Ausdruck `1/0` löst bekanntermaßen die Ausnahme `ZeroDivisionError` aus; diese ist jedoch eine Spezialisierung von `ArithmeticError`, welche im obigen Beispiel zuerst bearbeitet wird. Die Behandlung von `ZeroDivisionError` findet also unter keinen Umständen statt.

9.1.3. `else` und `finally`

Soll Code nur ausgeführt werden, wenn im `try`-Block *kein* Fehler aufgetreten ist, so kann dies mit einem `else`-Block erreicht werden. Dieser Block steht hinter den `except`-Blöcken und wird ausgeführt, nachdem die Behandlung des `try`-Blocks abgeschlossen wurde.

Das Schlüsselwort `finally` leitet einen weiteren Block ein, in dem Code steht, der ausgeführt wird, nachdem alle `except`- und `else`-Blöcke bearbeitet wurden, selbst wenn der tatsächlich eingetretene Fehler noch nicht abgefangen wurde:

Beispiel: Mehrere `try-except-else-finally`

```
1  def divide(x, y):
2      try:
3          result = x / y
4      except ZeroDivisionError:
5          print("division by zero!")
6      else:
7          print("result is", result)
8      finally:
9          print("executing finally clause")
10
11 print("divide(2, 1)")
12 divide(2, 1)
13 print()
14
15 print("divide(2, 0)")
16 divide(2, 0)
17 print()
18
19 print('divide("2", "0")')
20 divide("2", "1")
```

Ausgabe: Mehrere `except`-Blöcke

```
divide(2, 1)
result is 2.0
executing finally clause

divide(2, 0)
division by zero!
executing finally clause

divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

9.1.4. `raise`

In Kapitel 7 (Klassen) haben wir bereits gesehen, dass wir mit dem Befehl `raise` die Fehlerbehandlung auslösen können: Nach `raise` können wir eine Fehlerklasse (z. B. ValueError) nennen. Eine Instanz dieser Fehlerklasse wird dann angelegt und ruft das oben beschriebene Verhalten aus:

- Unbehandelt bricht der Python-Interpreter die Code-Ausführung ab und gibt eine Fehlermeldung aus
- In einem *passenden* `except`-Block kann der Fehler „aufgefangen“ und behandelt werden.

In der Regel erlauben die Fehlerklassen, ihrem Konstruktor noch ein String-Argument mitzugeben, in dem dann der aufgetretene Fehler in für Menschen lesbarer Form genauer beschrieben wird. In einem früheren Beispiel finden Sie etwa die Zeile

Beispiel: Auszug aus einem früheren Code

```
  raise Exception("Incompatible Shapes")
```

Diese Zeile löst also eine Ausnahme vom Typ `exception` aus. Die Zusatzinformation `"Incompatible Shapes"` wird der Ausnahme angehängt und taucht auf, wenn Informationen zum aufgetretenen Fehler ausgegeben werden sollen.

Fehlerklassen möglichst spezifisch wählen

Die Klasse `Exception` ist die allgemeinste Fehlerklasse. *Alle* auftretenden Fehler werden über `except Exceptoin as e :` abgefangen. Die Behandlung von verschiedenen Arten von Fehlern kann sehr verschiedenartig sein. In manchen Fällen wollen Sie vielleicht nur eine Warnung auf dem Bildschirm ausgeben, während andere Fehlerklassen eine Datei bearbeiten sollten oder ein extern angeschlossenes Gerät ausschalten müssen.

Damit Sie ihren Fehlerbehandlungs-Code richtig schreiben können und nicht in unerwarteten Situationen auslösen, sollten Sie auch geeignete, *spezifische* Fehlerklassen wählen. Vermeiden Sie daher `raise Exception` bzw. `except Exceptoin as e :` und benutzen Sie stattdessen lieber Fehlerklassen wie `ZeroDivisionError`.

Ausnahmen können auch nach der Behandlung „neu aufgelegt“ werden. Wollen Sie beispielsweise eine detaillierte Fehlerbeschreibung ausgeben, aber dennoch das Programmende auslösen, so erreichen Sie dies, indem Sie im `except`-Block erneut `raise` (ohne weitere Argumente) schreiben:

Beispiel: Re-raise

```
1  try :
2      x = 7
3      y = 0
4      print(x / y)
5  except ZeroDivisionError as e :
6      print("Division durch 0 aufgetreten!")
7      print("Fehlerbeschreibung: ", e)
8      print("x =", x)
9      print("y =", y)
10     raise
11
12 print("Wird nie ausgeführt.")
```

Ausgabe: Re-raise

```
Division durch 0 aufgetreten!
Fehlerbeschreibung:  division by zero
x = 7
y = 0
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ZeroDivisionError: division by zero
```

Wie Sie sehen, nennt die Ausgabe immer noch Zeile 4 als Punkt, an dem die Ausnahme ausgelöst wurde. Würden wir in Zeile 10 statt einem einfachen `raise` den Ausdruck `raise ZeroDivisionError`, so würde die Ausgabe auch Zeile 10 als Auslöser der Ausnahme melden.

Ein solches re-raise verlässt den gesamten `try`-Block. Auch nachgelagerte, passende `except`-Blocks werden ignoriert. Ein äußerer `try`-Block dagegen kann die re-raised Ausnahme dagegen immer noch auffangen:

Beispiel: Re-raise

```
1  try :
2      try :
3          print(1/0)
4      except ArithmeticError as e :
5          print("Arithmetic Unit Error")
6          raise   # leaves entire structure
7      except ZeroDivisionError as e :
8          print("Innerer Block wird übersprungen.")
9  except ZeroDivisionError as e :
10     print("Äußerer Block wird behandelt")
```

Ausgabe: Re-raise

```
Arithmetic Unit Error
Äußerer Block wird behandelt
```

Zeile 3 löst bekanntermaßen einen `ZeroDivisionError` aus. Dieser ist eine spezielle Form eines `ArithmeticError`s, und löst somit die Fehlerbehandlung in Zeile 4 aus (und damit die Ausgabe `Arithmetic Unit Error`). Das `raise` in Zeile 5 dann markiert den Fehler als noch nicht vollständig bearbeitet und verlässt den gesamten `try`-Block, der in Zeile 2 begonnen hat. Auch `except ZeroDivisionError` in Zeile 7 kommt damit nicht mehr in Frage.

Dagegen findet die Fehlerbehandlung jetzt eine Ebene höher statt, d. h. in dem `try`-Block, der in Zeile 1 begonnen hat. Damit wird also die Fehlerbehandlung in Zeile 9 ausgelöst, und wir finden eine zweite Meldung `Äußerer Block wird behandelt` auf dem Bildschirm.

9.2. Eigene Ausnahmen

Wir sollten Ausnahmen immer so spezifisch wie möglich aufwerfen. Die vorgefertigten Ausnahmen, die mit dem Standardumfang der Sprache Python angeboten werden, sind aber nicht besonders zahlreich und können unmöglich alle erdenklichen Szenarios abdecken, die wir als fehlerhaft behandeln wollen könnten. Wir müssen also auch eigene Fehlerklassen erstellen können.

Dies Geschieht nach dem Formalismus von von Klassen, wie wir ihn in Kapitel 7 bereits kennengelernt haben. Ausnahmen sind damit einfach von `Exception` abgeleitete Klassen.

Beispiel: Wir wollen fehlerhafte User-Eingaben als eigene Fehlerklasse abbilden. Dazu legen wir eine neue Klasse `UserInputError` an, die wir von `Exception` ableiten:

Beispiel: Eigene Fehlerklasse anlegen

```
1  class UserInputError(Exception) :
2      pass
3
4  try:
5      x = float(input("Bitte eine positive Zahl eingeben"))
6      if x < 0 :
7          raise UserInputError("Zahl war negativ!")
8  except UserInputError as e :
9      print(e)
```

Die Klasse `UserInputError` erbt alle Eigenschaften von `Exception` und ist damit effektiv eine Kopie von `Exception` mit neuem Namen. Wir müssen mindestens eine Code-Zeile in der Block-Anweisung `class` schreiben; daher setzen wir hier in `pass`, das keine weitere Auswirkung hat, aber die Anforderung an die Syntax erfüllt (keine leeren Block-Anweisungen).

Mit dieser neuen Klasse können wir jetzt also alle Techniken anwenden, die wir oben bereits besprochen hatten.

Natürlich können Sie Ihre Fehlerklassen auch weiter ausgestalten, und eigene Attribute und Methoden hinzufügen. Beim `raise` wird ein von Ihnen frei gestaltbarer Konstruktor aufgerufen (d. h. die Methode `__init__`), was es Ihnen erlaubt, zusätzliche Informationen an die Fehlerbehandlung zu übertragen. Als Minimalbeispiel reicht aber bereits der oben gezeigte Code.

Selbstverständlich können Sie auch ihre eigenen Fehlerklassen als Basisklassen verwenden. Das bedeutet, dass eine weitere Klasse `UserInputNumericError` von der bereits beschriebenen Klasse `UserInputError` erben könnte. Damit erzeugen Sie genau eine solche hierarchische Struktur, wie sie im Infokasten *Ausgewählte Fehlerklassen* bereits beschrieben ist.

10. Grafische Darstellung von Daten – Die MatPlotLib

> Real life seems to have no plots.
>
> Ivy Compton-Burnett

Menschen sind unheimlich gut darin, Bilder zu interpretieren. Computer haben ihre Stärke in der Auswertung von rohen Zahlen. Um nun die Früchte unserer Datenverarbeitung mit Python zu ernten, wollen wir Daten als graphische Plots ausgeben. Ein einfach zu bedienendes Mittel hierzu ist die MatPlotLib bzw. das Untermodul PyPlot. Das Modul MatPlotLib bietet tatsächlich so viele Funktionen, dass damit ein eigenständiger Kurs gefüllt werden könnte. Hier soll Ihnen eine Basis gezeigt werden, mit der Sie die häufigsten Aufgaben lösen können, und auf der Sie im Selbststudium leicht aufbauen können.

Eine erste Übersicht über die Funktionen der MatPlotLib finden Sie auch unter `https://matplotlib.org/tutorials/introductory/pyplot.html`.

10.1. Grundlagen

An dieser Stelle möchte ich Ihnen zuerst einen einfachen Code zeigen, und diesen dann Zeile für Zeile „entziffern“:

Beispiel: Einfacher Plot

```
1  import math
2  import matplotlib.pyplot as plt
3
4  N = 100
5  X = [(x - N/2) / 10 for x in range(N)]
6  Y = [math.sin(x) for x in X]
7
8  plt.plot(X, Y)
9  plt.show()
```

Ausgabe: Einfacher Plot

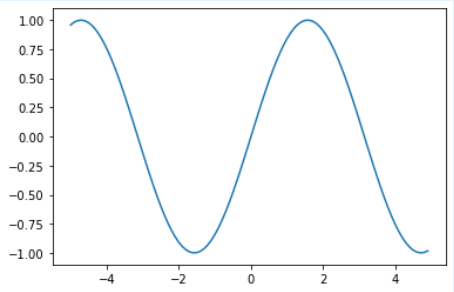


Abbildung 10.1.: Einfacher Plot der MatPlotLib

In den Zeilen 1 und 2 laden wir Module – zum einen das Modul `math`, mit dem wir die Daten generieren, die auf unserem Plot erscheinen sollen, und zum anderen die MatPlotLib. Tatsächlich handelt es sich dabei um ein extrem umfangreiches Paket, weswegen wir nur einen Teil davon in unser Projekt integrieren:

Das Untermodul `pyplot`[1]. Da dieser Modulname `matplotlib.pyplot` eher unhandlich ist, hat es sich eingebürgert, `plt` als Kurzname hierfür zu verwenden.

In den kommenden drei Zeilen generieren wir Werte, die schließlich geplottet werden sollen. `X` und `Y` sind jeweils Listen mit `N = 100` Elementen. Die Werte in `X` sind einfach gleichverteilte Werte im Abstand von `0.1`, rund um die `0` herum. Die Werte in `Y` enthalten jeweils den Sinus dieser `X`-Werte. Bis hierhin also haben wir noch nichts Neues gesehen.

In Zeile 8 wird nun die Funktion `plot` aus dem Modul `plt` aufgerufen. Diese Funktion bereitet alles vor, das nötig ist, um einen Plot zu generieren: Ein Arbeitsfenster, Achsen mit Beschriftung, Datenpunkte in den Graphen eintragen, ... All das wird aber nur im Arbeitsspeicher vorbereitet, jedoch noch nicht sichtbar gemacht. Grund hierfür ist, dass wir die Standard-Einstellungen noch abändern könnten: eine andere Linienfarbe, Skalierung, Bemaßung, ... Wenn jede Änderung in Echtzeit auf dem Bildschirm umgesetzt würde, hätte dies ein unangenehmes Flackern zur Folge, bevor der Plot endlich fertig aufgebaut ist. Stattdessen müssen wir manuell festlegen, wann unser Plot fertig beschrieben ist, d. h. wann er auf dem Bildschirm erscheinen soll. Dies geschieht in Zeile 10.

Im einfachsten Fall müssen wir also folgende Arbeitsschritte erledigen:

- Das Modul `matplotlib.pyplot` laden
- X- und Y-Werte als getrennte Listen vorbereiten
- Die Funktion `plot` aufrufen
- Die Funktion `show` aufrufen

Tatsächlich ist das Vorbereiten von X-Werten *streng genommen* sogar überflüssig:

Beispiel: Einfacher Plot ohne X-Werte

```
1  import matplotlib.pyplot as plt
2
3  N = 100
4  Y = [(x - 50) * x for x in range(N)]
5
6  plt.plot(Y)
7  plt.show()
```

Ausgabe: Einfacher Plot ohne X-Werte

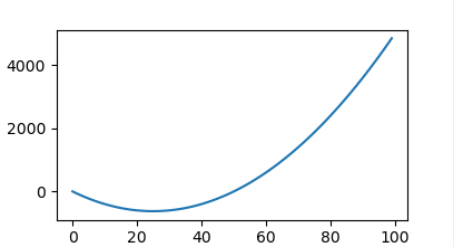


Abbildung 10.2.: Einfacher Plot ohne explizite x-Werte

Wenn keine X-Werte angegeben werden, erhalten wir zwar einen „sinnvollen“ Plot; Python hat jedoch natürlich keine Möglichkeit, zu entscheiden, welcher Wert an welche Position gehört. Daher geht der Plotter hier davon aus, dass die Werte bei `x=0` beginnen und jeweils einen Abstand von `1` zueinander haben.

Folgen zwei `plot()`-Befehle direkt aufeinander ohne ein `show()` dazwischen, so werden beide Graphen auf denselben Plot gezeichnet. Jeder Plot bekommt dabei seine eigene Farbe. Eine Legende wird jedoch nicht *automatisch* angezeigt.

[1] Die MatPlotLib enthält Code zum Fenstermanagement, zur Interpolation von Kurven, Umgang mit Dateien, ... Alle diese Features bilden den Unterbau von `pyplot`, müssen aber nicht „offengelegt“ werden, um für uns nützlich zu sein. Während `pyplot` intern alle diese Objekte und Funktionen benutzt und korrekt verwaltet, können wir uns auf das *Interface* konzentrieren, das uns `pyplot` auf all diese Features bietet.

Beispiel: Zwei Graphen im selben Plot

```
1  import matplotlib.pyplot as plt
2
3  N = 100
4  W = 10
5  X = [(x - N/2) / W for x in range(N)]
6  Y = [(x - W/2) * x for x in X]
7
8  plt.plot(X, Y)
9  plt.plot(X, [2 * y for y in Y])
10
11 plt.show()
```

Ausgabe: Zwei Graphen im selben Plot

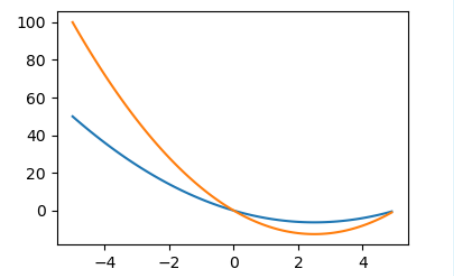


Abbildung 10.3.: Zwei Graphen

10.2. Einfache Formatierungen und andere Plot-Arten

Die folgenden Abschnitte stellen eine Auswahl an Themen dar; für eine volle Übersicht der Features von MatPlotLib-Plots siehe `https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.pyplot.plot.html`.

10.2.1. Format-Angaben im Befehl `plot`

Es wurde schon angesprochen, dass die Standard-Einstellungen der MatPlotLib nicht übernommen werden müssen. Im einfachsten Fall können wir direkt im `plot`-Befehl eine andere Linienfarbe und -form festlegen: Nach den Y-Werten kann ein optionaler String übergeben werden, in dem diese Information enthalten ist:

```
plt.plot(X, Y, "r--")
```

weist die MatPlotLib dazu an, den Plot in Rot mit Strichlinien zu zeichnen. Dagegen steht

```
plt.plot(X, Y, "b-.")
```

für einen Graphen aus blauen Punkten, die durch mit einer Linie verbunden sind. Folgende Zeichen werden verstanden und können auch miteinander kombiniert werden:

Format-Strings für `plt.plot()`

Punktarten

Symbol	Punkt	Symbol	Punkt	Symbol	Punkt
`,`	Pixel	`.`	kleiner Punkt	`o`	großer Punkt
`s`	Quadrat	`d`	schmale Raute	`D`	breite Raute
`p`	Fünfeck	`h`	Sechseck stehend	`H`	Sechseck liegend
`|`	Strich	`+`	Plus	`x`	Kreuz
`<`	Dreieck links	`>`	Dreieck rechts	`*`	Stern
`v`	Dreieck unten	`^`	Dreieck oben		

Linienarten

Symbol	Linie	Symbol	Linie	Symbol	Linie	Symbol	Linie
`-`	durchgezogen	`--`	gestrichelt	`:`	gepunktet	`-.`	strichpunkt

Farben

Symbol	Farbe	Symbol	Farbe	Symbol	Farbe	Symbol	Farbe
`b`	blau	`c`	cyan	`g`	grün	`k`	schwarz
`m`	magenta	`r`	rot	`y`	gelb	`w`	weiß

Tabelle 10.1.: Formatstring-Elemente für `plot`

Soll eine Kurve in einer Farbe gezeichnet werden, die nicht in Tabelle 10.1 aufgeführt sind, kann das Keyword-Argument `color` verwendet werden. Hier gibt man die Farbe als RGBA-String mit führendem Raute-Symbol an. Das bedeutet, dass der Rot- Grün- und Blau-Anteil der Farbe sowie die Deckkraft („Alpha-Wert“) als zweistellige Hexadezimalzahl geschrieben wird und diese vier Zahlen dann aneinander gereiht werden. Für ein dunkles Rot kann man also schreiben:

```
plt.plot(X, Y, color="#7F0000FF")
```

Dabei ist `7F` der Rot-Anteil (entspricht 50% Intensität), `00` jeweils der Grün- und Blau-Anteil und `FF` die Deckkraft (entspricht 100%).

10.2.2. Legenden Anzeigen und Gitter anzeigen – `legend`, `label` und `grid`

Weiter ist es natürlich auch möglich, Graphen zu benennen und eine Legende anzeigen zu lassen. Dazu sind zwei Dinge nötig:

- Im Plot-Befehl muss mit dem Keyword-Argument `label` ein Titel angegeben werden
- Mit dem Befehl `legend()` muss die Legende auch angezeigt werden:

Beispiel: Plot mit Legende

```
1  import math
2  import matplotlib.pyplot as plt
3
4  N  = 100
5  X  = [(x - N/2) / 10 for x in range(N)]
6  Y1 = [math.sin(x) for x in X]
7  Y2 = [math.cos(x) for x in X]
8
9  plt.plot(X, Y1, label="Sinus")
10 plt.plot(X, Y2, label="Cosinus")
11
12 plt.legend()
13 plt.show()
```

Ausgabe: Plot mit Legende

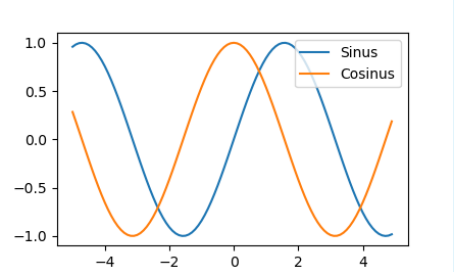


Abbildung 10.4.: Plot mit Legende

Dem Befehl `plot` können noch viele weitere Keyword-Arguments mitgegeben werden, auf die hier nicht weiter eingegangen werden kann. Sie können sich bei Bedarf selbst die Bedeutung und Anwendung dieser Schlüsselworte anlesen; siehe hierzu die Dokumentation unter `https://matplotlib.org/2.1.2/api/_as_gen/matplotlib.pyplot.plot.html`

Um die Werte aus dem Plot besser ablesbar zu machen können auch Hilfslinien dazugeschalten werden. Dies erreicht der Befehl `grid`:

Beispiel: Plot mit Gitterlinien

```
1  import math
2  import matplotlib.pyplot as plt
3
4  N  = 100
5  X  = [(x - N/2) / 10 for x in range(N)]
6  Y1 = [math.sin(x) for x in X]
7  Y2 = [math.cos(x) for x in X]
8
9  plt.plot(X, Y1, label="Sinus")
10 plt.plot(X, Y2, label="Cosinus")
11
12 plt.grid()
13
14 plt.legend()
15 plt.show()
```

Ausgabe: Plot mit Gitterlinien

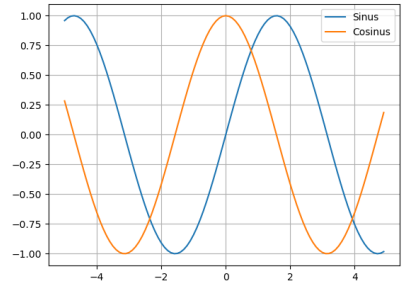


Abbildung 10.5.: Plot mit Gitterlinien

10.2.3. Titel und Achsenbeschriftung hinzufügen – `title`, `xlabel`, `ylabel`

Neben einer Legende sollten bei Plots auch Titel und Achsenbeschreibungen nicht fehlen. Diese lassen sich einfach über die Befehle `title`, `xlabel` und `ylabel` hinzugefügt werden. Das folgende Beispiel illustriert dies anhand der *spektralen Strahlungsdichte eines idealen schwarzen Körpers* [2]:

Beispiel: Plot mit Titel und Labels

```
1  import math
2  import matplotlib.pyplot as plt
3
4  h  = 6.62607015e-34     # Planck constant
5  T  = 300                # temperature in Kelvin
6  c  = 299792458          # speed of light
7  kB = 1.380649e-23       # Boltzmann constant
8
9  spectralDensity = lambda nu : ((2 * h * nu**3) / (c**2))  / \
10                               (math.exp((h * nu) / (kB * T)) - 1)
```

[2] Crash-Kurs Physik:
Jeder Körper strahlt elektromagnetische Strahlung aus – in einfachen Worten, jeder Körper leuchtet. Die Intensität und Lichtfarbe sind von der Temperatur abhängig. Heiße Körper glühen daher rot, sehr heiße Körper kommen sogar bis zur Weißglut; bei „kühlen“ 37 °C „leuchten“ wir Menschen nur im Infrarot-Bereich. Aus diesem Grund sehen wir zwar nicht immer die Strahlung, die von jedem Körper ausgestrahlt wird, können diese aber trotzdem messen. Das gezeigte Programm berechnet die Intensität der einzelnen Licht-Wellenlängen, die ein Körper mit einer gegebenen Temperatur `T` ausstrahlt.
Streng genommen spielt hierbei auch die Farbe des Körpers eine Rolle; daher sprechen PhysikerInnen auch von *Schwarzkörperstrahlung*. In der Praxis ist dieser Zusammenhang oft vernachlässigbar. Sekunden-Thermometer und Wärmebildkameras funktionieren nach diesem Prinzip: Ein Bild im Infraroten wird vom Objekt aufgenommen; die „Farbe“ der gemessenen Strahlung wird dann als Temperatur interpretiert. Im gezeigten Plot liegt das Strahlungs-Maximum bei ca. 20 THz, was einer Wellenlänge von 15 µm entspricht – weit außerhalb des Rahmens menschlicher Wahrnehmung, die zwischen ca. 400 und 800 nm stattfindet, und damit in Übereinstimmung mit der täglichen Wahrnehmung, dass wir unsere Mitmenschen nicht glühen sehen.
Wenn Sie in Zeile 5 den Wert für `T` auf `5700` ändern und in Zeile 12 den Plot-Bereich auf `1e+15` erweitern, wird Ihnen das Spektrum der Sonne mit Maximum im sichtbaren Licht dargestellt.

```
11  X = [x for x in range(1, int(1e+14), int(1e+10))]
12  Y = [spectralDensity(x) for x in X]
13
14  plt.title("Schwarzkörperstrahlung")
15  plt.xlabel("Strahlungsfrequenz in Hz")
16  plt.ylabel("Intensität in W/m²")
17
18  plt.plot(X, Y)
19  plt.show()
```

Ausgabe: Plot mit Titel und Labels

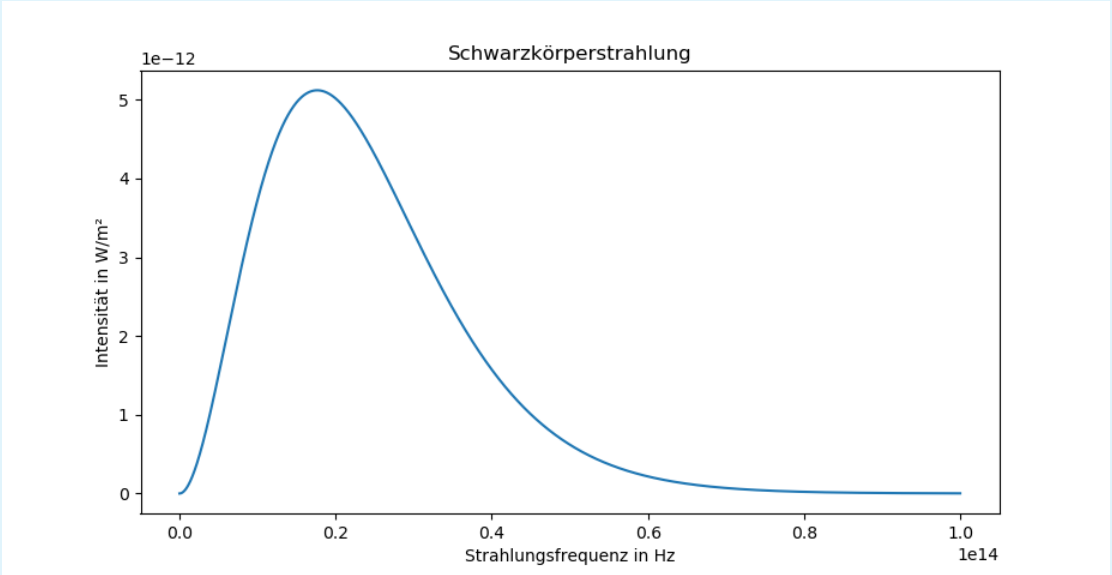


Abbildung 10.6.: Plot mit Titel und Achsenbeschriftungen

Siehe `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.axes.Axes.legend.html#matplotlib.axes.Axes.legend` für weitere Optionen zum Befehl `legend`.

10.2.4. Skalierung der Achsen – `xscale`, `yscale` und `xlim`, `ylim`

Wenn Plots einen großen Wertebereich abdecken, kann es sinnvoll sein, die Werte *logarithmisch* aufzutragen. Eine oder mehrere Achsen werden dabei verzerrt, um über mehrere Größenordnungen hinweg Änderungen gut zu verfolgen. Hierzu dienen die Befehle `xscale` und `yscale`.

Beispiel: Linearer und Logarithmischer Plot

```
1  import matplotlib.pyplot as plt
2
3  W  = 500
4  X  = [x / 10 for x in range(-W, W)]
5  Y1 = [2 ** x for x in X]
6  Y2 = [x ** 7 for x in X]
```

```
7  plt.title("Linear Plot")
8  plt.plot(X, Y1, label="exponential")
9  plt.plot(X, Y2, label="power")
10 plt.legend()
11 plt.show()
12
13 plt.title("Logarithmic Plot")
14 plt.yscale("log")
15 plt.plot(X, Y1, label="exponential")
16 plt.plot(X, Y2, label="power")
17 plt.legend()
18 plt.show()
```

Lineare Auftragung

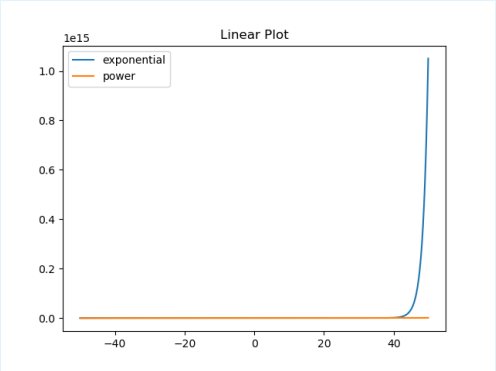


Abbildung 10.7.: Linearer Plot

Logarithmische Auftragung

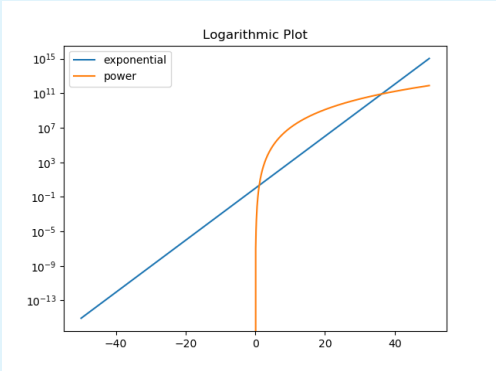


Abbildung 10.8.: Logarithmischer Plot

In Abbildung 10.7 können wir kaum Details erkennen: Beide Plots bleiben „nahe“ an der Null, bis die Exponentialfunktion irgendwann „explodiert“. Erst bei der logarithmischen Auftragung (Abbildung 10.8) sehen wir, dass das Polynom (orange Linie) zeitweise sogar größer ist als die Exponentialfunktion.

Wir haben diese Verzerrung erreicht, indem wir in Zeile 19 die y-Achse logarithmisch skaliert haben. Ebenso könnten wir mit `plt.xscale("log")` auch die x-Achse verzerren.

Der Logarithmus von 0 oder negativen Werten ist nicht definiert[3]. Um dennoch mit Werten umzugehen, die das Vorzeichen wechseln können und aber sinnvoller logarithmisch aufgetragen werden sollten, existiert auch die Option `"symlog"`. Hier wird jeweils der Logarithmus *des Betrags* der Werte aufgetragen; abhängig vom Vorzeichen geschieht dies dann nach oben oder unten (`plt.yscale("symlog")` bzw. nach links oder rechts (`plt.xscale("symlog")`. Werte nahe der 0 werden linear aufgetragen. Was als nahe der 0 gelten soll, kann mit den Keyword-Arguments `linthreshx` bzw. `linthreshy` bestimmt werden. Die Grenzen des linearen bereichs sind auch als Hilfslinien sichtbar, wenn `grid` benutzt wird.

Für statistische Auswertungen ist manchmal auch der „Logit“ einer Wahrscheinlichkeit interessant ($\text{logit}(y) = \log\left(\frac{y}{1-y}\right)$). Für y-Werte zwischen 0 und 1 kann so auch `plt.xscale("logit")` eingestellt werden.

[3] liebe MathematikerInnen: natürlich beziehe ich mich auf den reellwertigen Logarithmus bzw. auf den Hauptzweig des Logarithmus.

Beispiel: Auftragung mit `symlog`

```
import math
import matplotlib.pyplot as plt

W  = 500
X  = [x / 10 for x in range(-W, W)]
Y1 = [math.sin(x) for x in X]
Y2 = [x / 10      for x in X]

plt.xscale("symlog")
plt.plot(X, Y1)
plt.plot(X, Y2)
plt.grid()
plt.show()

plt.xscale("symlog", linthreshx=10)
plt.plot(X, Y1)
plt.plot(X, Y2)
plt.grid()
plt.show()
```

Auftragung mit `symlog`

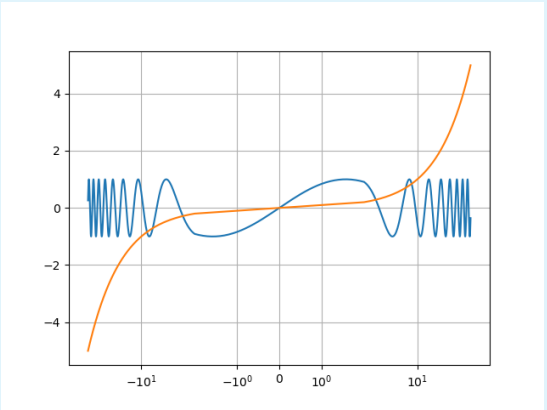


Abbildung 10.9.: symmetrisch-logarithmische Auftragung mit Standard-Schranke für lineare Auftragung

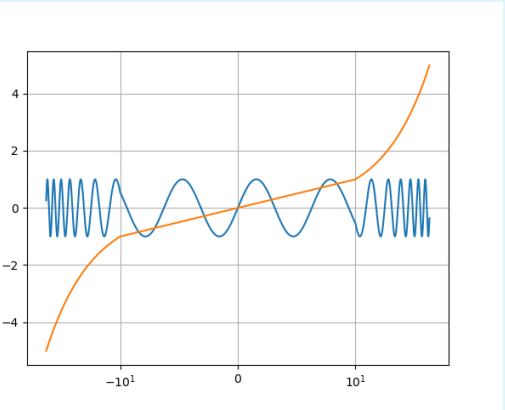


Abbildung 10.10.: symmetrisch-logarithmische Auftragung mit höherer Schranke für lineare Auftragung

In Ähnlicher Manier kann mit `xlim` und `ylim` festgelegt werden, in welchen Grenzen die X- und Y-Achse skaliert werden sollen, unabhängig von den „Ausmaßen“ des tatsächlichen Graphen. Der Graph füllt dann nicht mehr die gesamte Plot-Fläche aus, bzw. wird gegebenenfalls abgeschnitten:

Beispiel: Manuell gewählte Plot-Skalierung

```
import math
import matplotlib.pyplot as plt

N = 100
X = [(x - N/2) / 10 for x in range(N)]
Y = [math.sin(x) for x in X]

plt.plot(X, Y)
plt.xlim(-6  , +6)
plt.ylim(-0.7, +3)
plt.grid()
plt.show()
```

Manuell gewählte Plot-Skalierung

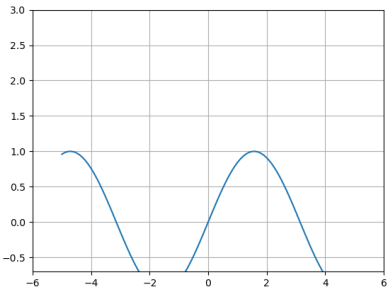


Abbildung 10.11.: Manuelle Skalierung

10.2.5. Andere Plot-Arten

Neben Kurven in einem X-Y-Diagramm kann die MatPlotLib auch andere Arten von Visualisierungen erzeugen.

Balkendiagramme

Barplots oder Balkendiagramme verhalten sich ähnlich wie die Punkt- oder Liniendiagramme, die im letzten Abschnitt gezeigt wurden. Alle oben erwähnten Befehle funktionieren auch hier, außerdem kann auch das Keyword-Argument `color` verwendet werden. Erstellt werden Barplots mit den Befehlen `bar` (vertikale Balken) und `barh` (horizontale Balken). Die „X-Werte“ dürfen für Barplots auch Strings enthalten, und werden entsprechend als Beschriftung angebracht. Wie schon bei den Liniendiagrammen können auch hier mehrere Graphen auf demselben Plot dargestellt werden.

Beispiel: Barplots

```
import matplotlib.pyplot as plt

X = ["Smoot", "Fnord", "R'lyeh"]
Y = [random.randint(0, 11) for x in X]    # this one goes up to eleven

plt.bar(X, Y, color="#0040B0FF")
plt.ylim(0, 11)
plt.show()

plt.barh(X, Y, color="#0040B0FF")
plt.xlim(0, 11)
plt.show()
```

Vertikaler Barplot

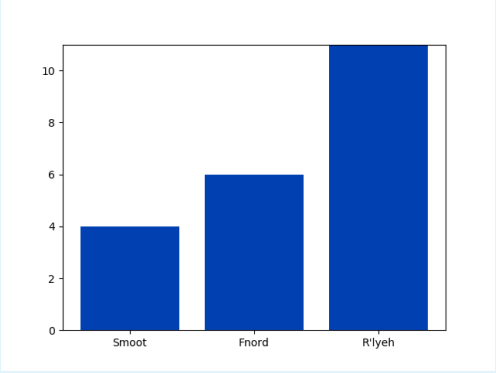


Abbildung 10.12.: Vertikaler Barplot

Horizontaler Barplot

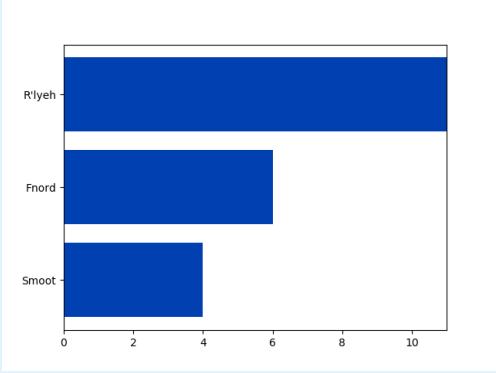


Abbildung 10.13.: Horizontaler Barplot

Siehe auch `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.bar.html` für weitere Details und `https://matplotlib.org/3.1.1/gallery/lines_bars_and_markers/bar_stacked.html` für ein weiteres Beispiel.

Kuchendiagramme

Kuchendiagramme sind die klassische Darstellungsform, wenn die Anteile verschiedener Beiträge zu einem Gesamten visualisiert werden sollen. Die MatPlotLib erlaubt das Erstellen solcher Diagramme mit dem Befehl `pie`. Wie auch schon zuvor kann im einfachsten Fall eine einfache `list` übergeben werden, um ein minimales Kuchendiagramm zu erstellen. Die Werte müssen sich dabei *nicht* zu 100% bzw. zu 1 aufaddieren.

Beispiel: Einfaches Kuchendiagramm

```
1  import matplotlib.pyplot as plt
2
3  contributions = {
4      "trial & error"                : 90,
5      "searching the web"            : 50,
6      "despair, fits of anger, etc" : 10,
7      "writing small bits of code"   : 30,
8      "brilliant ideas"              : 1
9  }
10
11 plt.pie( contributions.values() )
12 plt.show()
```

Ausgabe: Einfaches Kuchendiagramm

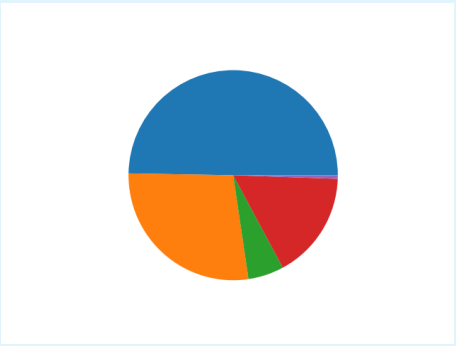


Abbildung 10.14.: Einfaches Kuchendiagramm der matplotlib

Selbstverständlich kann auch hier ein Titel mit `title` hinzugefügt werden. Die Befehle `xlabel` und `ylabel` erstellen tatsächlich Textfelder an den zu erwartenden Stellen, und können für Beschriftungen „missbraucht“ werden. Labels funktionieren wie schon bei `plot` gezeigt, d. h. über das optionale Argument `labels` werden Texte zugewiesen; der Befehl `legend` sorgt dafür, dass diese dann in einer Box angezeigt werden. Zusätzlich werden Labels direkt neben den Kuchenstücken angezeigt, wenn solche mit dem Keyword Argument `labels` als `list` übergeben werden. Dagegen hat der Befehl `grid` keine Effekt.

Eine Auswahl an optionalen Argumenten für den Befehl `pie` ist in Tabelle 10.2 aufgeführt. Weitere Argumente sind unter `https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.pyplot.pie.html` aufgeführt.

Auswahl an Optionalen Argumenten bei `pie`

Schlüsselwort	**Datentyp**	**Effekt**
`labels`	`list` oder `tuple`	Liste von Strings, die neben jedem Kuchenstück angezeigt werden soll. Die Länge der Liste muss mit der Anzahl der Kuchenstücke übereinstimmen.
`colors`	`list` oder `tuple`	Liste von Strings mit Farbangaben. Farbangaben können entweder einzelne Buchstaben wie in Tabelle 10.1 sein, ausgewählte englische Farbnamen oder hexadezimale Farbangaben
`explode`	`list` oder `tuple`	Liste von Zahlen, die angeben, wie weit jedes Kuchenstück vom Mittelpunkt abgesetzt sein soll in Einheiten des Kreisradius (d. h. 0 bedeutet: Kuchenstück-Spitze berührt den Mittelpunkt; 1 bedeutet: Kuchenstück-Spitze würde den Rand des Kreises berühren)
`autopct`	String oder Funktion	Wenn hier ein String übergeben wird, interpretiert die MatPlotLib diesen als Format-String (siehe Abschnitt B.2). Bei einer Funktion dagegen wird erwartet, dass die Funktion eine Zahl zwischen 0 und 100 als Argument akzeptiert, und einen String zurückgibt, der eine Textentsprechung dieser Zahl enthält.

Tabelle 10.2.: Optionale Argumenten bei `matplotlib.pyplot.pie` (Auswahl)

Beispiel: Verändertes Kuchendiagramm

```
import matplotlib.pyplot as plt

contributions = {
    "trial & error"                 : 99,
    "searching the web"             : 50,
    "despair, fits of anger, etc"   : 10,
    "writing small bits\n of code"  : 30,
    "brilliant ideas"               : 1
}

def percentToWords(x) :
    if        x <  5 : return "virtually nothing"
    elif  5 < x < 20 : return "a little"
    elif 20 < x < 50 : return "quite a bit"
    else             : return "the majority"

plt.title("Time Allocation in a coding project")
plt.xlabel("taken from experience")

plt.pie(
    contributions.values(),
    labels=contributions.keys(),
    colors=["#0000AAFF", "blue", "r", "green", "gold"],
    explode=(0, 0.1, 0, 0, .3),
    autopct=percentToWords
)
plt.show()
```

Ausgabe: Verändertes Kuchendiagramm

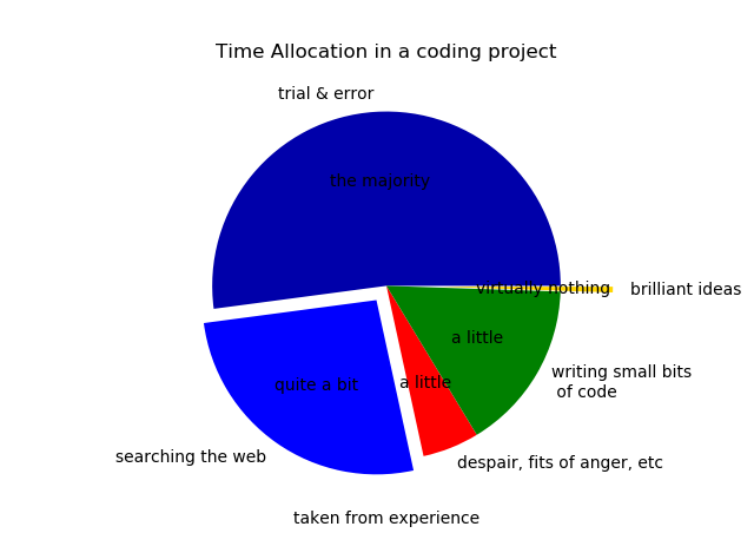


Abbildung 10.15.: Kuchendiagramm mit Optionen

Stackplots

Wo eine Zusammensetzung über eine Zeit verfolgt werden soll, bietet sich die Darstellungsfomr *stacked plot* an, die mit dem Befehl `stackplot` erstellt wird. Wie schon bei normalen Plots ist das erste, optionale Argument eine Liste von X-Werten. Danach können entweder mehrere eindimensionale Listen stehen, die als Y-Werte aufgetragen werden, oder eine zweidimensionale Liste:

Beispiel: Stackplot

```
import csv
import matplotlib.pyplot as plt

years    = []
africa   = []
americas = []
asia     = []
europe   = []
oceania  = []
colors   = ["gold", "black", "red", "blue", "green"]

with open("annual-number-of-births-by-world-region.csv", "r") as handle :
    # Daten von https://ourworldindata.org/world-population-growth
    # Datei enthält Daten in der Form:
    # Year,Asia,Africa,America,Europe,Oceania
    # 1950,11071.722,8896.969,62638.002,11837.577,360.997
    # 1951,11178.006,9031.823,62344.074,11917.33,366.362
    # ...

    reader = csv.reader(handle)

    labels = next(reader)[1:]    # erste Zeile lesen;
                                 # Spaltenüberschriften ohne "Years" übernehmen

    for line in reader :         # verbleibende Zeilen einlesen
        years   .append(float(line[0])       )
        africa  .append(float(line[1]) / 1000)  # line[i] / 1000: Daten sind in
        americas.append(float(line[2]) / 1000)  # Tausend Geburten angegeben.
        asia    .append(float(line[3]) / 1000)  # Plot soll in Millionen sein.
        europe  .append(float(line[4]) / 1000)
        oceania .append(float(line[5]) / 1000)


plt.title ("annual number of births by world region")
plt.xlabel("year")
plt.ylabel("annual number of births in millions")
plt.legend(loc="upper left")
```

```
38  plt.stackplot(
39      years,
40      asia, africa, americas, europe, oceania,
41      labels=labels,
42      colors=colors
43  )
44
45  # Alternative, erzeugt selbes Ergebnis:
46  # data = [asia, africa, americas, europe, oceania]
47  # plt.stackplot(years, data, labels = labels, colors=colors)
48
49  plt.show()
```

Ausgabe: Stackplot

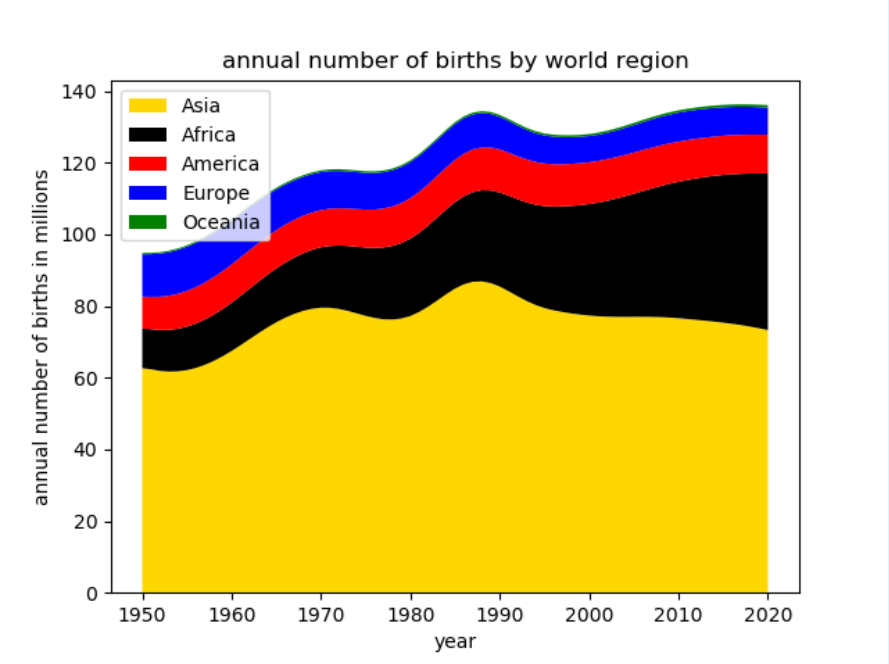


Abbildung 10.16.: Stackplot: Anzahl Geburten über die Zeit je Kontinent

Für weitere Details, siehe `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.stackplot.html`

Scatterplots

Scatterplots erlauben es, Orten in einer Fläche ein Gewicht zuzuweisen. Dem Befehl `scatter` können bis zu vier Parameter übergeben werden. Die ersten beiden davon sind X- und Y-Koordinaten der Punkte, die geplottet werden sollen, übergeben als `list`. Der dritte Parameter enthält eine `list` mit der Größe der Punkte und ist optional. Als vierten Parameter kann die Farbe der einzelnen Punkte als `list` übergeben werden.

Beispiel: Scatterplot

```
import csv
import matplotlib.pyplot as plt

lat = []       # geographische Länge
lng = []       # geographische Breite
pop = []       # Einwohnerzahl
col = []       # Farbe

colors = {
    "primary" : "red",      # Bundeshauptstadt
    "admin"   : "blue",     # Landeshauptstadt
    "minor"   : "black"     # andere Stadt
}

# Größe des Punktes aus Einwohnerzahl berechnen: kleine Städte sollen dabei
# nicht "verschwinden"
sizer = lambda x : 0.5 if x < 10000 else x / 10000

with open("cities-locations-populations-Germany.csv", "r") as handle :
    # Daten von https://simplemaps.com/data/de-cities
    # Datei enthält Daten in der Form:
    # city,lat,lng,country,iso2,admin_name,capital,population,population_proper
    # Berlin,52.5167,13.3833,Germany,DE,Berlin,primary,3644826,3644826
    # Hamburg,53.5500,10.0000,Germany,DE,Hamburg,admin,1841179,1841179
    # ...

    reader = csv.reader(handle)

    next(reader)

    for line in reader :
            lat.append(float(line[1]))
            lng.append(float(line[2]))
            pop.append( sizer(float(line[7]))  )
            col.append( colors[line[6]] )

plt.figure(figsize=(6,7.5))
plt.title("Deutschland: Städte")
plt.xlabel("Geographische Länge")
plt.ylabel("Geographische Breite")
plt.scatter(lng, lat, pop, col)
plt.show()
```

Ausgabe: Scatterplot

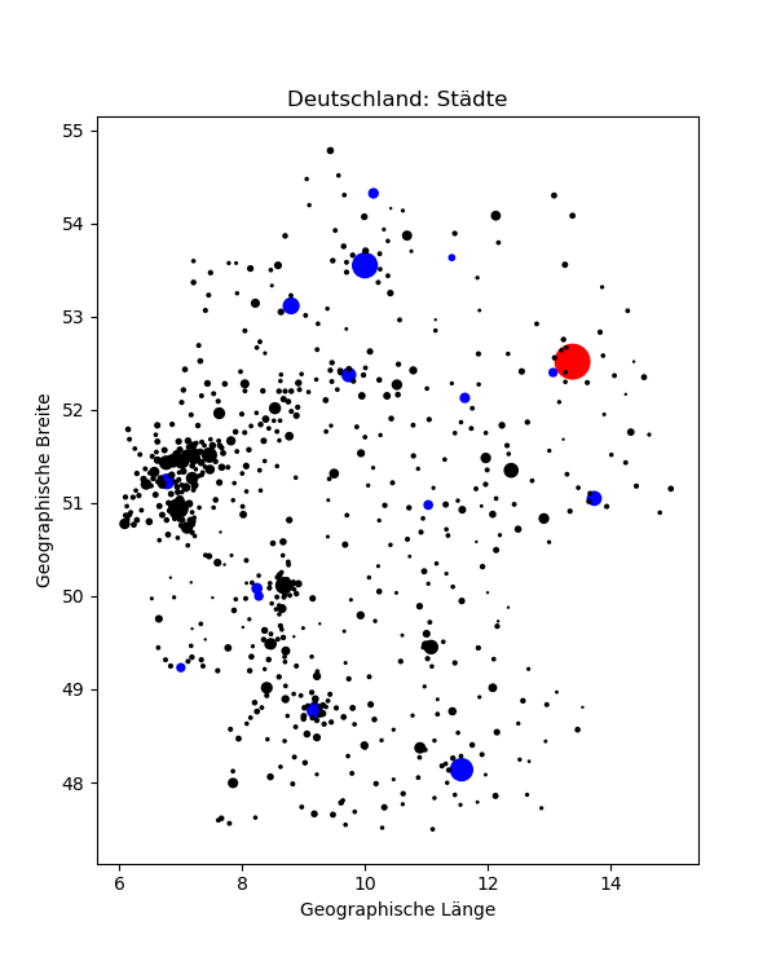


Abbildung 10.17.: Scatterplot: Städte Deutschlands

Siehe auch `https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.axes.Axes.scatter.html`

Histogramme

Häufig wird dieselbe Simulation mehrfach mit verschiedenen Ausgangswerten durchgeführt. Wir erhalten so eine Liste von Ergebnissen. Oft interessiert uns dann, wie häufig ein bestimmtes Ergebnis erhalten wurde.

Beispiel:
In der folgenden Liste:

```
[1, 5, 2, 1, 6, 2, 6, 8, 4]
```

finden wir die folgende Häufigkeit der Zahlen:

```
{1:2, 2:2, 4:1, 5:1, 6:2, 8:1}
```

Eine solche Aufschlüsselung nennt sich *Histogramm*. In der Regel führen wir dann *bins* ein, d. h. Wertebereiche, die als eine Klasse gezählt werden sollen. Wollen wir also jeweils zwei aufeinanderfolgende Zahlen als eine Klasse zählen, so erhalten wir folgende Aufschlüsselung:

```
{1:4, 3:1, 5:3, 7:1}
```

Mit der MatPlotLib können solche Histogramme automatisch erstellt[4] und als Plot dargestellt werden. Hierzu dient der Befehl `hist`. Als Parameter wird eine Liste von Ergebnissen übergeben, aus der automatisch das Histogramm berechnet und daraus ein Balkendiagramm erstellt. Rückgabewert von `hist` ist ein Tupel bestehend aus den `lists`[5] `count` und `bins` sowie einem `Patch`-Objekt. In `count[i]` ist gespeichert, wie viele Ergebnisse jeweils in die `i`-te Klasse fallen; `bins[i]` gibt an, bei welchem Wert die `i`-te Klasse beginnt. Auf das `Patch`-Objekt kann hier nicht näher eingegangen werden.

Beispiel: Einfaches Histogramm

```
1  import matplotlib.pyplot as plt
2
3  data = [1, 5, 2, 1, 6, 2, 6, 8, 4]
4
5  count, bins, patches = plt.hist(data)
6  print(bins)
7  print(count)
8  plt.show()
```

Konsolenausgabe: Einfaches Histogramm

```
[1.  1.7 2.4 3.1 3.8 4.5 5.2 5.9 6.6
 7.3 8. ]
[2. 2. 0. 0. 1. 1. 0. 2. 0. 1.]
```

Plot: Einfaches Histogramm

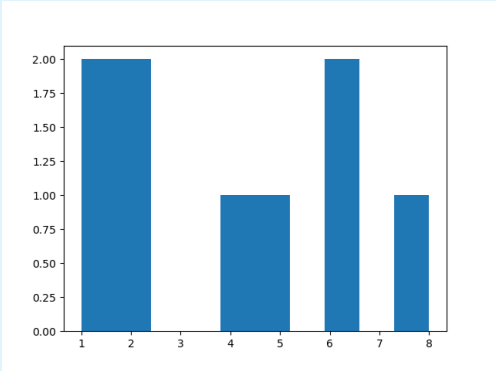


Abbildung 10.18.: Einfaches Histogramm

Der optionale `int`-Parameter `bins` gibt an, in wie viele Klassen diese Liste unterteilt werden soll. Über das Keyword-Argument `range` kann ein `tuple` übergeben werden, der die untere und obere Schranke für die Klassierung enthält.

Hier sei ein beschränkter *Random Walk* gezeigt: Ein Betrunkener geht `N` Schritte eine Straße entlang. Bei jedem Schritt wird er entweder nach links oder nach rechts torkeln; die Wahrscheinlichkeit für einen Schritt nach Links beträgt dabei `pleft`. Die Straße ist insgesamt `B` Schritte breit. Der Plot zeigt, wie wahrscheinlich der Betrunkene am Ende seines Laufs an einem

[4] Tatsächlich erfolgt die Datenanalyse im Hintergrund mit dem Modul NumPy, das in Kapitel 11 besprochen wird. Aus didaktischen Gründen wollen wir hier zuerst die MatPlotLib besprechen.
[5] Eigentlich NumPy-Arrays; diese verhalten sich aber im Wesentlichen wie `lists`. Siehe Kapitel 11 für Details.

Beispiel: Drunk Walk als Histogramm

```
import random
import matplotlib.pyplot as plt

runs   = 10000
N      = 100
B      = 20
pLeft  = 0.5
drifts = [0] * runs

for run in range(runs) :
    drift  = 0
    for step in range(N) :
        r = random.uniform(0, 1)

        if r < pLeft :                          # Schritt nach links
            if drift != -B : drift -= 1
        else :                                  # Schritt nach rechts
            if drift != +B : drift += 1

    drifts[run] = drift

data, bins, patches = plt.hist(drifts,
                               bins=B+1, range=(-B-1, B+1),
                               histtype='step'
)

plt.title ("Drunk Walk")
plt.xlabel("Drift")
plt.ylabel("Häufigkeit")

print("Histogramm-Daten:")
for b, d in zip(bins, data) :
    print(f"\t{b:+3.0f} bis {b+2:+3.0f}: {d:5.0f}")

plt.show()
```

Konsole: Drunk Walk als Histogramm

```
Histogramm-Daten:
        -21 bis -19:    93
        -19 bis -17:   253
        -17 bis -15:   254
        -15 bis -13:   317
        -13 bis -11:   388
        -11 bis  -9:   519
         -9 bis  -7:   589
         -7 bis  -5:   671
         -5 bis  -3:   735
         -3 bis  -1:   783
         -1 bis  +1:   789
         +1 bis  +3:   769
         +3 bis  +5:   775
         +5 bis  +7:   669
         +7 bis  +9:   571
         +9 bis +11:   484
        +11 bis +13:   367
        +13 bis +15:   308
        +15 bis +17:   281
        +17 bis +19:   191
        +19 bis +21:   194
```

Plot: Drunk Walk als Histogramm

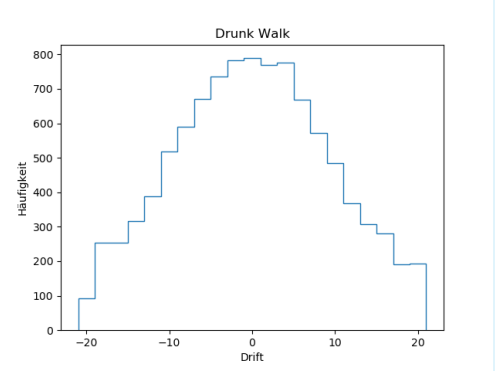


Abbildung 10.19.: Histogramm zum Drunk Walk

Für weitere Parameter, siehe `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.axes.Axes.hist.html`

Vektorfelder

Die Größen, die wir auftragen wollen, können nicht nur eindimensionale Größen sein, sondern auch eine „Richtung“ haben. Beispielsweise könnten wir Windgeschwindigkeit und -Richtung über einem großen Gebiet darstellen wollen. Zu diesem Zweck dient der Befehl `quiver`.

Ähnlich wie schon bei `plot` und den anderen gezeigten Befehlen reicht es im einfachsten Fall, nur die zu plottenden Daten selbst zu übergeben. Diese müssen dann in *zwei* Listen übergeben werden, da ja auch *zwei*dimensionale Daten dargestellt werden sollen. Die erste Liste enthält also die X-Komponenten der Windrichtungen; in der zweiten Liste sind die Y-Komponenten angegeben. Die Listen selbst müssen dann ebenfalls *zwei*dimensional sein, da die Windgeschwindigkeit an einem *Ort* angegeben wird, der selbst eine X- und Y-Komponente hat.

Beispiel: Wirbelfeld ohne explizite Ortsangabe

```
1  import matplotlib.pyplot as plt
2
3  width  = 10
4  height = 15
5
6  xPoints = [x / 10 for x in range(- width, + width)]
7  yPoints = [y / 10 for y in range(-height, +height)]
```

```
dataX = []
dataY = []

for r, y in enumerate(yPoints) :
    dataX.append([])
    dataY.append([])
    for x in xPoints :
        dataX[r].append(-y)
        dataY[r].append(+x)

plt.quiver(dataX, dataY)
plt.show()
```

Plot: Wirbelfeld

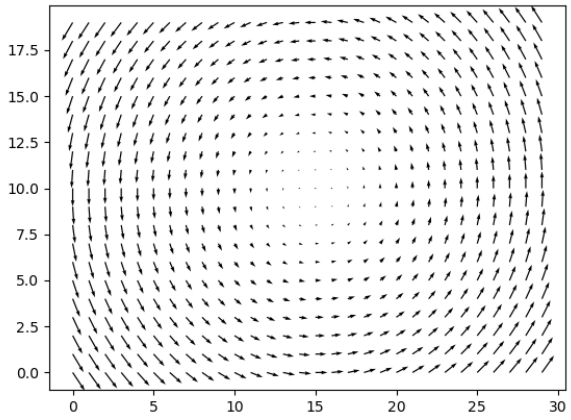


Abbildung 10.20.: Wirbelfeld

Wie schon zuvor nimmt die MatPlotLib hierzu an, dass die Punkte einen Abstand von 1 zueinander haben, sowohl in X- als auch Y-Richtung. Wo dies nicht der Fall ist, können auch zwei Listen `X, Y` mit übergeben werden, die jeweils die X- bzw. Y-Koordinate zum `i`-ten Datenpunkt[6]. In diesem Fall dürfen die Listen mit den Pfeil-Daten auch als *eindimensionale* Liste übergeben werden, d. h. ihr Index weist ihnen schon ihre Position im Raum zu. Den obigen Plot kann man daher auch mit diesen Zeilen erreichen:

Beispiel: Wirbelfeld ohne explizite Ortsangabe

```
import matplotlib.pyplot as plt

width  = 10
height = 15
xPoints = [x / 10 for x in range(- width, + width)]
yPoints = [y / 10 for y in range(-height, +height)]
```

[6] solche Koordinatenlisten können sehr bequem mit der Funktion `meshgrid` aus dem Modul `numpy` erstellt werden. Siehe dazu Kapitel 11

```
X = [xPoints for i in range(2 * height)]
Y = [[y] * 2 * width for y in yPoints]

dataX = []
dataY = []

for y in yPoints :
    for x in xPoints :
        dataX.append(-y)
        dataY.append(+x)

plt.quiver(X, Y, dataX, dataY)
plt.show()
```

Für weitere Parameter, siehe `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.axes.Axes.quiver.html`

Weitere Diagrammtypen

Unter `https://matplotlib.org/3.1.0/gallery/index.html` sind diverse Beispiele zum Umgang mit der MatPlotLib aufgeführt.

Beachten Sie auch, dass verschiedene Plot-Typen miteinander vermischt werden können. Ein Balkendiagramm und eine Datenkurve können sich problemlos überlagern.

10.3. Multiplots

Es ist auch möglich, in einem Fenster mehrere Plots darzustellen. Hierzu dienen die Befehle `subplot` und `subplots`. Mit `subplots` wird ein Gitter festgelegt, auf dem die einzelnen Plots im Fenster angeordnet werden. Übergeben werden muss dazu die Anzahl an Zeilen und Spalten, die dieses Gitter haben soll.

```
plt.subplots(2, 4)
```

bereitet also ein Fenster vor, in dem insgesamt 8 Plots dargestellt werden können. Diese Plots sind dann in 2 Zeilen und 4 Spalten angeordnet.

Der Befehl `subplot` wählt aus, an welchen Gitterpunkt der nächste Plot platziert wird. Hierzu ist nochmal die Höhe und Breite des Gitters notwendig; zusätzlich muss auch die „Nummer des Gitterpunkts“ angegeben werden, an die der Plot platziert werden soll. Diese Nummer beginnt bei 1 und wird von links nach rechts und von oben nach unten durchgezählt. Im oben angelegten 2x4-Gitter wählt also

```
plt.subplot(2, 4, 5)
```

den Rasterpunkt in der zweiten Zeile, erste Spalte aus.

Sofern Höhe, Breite und Gitterpunkt-Nummer nur einstellige Ziffern sind, können diese auch zu einer dreistelligen Zahl zusammengefasst werden.

```
plt.subplot(245)
```

hat also denselben Effekt, wählt ebenfalls den Rasterpunkt in der zweiten Zeile, erste Spalte aus.

Weiter kann dem Befehl `subplot` noch über das Keyword-Argument `polar` ein `boolean` mitgegeben werden, der eine Auftragung in Polarkoordinaten verursacht.

Beispiel: Lissajous-Figuren

```
1  import math
2  import matplotlib.pyplot as plt
3
4  X = [x / 100 for x in range(628)]
5  Y = [math.sin(3 * x) for x in X]
6
7  plt.subplots(1, 2)
8  plt.suptitle("Lissajous-Figuren")
9
10 plt.subplot(121)
11 plt.title("Kartesisch")
12 plt.plot(X, Y)
13
14 plt.subplot(122, polar=True)
15 plt.title("Polarkoordinaten")
16 plt.plot(X, Y)
17
18 plt.show()
```

Ausgabe: Lissajous-Figuren

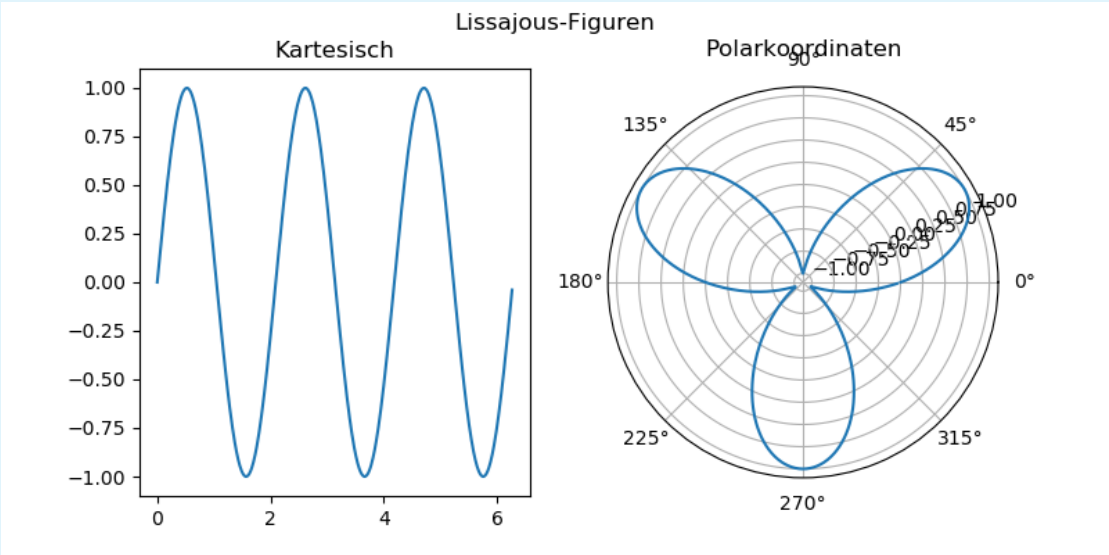


Abbildung 10.21.: Zwei Plots im selben Fenster

Siehe `https://matplotlib.org/3.3.3/api/_as_gen/matplotlib.pyplot.subplots.html` und `https://matplotlib.org/3.3.3/api/_as_gen/matplotlib.pyplot.subplot.html` für weitere Details.

10.4. Plot-Objekte

Wir haben bisher Befehle kennengelernt, die Plots verschiedener Art erstellen und deren Eigenschaften steuern. Im Hintergrund werden hierzu Instanzen von verschiedenen Klassen angelegt. Uns als EntwicklerInnen steht ein solches *objektorientiertes* Herangehen an die Plots auch zur Verfügung: wir können mithilfe der MatPlotLib Objekte erstellen, die die verschiedenen Elemente eines Plots (Fenster, Achsen, Kurven, ...) repräsentieren, und diese Elemente dann über *Methoden* verändern. Dieses Herangehen verlangt eine etwas andere Denkweise, die vielen als natürlicher vorkommt.

Neben diesem Paradigmenwechsel ist es durch die objektorientierte Formulierung von Plots auch möglich, mehrere Plot-Fenster gleichzeitig zu steuern. Außerdem kann Python Code *parallel zum Fenster* ausgeführt werden: normalerweise wird die Code-Ausführung mit `plt.show()` angehalten, bis das Plot-Fenster geschlossen wird. Im folgenden zeige ich, wie man Python-Code dazu bringt, gleichzeitig zum Betreiben des/der Plot-Fenster(s) weiter den Code auszuführen.

10.4.1. Plot-Fenster und `AxesSubplot`-Objekte

Bisher haben wir alle Befehle der MatPlotLib direkt über den Alias `plt` aufgerufen. Tatsächlich werden hierbei verschiedene Instanzen von Klassen angelegt und von der MatPlotLib im Hintergrund verwaltet. Besondere Bedeutung haben dabei die Klassen `Figure` und `AxesSubplot`.

Eine `Figure` stellt ein Plot-Fenster dar. Wir erhalten ein *Handle auf ein neues Fenster* über den Befehl `figure`:

```
fig = plt.figure()
```

Aktionen, die *das Fenster als Ganzes* betreffen, sollten über dieses Handle geschehen.

```
fig.suptitle("Überschrift")
```

ändert beispielsweise die Titelzeile des Fensters.

Der Befehl `figure` hat eine Reihe von Keyword-Parametern, von denen hier nur `figsize` vorgestellt werden soll. `figsize` erwartet einen Tupel aus zwei Werten, die die Breite und Höhe des Plots repräsentieren. Für weitere Details, siehe `https://matplotlib.org/3.3.3/api/_as_gen/matplotlib.pyplot.figure.html`.

`AxesSubplot`s repräsentieren die eigentlichen Graphen. Wir erhalten ein Handle auf ein solches Objekt über verschiedene Methoden von `figure`. Am geläufigsten ist die Methode `add_subplot`, die sich im Wesentlichen wie der Befehl `subplot` verhält:

```
pol = fig.add_subplot(1, 2, 2, projection="polar")
```

erzeugt einen Subplot in Polarkoordinaten in der ersten Zeile, zweite Spalte von `fig`. Siehe `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.figure.Figure.html?highlight=add_subplot#matplotlib.figure.Figure.add_subplot` für weitere Details.

Die obige Ausgabe von Abb. 10.21 kann so auch mit dem folgenden Code erreicht werden:

Beispiel: Lissajous-Figuren, Objektorientiert

```
1  import math
2  import matplotlib.pyplot as plt
3
4  X = [x / 100 for x in range(628)]
5  Y = [math.sin(3 * x) for x in X]
6
7  fig = plt.figure(figsize=(8,4))
8  fig.suptitle("Lissajous-Figuren")
9
10 crt = fig.add_subplot(1, 2, 1)
11 crt.set_title("Kartesisch")
12 crt.plot(X, Y)
13
14 pol = fig.add_subplot(1, 2, 2, projection="polar")
15 pol.set_title("Polarkoordinaten")
16 pol.plot(X, Y)
17
18 fig.show()
19
20 print("Ausgabe noch während der Plot dargestellt wird")
21 input()
```

Hier erfolgt die Ausgabe von Zeile 20 schon während der Plot dargestellt wird, und nicht erst, nachdem das Plot-Fenster geschlossen wird. Aus demselben Grund ist auch Zeile 21 notwendig: Der Code läuft nach `fig.show()` weiter, bis die letzte Zeile ausgeführt wurde; danach beendet Python alle Prozesse, die mit dem Code zu tun hatten, einschließlich der Darstellung des Plots. Mit `input` zwingen wir Python dazu, auf eine Usereingabe zu warten.

10.4.2. Gridspecs

Erweiterte Kontrolle darüber, wie die Graphen im Plot-Fenster platziert werden, erhalten wir über das `GridSpec`-Objekt. Dieses ist eine Representation des Gitters, in das die Plots eingetragen werden. Dieses Gitter wird auf das Fenster gelegt, welches wir mit `plt.figure` angelegt haben; daher auch erhalten wir ein `GridSpec`-Objekt über eine Methode der Klasse `figure`. Diese Methode heißt `add_gridspec` und erwartet zwei `int`-Werte, die die Anzahl der Zeilen und Spalten im Gitter darstellen – also dieselben Werte, wie wir sie schon von `subplots` bzw. `subplot` kennen.

Der Rückgabewert von `add_gridspec` ist ein Array-artiges Objekt, d. h. wir können die einzelenen Zellen des Gitters über Indices in eckigen Klammern ansprechen. Auch Slicing ist möglich, um zusammenhängende Bereiche auszuzeichnen. Wenn `gs` das `GridSpec`-Objekt ist, dann ist `gs[1, 0]` die Zelle in der zweiten Zeile, erste Spalte; `gs[:, 2]` stellt die *gesamte* dritte Spalte dar.

Dieses `GridSpec`-Objekt kann auch an `add_subplot` übergeben werden, und legt wieder fest, wo im Fenster as zugehörige `AxisSubplot`-Objekt platziert werden soll.

Wenn zwei Graphen „verwandte“ Daten darstellen sollen, wollen Sie vielleicht auch, dass die Skalierung der Achsen in beiden Graphen gleich sind. Dies erreichen Sie durch die Keyword-Arguments `sharex` und `sharey` in `add_subplot`. Diesen wird jeweils das `AxisSubplot`-Objekt übergeben, an das die X- bzw. Y-Achse gekoppelt werden soll.

Das folgende Beispiel nutzt ein `GridSpec`-Objekt, um einen großen Plot und zwei kleinere Plots anzuordnen. Nicht alle hierin verwendeten Methoden der MatPlotLib wurden bereits besprochen; sicher aber können Sie sich aus dem Kontext erschließen, welchen Zweck diese Stellen erfüllen[7].

Beispiel: 2D-Normalverteilung

```
1  import random
2  import matplotlib.pyplot as plt
3
4  N      = 1000
5  X      =   50
6  Y      =   40
7  sigmaX =    5
8  sigmaY =   10
9
10 dataX = [random.gauss(X, sigmaX) for _ in range(N)]
11 dataY = [random.gauss(Y, sigmaY) for _ in range(N)]
12
13 fig = plt.figure(figsize=(8,8))
14 gs  = fig.add_gridspec(3, 3)
15
16 fig.suptitle("2D-Normalverteilung")
17 fig.subplots_adjust(wspace=0.2, hspace=0.3)
18
19 axScatter = fig.add_subplot(gs[0:2, 0:2])
20 axScatter.set_xlim(0, 100)
21 axScatter.set_ylim(0, 100)
22 axScatter.set_xlabel("x")
23
24 axHistX   = fig.add_subplot(gs[ 2 , 0:2], sharex=axScatter)
25 axHistY   = fig.add_subplot(gs[0:2,  2 ], sharey=axScatter)
26
27 axScatter.scatter(dataX, dataY, marker=".")
28 axHistX.hist(dataX, orientation='vertical'   , bins=20)
29 axHistY.hist(dataY, orientation='horizontal', bins=20)
30
31 fig.show()
32 input()
```

[7] Die MatPlotLib bringt eine gewaltige Fülle an Funktionen mit, die kaum zu überblicken oder im Kopf zu behalten ist. In der Praxis heißt die Arbeit mit de MatPlotLib oft, im Internet nach Beispielen zu suchen, und aus den dort gezeigten Code-Schnipseln die Stellen zu isolieren, die das gesuchte Feature aktivieren. Sie üben hier also eine Kernkompetenz des Programmierens: unbekannten Code zu interpretieren. Kopieren Sie den Code und experimentieren Sie mit den Einstellungen, um tiefere Einblicke zu gewinnen! Sie können auch immer die QuickSearch-Funktion auf `https://matplotlib.org/3.1.1/index.html` zur Hilfe nehmen.

Ausgabe: 2D-Normalverteilung

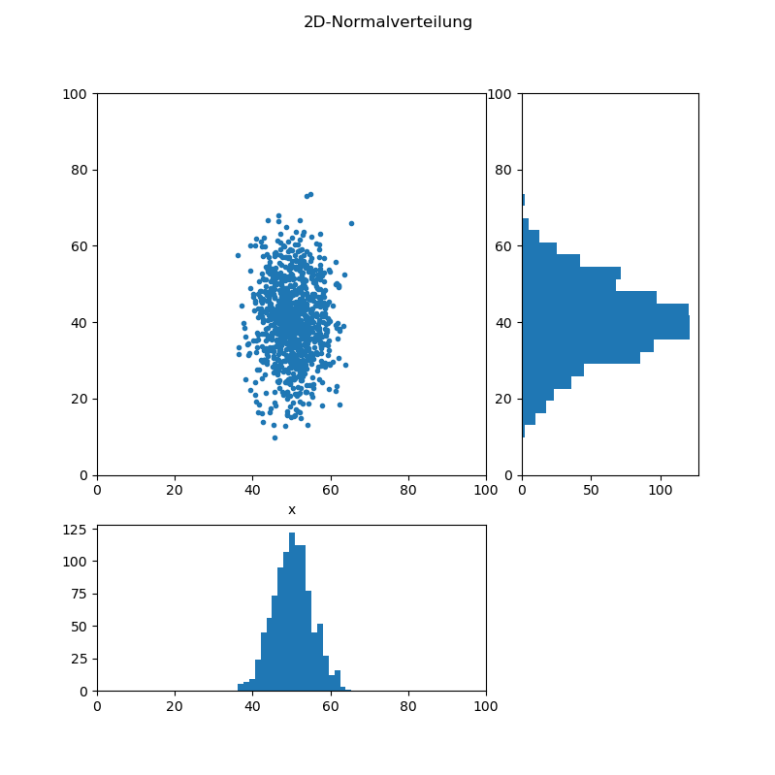


Abbildung 10.22.: Komplexere Anordnung von Plots: 2D-Normalverteilung

10.4.3. Manipulation des `AxisSubplot`-Objekts

Wie Sie sich schon aus dem letzten Beispiel erschlossen haben, können mit `set_xlim`, `set_ylim` die Grenzen festgelegt werden, auf die die X- bzw. Y-Achse skaliert werden. In ähnlicher Weise dienen die Befehle `set_title`, `set_xlabel`, `set_ylabel`, `set_xscale` und `set_yscale` dazu, Überschrift des Graphen, Achsenbeschriftung und Auftragung (linear, logarithmisch, ...) zu bestimmen. Alle diese Methoden können auf ein `AxisSubplot`-Objekt angewandt werden.

Getter und Setter

Ebenso wie es mit `set_XXX` möglich ist, den Wert `XXX` eines `AxisSubplot`-Objekts festzulegen, kann mit `get_XXX` der aktuell eingespeicherte Wert abgefragt werden. Dies ist eine gängige Praxis in der objektorientierten Programmierung: Wenn eine Klasse ein Attribut `XXX` hat, so sollte es hierzu auch die Methoden `get_XXX` und `set_XXX` geben.

Wir werden auf solche Design-Pattern in Kaptiel **??** nochmals eingehen.

Die Bemaßung eines `AxisSubplot`-Objekts `ax` kann auch mit `ax.axis` gesetzt bzw. abgefragt werden.

```
xmin, xmax = ax.get_xlim()
ymin, ymax = ax.get_ylim()
```
bzw.
```
ax.set_xlim(xmin, xmax)
ax.set_ylim(ymin, ymax)
```

sind also äquivalent zu

```
xmin, xmax, ymin, ymax = ax.axis()
```
bzw.
```
ax.axis(xmin, xmax, ymin, ymax) .
```

Zusätzlich erlaubt die Methode `axis` auch, die Achsen eines Graphen komplett zu verbergen (`ax.axis("off")`), so zu skalieren dass gleiche Abstände in X- und Y-Richtung auch gleichen Skalenschritten entsprechen (`ax.axis("equal")`), oder dass die Graphen-Fläche zu einem Quadrat verzerrt wird (`ax.axis("square")`). Siehe hierzu auch `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.axes.Axes.axis.html#matplotlib.axes.Axes.axis`.

Über `set_xtics` und `set_ytics` lässt sich einstellen, an welchen Punkten die X- bzw. Y-Achse unterteilt werden sollen. Wir unterscheiden hierbei *major ticks* (größere Abstände und längere Striche, i. d. R. mit Zahlenwert) und *minor ticks* (kleinere Abstände, kurzere Striche, i. d. R. ohne Zahlenwert). Übergeben werden hierfür verpflichtend eine `list` mit den Zahlenwerten, an denen Ticks gesetzt werden sollen. Das Optionale Schlüsselwort `minor` kann entweder auf `True` oder `False` gesetzt werden; entsprechend gilt die Liste der Ticks dann auch für die *minor* oder *major* Ticks.

Beispiel: Major- und Minor Ticks

```
1  import matplotlib.pyplot as plt
2
3  fig = plt.figure(figsize=(6,2))
4  gfx = fig.subplots()
5
6  xMajor = [10 / x    for x in range(1,  5)]
7  xMinor = [5 + x / 2 for x in range(1, 10)]
8
9  gfx.set_xticks(xMajor)
10 gfx.set_xticks(xMinor, True)
11
12 plt.show()
```

Ausgabe: Major- und Minor Ticks

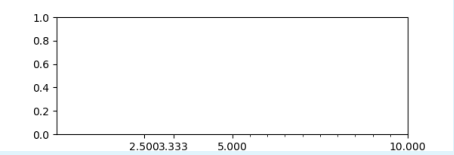


Abbildung 10.23.: Major- und Minor Ticks
Man beachte die kleinen Unterteilungen zwischen 5.0 und 10.0 (minor ticks)

Siehe auch `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.axes.Axes.grid.html#matplotlib.axes.Axes.grid` für Methoden zur Beeinflussung der Gitternetzlinien.

10.4.4. Ticker

Wir können nicht nur beeinflussen, welche Werte an den Achsen durch Ticks markiert sein sollen, sondern auch, welche Beschriftungen dort stehen sollen. Im einfachsten Fall können wir einfach direkt eine Liste von Texten vorgeben. Hierzu dienen die Befehle `set_xticklabels` und `set_yticklabels`. *Verpflichtend* muss eine `list` von Strings übergeben werden, die nacheinander an der X- bzw. Y-Achse angetragen werden sollen. Der erste Wert der Liste wird an den ersten Tick gesetzt, der Zweite an den Zweiten, etc.

Weiter gibt es die optionalen Keyword-Arguments `minor` und `fontdict`. Wie schon bei `set_xtics` / `set_ytics` ist `minor` ein `bool` und dient der Zuordnung zu den Major- oder Minor Ticks. Standardmäßig ist dieses Argument `False`, d. h. man beschreibt die Major Ticks.

Wie es der Name vermuten lässt, ist `fontdict` ein `dict`, in dem Einstellungen zur Schriftart enthalten sind.

Beispiel:

```
ax.set_xticklabels(
    ['Bill', 'Fred', 'Mary', 'Sue'],
    fontdict={'fontsize' :15, 'fontweight' : 'bold'}
)
```

weist der X-Achse die vier Labels `'Bill'`, `'Fred'`, `'Mary'` und `'Sue'` zu. Diese werden In Schriftgröße 15 in Fettdruck gerendert.

Siehe `https://matplotlib.org/3.3.3/api/_as_gen/matplotlib.axes.Axes.set_xticklabels.html` für weitere Details.

Wo mehr Flexibilität nötig ist, kann das Modul `matplotlib.ticker` verwendet werden. Dieses stellt mehrere Methoden zur Verfügung, wie automatisiert Beschriftungen aus den Achsenwerten generiert werden können. Besonders praktisch ist hierbei die Idee eines *FuncFormatters*: Hierbei handelt es sich um eine benutzerdefinierte Funktion, die aus dem Achsenwert eine vollständige Beschreibung (Text, Schriftart, Position, ...) der Beschriftung generiert. Einen solcher FuncFormatter erhalten wir, indem wir

- eine Funktion schreiben, die aus einem Zahlenwert den Text für die Beschriftung erzeugt.
  - Die Funktion muss zwei Parameter annehmen: Wert auf der Achse und Tick-Nummer
  - Der Rückgabetyp der Funktion muss ein String sein.
- diese Funktion an `matplotlib.ticker.FuncFormatter` übergeben.

Ein solches `FuncFormatter`-Objekt kann dann an `ax.xaxis.set_major_formatter`, `ax.xaxis.set_minor_formatter`, `ax.yaxis.set_major_formatter` oder `ax.yaxis.set_minor_formatter` übergeben werden.

Beispiel: FuncFormatter für Y-Ticks

```
1  from matplotlib.ticker import FuncFormatter
2  import matplotlib.pyplot as plt
3
4  x = list(range(4))
5  money = [1.5e5, 2.5e6, 5.5e6, 2.0e7]
6
7  # Der Plot selbst
8  fig = plt.figure(figsize=(7,4))
9  ax = fig.subplots()
10 ax.bar(x, money)
11
12 # Beschriftung der X-Achse
13 ax.set_xticks(x)
14 ax.set_xticklabels(
15   ['Bill', 'Fred', 'Mary', 'Sue'],
16   fontdict={'fontsize' :15, 'fontweight' : 'bold'}
17 )
18
19 # Beschriftung der Y-Achse
20 def myTicks(x, pos):
21     return f"{(x * 1e-6):1.1f}M, #{pos}"
22
23
24 formatter = FuncFormatter(myTicks)
25 print(formatter)
26 ax.yaxis.set_major_formatter(formatter)
27
28 plt.show()
```

Ausgabe: FuncFormatter für Y-Ticks

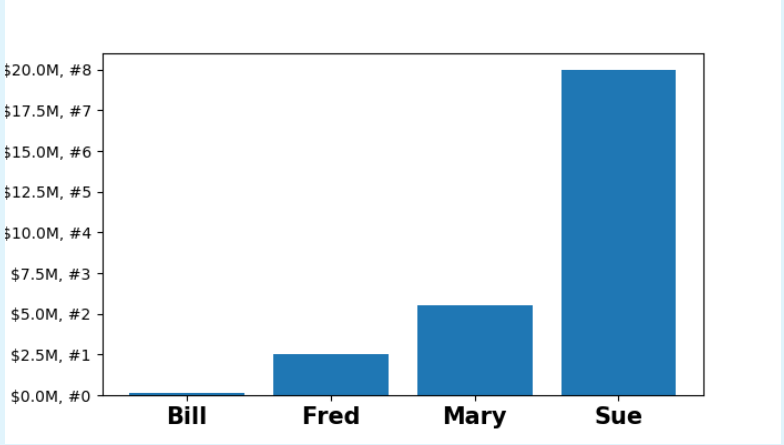


Abbildung 10.24.: Major- und Minor Ticks

Siehe hierzu auch `https://matplotlib.org/3.1.1/gallery/ticks_and_spines/tick-formatters.html` für Beispiele.

10.4.5. LaTeX-Elemente in Plots

Python enthält ein eigenständiges LaTeX-Modul, das zur Darstellung von einfachen mathematischen Gleichungen geeignet ist. Um Teile eines Strings als LaTeX-Code zu interpretieren, können diese einfach durch Dollarzeichen `$` eingerahmt werden. So führt beispielsweise

```
plt.title(r"$\frac {\pi }{2} \approx $ 3.1415")
```

zu der Ausgabe

$$\frac{\pi}{2} \approx 3.1415$$

Escaping Unterbinden: R-Strings

Befehle in LaTeXbeginnen mit einem Backslash (\). In Python wird der Backslash in Strings als Escape-Zeichen interpretiert, d. h. nachfolgende Zeichen werden anders interpretiert, z. B. `\n` für einen Zeilenumbruch. Um dies zu unterbinden kann einem String ein `r` vorangestellt werden, wie im obigen Beispiel gezeigt.

Die MatPlotLib verwaltet eine ganze Reihe von Standard-Einstellungen. Diese sind in einem `dict` namens `matplotlib.rcparams` gespeichert und können zu jeder Zeit geändert werden. Die Schlüssel des `dict`s sind einfache Strings. Soll für aufwändigere LaTeX-Elemente nicht das Python-Backend für LaTeXbenutzt werden sondern ein bereits auf dem Rechner vorliegender LaTeX-Compiler genutzt werden, so kann dies über den `dict`-Eintrag `text.usetex` geschehen. Zugeordnet ist ein `bool`, der festlegt, ob Labeltexte generell als LaTeX-Code Interpretiert werden sollen.

Indem wir also die Zeilen

```
import matplotlib
matplotlib.rcParams['text.usetex'] = True
```

zu unserem Plot hinzufügen, können wir sehr einfach professionell aussehende Labels erzeugen und auch Formel-Elemente einfügen:

Beispiel: Mathematisches Pendel mit LaTeX-Elementen

```
import math
import matplotlib
matplotlib.rcParams['text.usetex'] = True
import matplotlib.pyplot as plt

T = [x / 100 for x in range(100)]
S = [math.cos(4 * math.pi * t) + 2 for t in T]

plt.plot(T, S)
plt.xlabel(r'\textbf{time (s)}')
plt.ylabel('\\textit{Velocity (\N{DEGREE SIGN}/sec)}', fontsize=16)
plt.title (r'\TeX\ is Number $\displaystyle\sum_{n=1}^\infty'
           r'\frac{-e^{i\pi}}{2^n}$!', fontsize=16, color='r')
plt.show()
```

Ausgabe: Mathematisches Pendel mit LaTeX-Elementen

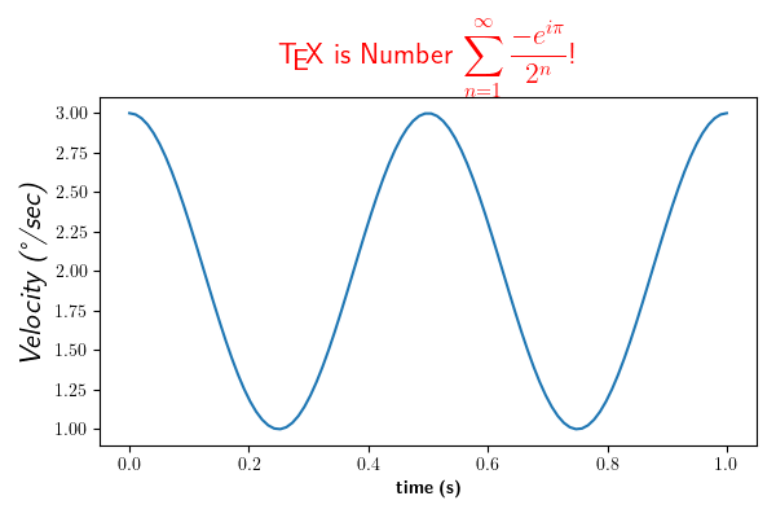


Abbildung 10.25.: LaTeX-Elemente im Plot

Fehlende Komponenten Nachinstallieren

Wenn Sie die Beispiele aus diesem Abschnitt ausführen, erhalten Sie möglicherweise die Fehlermeldung
RuntimeError: Failed to process string with tex because dvipng could not be found
oder ähnliche Hinweise auf fehlende Komponenten.

Bei der empfohlenen Anaconda-Installation kann das Python-eigene LaTeX-Modul aus der Anacona-Konsole heraus nachinstalliert werden mit:
`pip install latex`
Linux-UserInnen geben diesen Befehl in die normale Konsole ein (ggf. mit `sudo`).

Für das Beispiel *Mathematisches Pendel mit LaTeX-Elementen* muss ein LaTeX-Compiler installiert sein. Als Linux-UserIn können Sie die Komponenten nachinstallieren mit dem Konsolen-Befehl:
`sudo apt install texlive texlive-latex-extra texlive-fonts-recommended dvipng`
Beachten Sie, dass die Installation lange dauern kann und nennenswert Festplattenspeicher beansprucht.

Windows- und Mac-UserInnen halten sich hierfür am besten an die Standard-Installation, wie von der TeX Live Website `https://www.tug.org/texlive/` angeboten.

Siehe auch `https://matplotlib.org/3.3.3/tutorials/introductory/customizing.html?highlight=text.usetex#matplotlib-rcparams` und `https://matplotlib.org/3.3.3/gallery/text_labels_and_annotations/tex_demo.html#sphx-glr-gallery-text-labels-and-annotations-tex-demo-py` für weitere Details.

10.4.6. Rückgabewerte der Plot-Funktionen

Bei den Histogrammen haben wir schon gesehen, dass auch die Plot-Befehle selbst (`plot`, `bar`, `hist`, ...) Rückgabewerte haben können. Die Rückgabeobjekte sind im Detail jeweils verschieden; allen gemein ist jedoch, dass sie die Gesamtheit der gezeichneten Objekte repräsentieren.

Wir wollen hier zumindest auf den Rückgabewert von `plot` näher eingehen; von hier können Sie hoffentlich Analogien auf die anderen Objekte bilden, die Ihnen bei der Arbeit mit der MatPlotLib begegnen können.

Beginnen wir mit einem Code zur Analyse:

Beispiel: Rückgabewerte von `plt.plot`

```
1  import math
2  import matplotlib.pyplot as plt
3
4  X = [x / 100 for x in range(0, 628, 5)]
5  Y = [math.sin(2 * t) + 2 for t in X]
6
7  p = plt.plot(X, Y, "y-", label="foo")
8  q = plt.plot(X, Y, "r.", label="bar")
9
10 print(type(p), p[0], type(p[0]))
11 print(type(q), q[0], type(q[0]))
```

Ausgabe: Rückgabewerte von `plt.plot`

```
<class 'list'> Line2D(foo) <class 'matplotlib.lines.Line2D'>
<class 'list'> Line2D(bar) <class 'matplotlib.lines.Line2D'>
```

Wir sehen, dass der Rückgabewert von `plot` eine `list` von Instanzen der Klasse `matplotlib.lines.Line2D` ist. Jede solche Instanz repräsentiert nun wie erwähnt eine Linie des Graphen, d. h. enthält die Werte der Datenpunkte sowie die von uns explizit und implizit zugewiesenen Formatierungen (z. B. Name der Datenreihe `foo` bzw. `bar`, oder *Linienfarbe blau*). Wenn wir nun die zugehörige Seite der Dokumentation der MatPlotLib aufrufen (`https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.lines.Line2D.html`), finden wir eine große Zahl von Methoden, die auf dieses Objekt angewandt werden können, sowie eine kurze Erläuterung, welchen Zweck diese Methoden erfüllen, und welche Parameter erwartet werden.

Ein Anwendungszweck ist die Nutzung mit `legend`. Wie Sie wissen, sorgt `legend` dafür, dass die Legende zu einem Plot angezeigt wird. Im obigen Beispiel wird ein Plot vorbereitet, bei dem derselbe Datensatz einmal mit einer gelben Linie und einemal mit roten Punkten dargestellt wird. Würden wir hier direkt mit `plt.legend()` eine Legende erzeugen, so hätte diese auch zwei Einträge.

Um in einem solchen Fall den unnötigen Legendeneintrag zu unterdrücken, können wir dem Befehl `legend` zwei `list`s mitgeben. Die erste solche `list` enthält alle Handles auf darzustellende Linien (also alle `matplotlib.lines.Line2D`-Objekte); in die zweite `list` schreiben wir alle Legendeneinträge.

Beispiel: Legendeneinträge unterdrücken

```
1  import math
2  import matplotlib.pyplot as plt
3
4  X = [x / 100 for x in range(0, 628, 5)]
5  Y = [math.sin(2 * t) + 2 for t in X]
6
7  fig = plt.figure()
8  drw = fig.subplots()
9
10 p = drw.plot(X, Y, "y-", label="foo")
11 q = drw.plot(X, Y, "r.", label="bar")
12
13 drw.legend(p, ["FOOBAR"])
14 plt.show()
```

Legendeneinträge unterdrücken

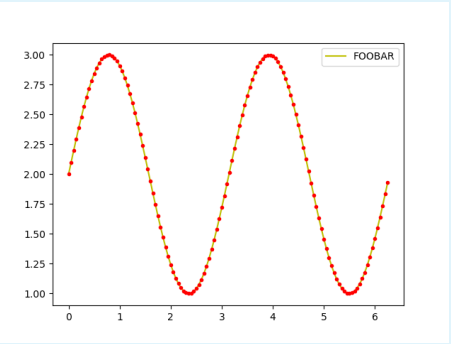


Abbildung 10.26.: Legendeneinträge unterdrücken

10.4.7. Text und Overlays

In ein Plotfenster können beliebige Zeichen- und Textelemente hinzugefügt werden. Hier soll nur gezeigt werden, wie Sie Textanmerkungen und Pfeile auf einem Plot anbringen. Für andere Elemente, siehe `https://matplotlib.org/3.1.1/gallery/text_labels_and_annotations/annotation_demo.html#sphx-glr-gallery-text-labels-and-annotations-annotation-demo-py`

Texte platzieren Sie in Ihrem Plot mit dem Befehl `annotate`. Hierzu müssen Sie angeben welcher Text geschrieben werden soll und an welchen Koordinaten er im Plot erscheinen soll; letzteres wird über den Schlüsselwort-Parameter `xy` als `tuple` angegeben. In seiner Minimalform lautet der Befehl dann also:

```
ax.annotate("text", xy=(x, y))
```

wobei `x, y` im Koordinatensystem des Plots sind.

Beispiel: Plot mit Text-Overlay

```
1  import math
2  import matplotlib.pyplot as plt
3
4  X = [x / 100 for x in range(0, 628, 5)]
5  Y = [math.sin(t) for t in X]
6
7  fig = plt.figure()
8  drw = fig.subplots()
9
10 drw.plot(X, Y)
11
12 drw.annotate("Die Sinus-Funktion,\n"      +
13             "wie von der math-library\n"+
14             "zur Verfügung gestellt",
15             xy=(3.5, 0.5)
16 )
17 plt.show()
```

Ausgabe: Plot mit Text-Overlay

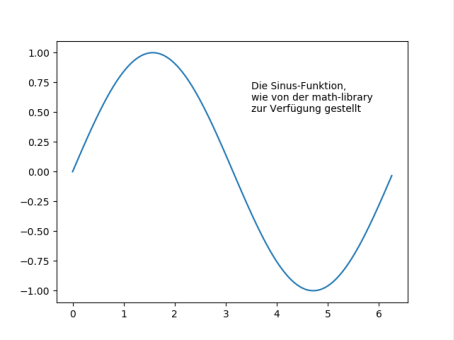


Abbildung 10.27.: Plot mit Text-Overlay

Manchmal ist es günstiger, die Koordinaten des Textfeldes nicht bezüglich des Plotfensters anzugeben, sondern ein anderes Koordinatensystem zu wählen. Andere mögliche Referenzpunkte können über den optionalen Paremeter `xycoords` festgelegt werden. Diesem wird einer der folgenden Strings zugewiesen:

String	Wirkung
`'figure points'`	Abstand vom linken oberen Rand des Plot-Fenstes in Punkten
`'figure pixels'`	Abstand vom linken oberen Rand des Plot-Fenstes in Pixeln
`'figure fraction'`	Abstand vom linken oberen Rand des Plot-Fenstes als Bruch der Gesamtbreite, d. h. `(0,0)` steht für die linke untere Ecke und `(1,1)` für doe rechte obere Ecke
`'axes points'`	Abstand vom Koordinatenursprung des Plots in Punkten
`'axes pixels'`	Abstand vom Koordinatenursprung des Plots in Pixeln
`'data'`	Koordinaten wie durch den Graphen definiert (default)

Tabelle 10.3.: Mögliche Koordinatenursprünge für `xycoords` in `ax.annotate`

Die Orientierung des Texts bezüglich des angegebenen Punkts kann weiter mit den optionalen Parametern `horizontalalignment` und `verticalalignment` gesteuert werden:

Parameter	erlaubte Werte
`horizontalalignment`	`'left'`, `'center'`, `'right'`
`verticalalignment`	`'top'`, `'center'`, `'bottom'`

Tabelle 10.4.: Mögliche Alignments für `ax.annotate`

Schließlich ist es auch möglich, mit dem Text auf einen bestimmten Aspekt im Plot hinzuweisen. Hierzu benutzen wir die optionalen Parameter `arrowprops`, `textxy` und `textcoords`.

Da wir nun zwei Dinge platzieren müssen – Text und Pfeilspitze – haben wir ein zusätzliches Koordinatenpaar `textxy`, das an die Stelle von `xy` im einfachen Fall tritt. D. h. die Pfeilspitze wird durch `xy` ausgezeichnet, während die Text-Koordinaten durch `textxy` beschrieben werden. Für `textcoords`

gilt dasselbe, was schon zu `xycoords` gesagt wurde, d. h. es handelt sich um Werte aus Tabelle 10.3. Zusätzlich sind auch noch diese Werte möglich:

String	Wirkung
`'offset points'`	Koordinaten relativ zu `xy` in Punkten
`'offset pixels'`	Koordinaten relativ zu `xy` in Pixeln

Tabelle 10.5.: Zusätzlich mögliche Koordinatenursprünge für `textcoords` in `ax.annotate`

Weiterhin beziehen sich `horizontalalignment` und `verticalalignment` auf die Ausrichtung des Texts.

Schließlich legt `arrowprops` fest, wie der Pfeil aussehen soll. Es handelt sich um ein `dict`, in dem verschiedene Schlüssel-Wert-Paare den Pfeil beschreiben. Eine (unvollständige!) Auswahl von möglichen Paaren finden Sie hier:

Schlüssel	Wert	Wirkung	Default
`'facecolor'`	(ein Farbwert) oder `'none'`	Füllfarbe des Pfeils	Standard-Blau
`'edgecolor'`	(ein Farbwert) oder `'none'`	Randfarbe des Pfeils	`'black'`
`'color'`	(ein Farbwert)	Rand- und Füllfarbe des Strings (überschreibt `facecolor` und `edgecolor`)	(nicht gesetzt)
`'shrink'`	(Zahl zwischen 0 und 1)	Verkürzt den Pfeil um angegebenen Faktor	1
`'width'`	(Zahl)	Breite des Pfeils in Punkten	4
`'frac'`			
`'headwidth'`	(Zahl)	Breite der Pfeilspitze in Punkte	10

Tabelle 10.6.: Mögliche Schlüssel-Wert-Paare für `arrowprops` in `ax.annotate`

Die Beispiele auf `https://matplotlib.org/3.1.1/gallery/text_labels_and_annotations/annotation_demo.html#sphx-glr-gallery-text-labels-and-annotations-annotation-demo-py` zeigen Ihnen weitere Einstellmöglichkeiten. Siehe auch `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.axes.Axes.annotate.html#matplotlib.axes.Axes.annotate` für weitere Details.

10.5. 3D-Plots

Auf einem zweidimensionalen Bildschirm (oder Ausdruck) kann natürlich nur eine zweidimensionale Projektion eines dreidimensionalen Objekts dargestellt werden. Dies schränkt die *Lesbarkeit* bzw. wissenschaftliche Verwertbarkeit von Visualisierungen von Daten ein; dafür erlauben Sie eine sehr viel intuitivere Wahrnehmung eines Sachverhalts. Und nicht zuletzt, 3D-Bilder sehen einfach *cool* aus.

Die MatPlotLib hält natürlich auch hierzu Methoden bereit. Wir wollen hier also auch einige Features der MatPlotLib ansprechen. Ein ausführlicheres Tutorial finden Sie unter `https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html`.

10.5.1. Das `Axes3D`-Objekt

Die Darstellung von 3D-Daten wird von einem eigenen Teil der MatPlotLib-Library geleistet, der zuerst in unser Projekt geladen werden muss. Dies geschieht über die Zeile:

```
from mpl_toolkits.mplot3d import Axes3D
```

Mit dieser Ergänzung können wir ein `Axes3D`-Objekt erzeugen, auf dem unsere 3D-Ausgaben erfolgen. Dieses Objekt erhalten wir über die Methode `add_subplot`, die wir bereits kennengelernt haben. Zusätzlich zu den früheren Aufrufen können wir jetzt den optionalen Parameter `projection='3d'` übergeben.

Beispiel: Leerer 3D-Plot

```
1  import matplotlib.pyplot as plt
2  from mpl_toolkits.mplot3d import Axes3D
3
4  fig = plt.figure()
5  drw = fig.add_subplot(projection='3d')
6
7  plt.show()
```

Ausgabe: Leerer 3D-Plot

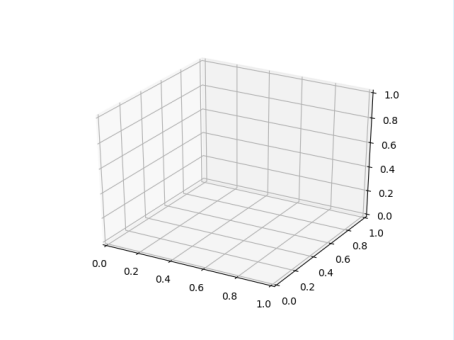


Abbildung 10.28.: Leerer 3D-Plot

Auf dieses `Axes3D`-Objekt können nun die bereits bekannten Methoden zum erstellen von Plots angewandt werden. Natürlich erwarten diese jetzt nicht nur Listen von X- und Y-Koordinaten, sondern auch die zugehörigen Z-Koordinaten. Grundsätzlich aber bleibt das Herangehen das Ihnen bereits bekannte. Im weiteren soll dies an zwei Beispielen genauer gezeigt werden.

10.5.2. Kurven im Raum

Sobald ein `Axes3D`-Objekt initialisiert wurde, kann die Methode `plot` wie gewohnt eingesetzt werden. Verlangt werden jetzt *3* Parameter, die die X- Y- und Z-Koordinaten enthalten. Ansonsten gelten die oben besprochenen Techniken weiter.

Einstellungen, die die dritte Achse betreffen, können über dieselben Methoden geleistet werden, die die ersten beiden Achsen betreffen, indem das `x` bzw. `y` durch ein `z` ersetzt wird. Die Achsenbeschriftung können wir beispielsweise durch `set_zlabel` ändern.

Um die Ansicht zu drehen kann der Befehl `view_init` benutzt werden. Dieser erwartet zwei Zahlen, die die Raumwinkel darstellen, in dem der Plot betrachtet wird. Die erste Zahl ist hierbei die *Inklination*, also wie steil „die Kamera“ auf den Plot gerichtet ist; die zweite Zahl dagegen ist die *Deklination*, also die Drehung in der X-Y-Ebene. Experimentieren Sie im Zweifel herum, um die für Sie geeigneten Einstellungen zu finden.

Beispiel: 3D-Kurve

```
import math
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
drw = fig.add_subplot(projection='3d')

T = [math.pi * t / 100 for t in range(1000)]
X = [math.exp(-0.05 * t) * math.cos(t) for t in T]
Y = [math.exp(-0.05 * t) * math.sin(t) for t in T]

drw.set_xlabel("x")
drw.set_ylabel("y")
drw.set_zlabel("time t")
drw.set_title("decaying orbit")
drw.view_init(80, 10)
drw.plot(X, Y, T)
plt.show()
```

Ausgabe: 3D-Kurve

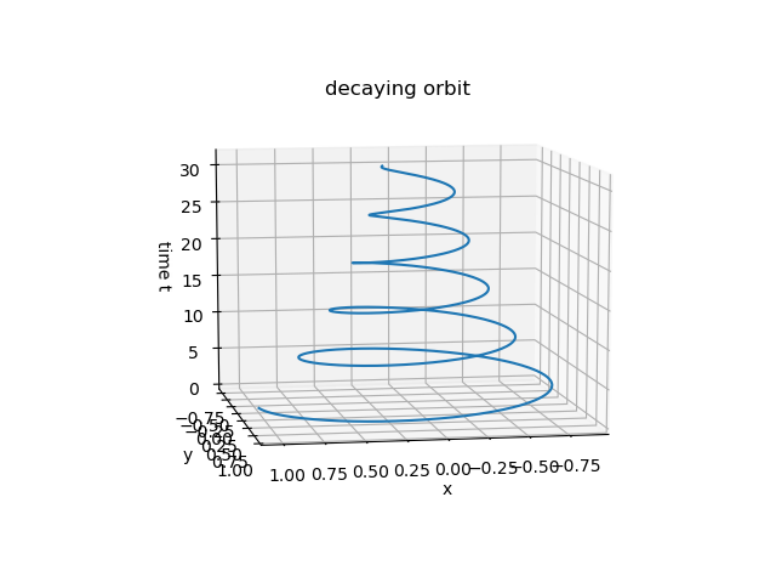


Abbildung 10.29.: 3D-Kurve

10.5.3. Plots von Flächen

Für die Darstellung von Flächen im Dreidimensionalen muss die Zuordnung von X-Y-Koordinaten zur Z-Koordinate derselben Logik folgen wie bei den Vektorfeldern (siehe Seite 157): Wir brauchen zwei *zweidimensionale Arrays* `X, Y`, die so das *Meshgrid* aufspannen. `X[i][j]` enthält dann die X-Koordinate

des `i`-ten Punkts in Zeilenrichtung und des `j`-ten Punkts in Spaltenrichtung. Gleiches gilt für `Y` bezüglich der Y-Koordinaten.

Die zugeordneten Daten `Z` müssen als *zweidimensionales NumPy-Array* vorliegen. Im Detail wird das Modul `numpy` erst in Kapitel 11 besprochen; es kann jedoch direkt aus einer *zweidimensionalen* `list` über den Konstruktor von `numpy.array` erzeugt werden.

Der folgende Code erzeugt die Daten, die wir in den weiteren Beispielen für die Plots verwenden werden:

Beispiel: Datenerzeugung: Sattelfläche

```
import numpy as np

W = 20
H = 30

func = lambda x, y : x**2 - y**2

X = [[j/10 - 1.0 for j in range(W)] for i in range(H)]
Y = [[i/10 - 1.5 for j in range(W)] for i in range(H)]
Z = [[None for i in range(W)] for j in range(H)]

for   i in range(H) :
  for j in range(W) :
    x = X[i][j]
    y = Y[i][j]
    Z[i][j] = func(x, y)

Z=np.array(Z)
```

Sneak Preview: NumPy

Die Plot-Daten können mit dem Modul `numpy` auch so erzeugt werden:

Beispiel: Datenerzeugung: Sattelfläche mit `numpy`

```
import numpy as np

X, Y = np.meshgrid(np.linspace(-1, 1, 20), np.linspace(-1.5, 1.5, 30))
Z = X**2 - Y**2
```

In Kapitel 11 wird dies im Detail besprochen.

Wireframes

Wireframes oder Drahtgittermodelle zeichnen Datenpunkte jeweils verbunden mit ihren nächsten Nachbarn. Die Flächen zwischen diesen Verbindungslinien bleiben transparent. Der Befehl zum Erstellen eines Drahtgittermodells ist `plot_wireframe(X, Y, Z)`:

Beispiel: Drahtgittermodell

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
drw = fig.add_subplot(
    111,
    projection='3d'
)

drw.plot_wireframe(X, Y, Z)

plt.show()
```

Ausgabe: Drahtgittermodell

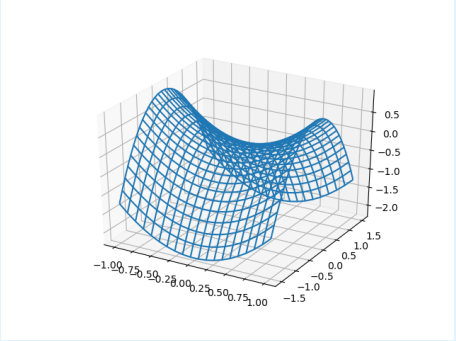


Abbildung 10.30.: Drahtgittermodell

Siehe auch `https://matplotlib.org/3.1.1/api/_as_gen/mpl_toolkits.mplot3d.axes3d.Axes3D.html?highlight=plot_wireframe` für weitere Details.

Surface Plots

Surface Plots oder ausgefüllte Drahtgittermodelle erreichen sie, indem Sie den Befehl `plot_wireframe` durch `plot_surface` ersetzen. Er folgt denselben Regeln wie `plot_wireframe`, unterstütz jedoch andere optionale Parameter, die Sie unter `https://matplotlib.org/api/_as_gen/mpl_toolkits.mplot3d.axes3d.Axes3D.html?highlight=plot_surface` nachschlagen können. Unterstützt werden auch Schattierungs und Transparenz-Effekte.

Beispiel: Ausgefülltes Drahtgittermodell

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
drw = fig.add_subplot(
    111,
    projection='3d'
)

drw.plot_surface(X, Y, Z)

plt.show()
```

Ausgabe: Ausgef. Drahtgittermodell

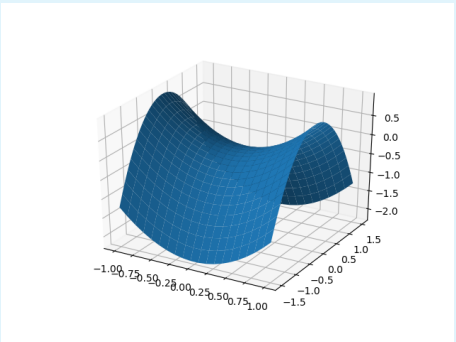


Abbildung 10.31.: Ausgefülltes Drahtgittermodell

Flache 3D-Darstellung mit Falschfarben und als Kontourplot

Wie schon angesprochen können 3D-Darstellungen oft schwerer zu lesen sein. Teile des Plots sind „hinter“ anderen verborgen, und auf den schräg über die Zeichenebene verlaufenden Achsen sind Werte oft schwerer exakt abzulesen. Eine beliebte Alternative sind daher Falschfarben-Darstellungen und Konturlinien.

Mit der Methode `pcolor` lässt sich eine einrache Falschfarbendarstellung dreidimensionaler Daten erreichen. Wie schon bei `plot_wireframe` und `plot_surface` müssen nur die X-, Y- und Z-Daten übergeben werden. Jeder Punkt in der X-Y-Ebene wird durch ein Rechteck in einer bestimmten Farbe dargestellt. Die Farbe richtet sich dabei nach dem Z-Wert. Der Rückgabewert von `pcolor` ist ein `matplotlib.collections.PolyCollection`-Objekt und kann u. a. dazu genutzt werden, um eine Farblegende zu erstellen. Hierzu dient die Methode `colorbar`. Sie wirkt auf ein `Figure`-Objekt, und fügt eine Farblegende zum zuletzt auf dieser `Figure` gezeichneten Sublot eine Farbskala hinzu. Als Parameter wird dabei ein Objekt erwartet, aus dem diese Farbskala erzeugt werden kann – beispielsweise eben der Rückgabewert von `pcolor`.

Die Methode `contour` zeichnet aus den X- Y- und Z-Daten eine Karte von „Höhenlinien“, d. h. Linien, entlang derer der Z-Wert konstant bleibt. „Zwischen“ den Gitterpunkten wird interpoliert.

Die Methode `contourf` schließlich erzeugt ebenfalls Höhenlinien, füllt aber die Flächen zwischen diesen Linien farbig aus. Lesen Sie diesen Befehl also am besten als *contour-filled*. Auf diese Art lässt sich eine „glattere“ Darstellung als mit `pcolor` erreichen. Natürlich ist dies nicht immer auch so erwünscht.

Weitere Details zu den Methoden finden Sie unter

- `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.axes.Axes.pcolor.html`
- `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.contour.html`
- `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.contourf.html`

Beispiel: Darstellung in Falschfarben und als Kontourplot

```
1  import matplotlib.pyplot as plt
2
3  fig = plt.figure(figsize=(12,4))
4  grd = fig.add_gridspec(1, 7)
5
6  drw = fig.add_subplot(grd[0,0:2])
7  drw.set_title("pcolor")
8  col = drw.pcolor(X, Y, Z)
9
10 drw = fig.add_subplot(grd[0,2:4])
11 drw.set_title("contour")
12 drw.set_yticks([])
13 drw.contour(X, Y, Z)
14
15 drw = fig.add_subplot(grd[0,4:6])
16 drw.set_title("contourf")
17 drw.set_yticks([])
18 drw.contourf(X, Y, Z)
19
20 drw = fig.add_subplot(grd[0,6])
21 drw.axis("off")
22 fig.colorbar(col)
23
24 plt.show()
```

Ausgabe: Darstellung in Falschfarben und als Kontourplot

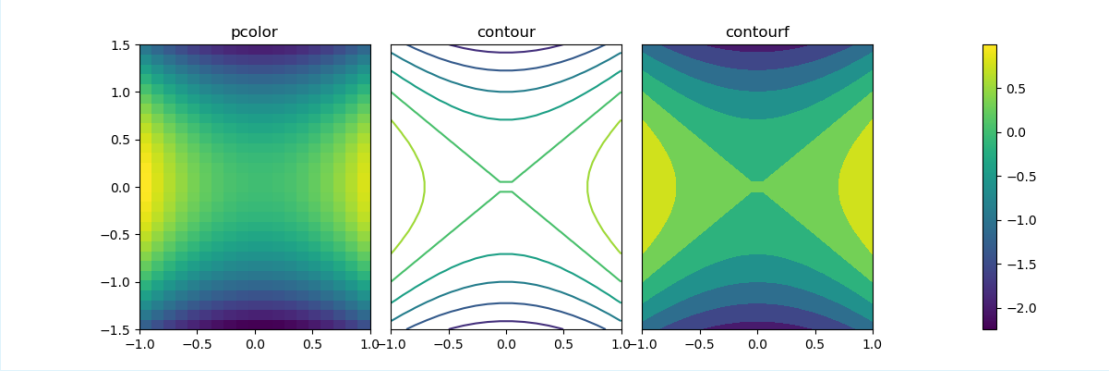


Abbildung 10.32.: Darstellung in Falschfarben und als Kontourplot

10.6. Ausgabe in Dateien

Die Graphen, die Sie mit der MatPlotLib erzeugen, können Sie auch direkt als Grafik-Dateien abspeichern. Hierzu dient der Befehl `plt.savefig` bzw. die Methode `savefig`.

Der Methode *muss* ein Dateiname als erster Parameter übergeben werden. Eine Datei mit entsprechendem Namen wird angelegt, falls sie noch nicht existiert; sollte eine Datei mit dem angegebenen Namen bereits existieren, wird diese ohne weitere Nachfrage überschrieben.

Optional kann auch mit dem Parameter `format` ein String übergeben werden, der das Dateiformat festlegt. Unterstützt werden die Formate eps, pdf, pgf, png, ps, raw, rgba, svg und svgz. Der Default-Wert ist `"png"`; jedoch versucht die MatPlotLib aber auch, aus dem Dateinamen ein geeignetes Format abzuleiten. Wird kein Format explizit angegeben, wird die Erweiterung des Dateinamens zur Bestimmung des Dateityps herangezogen. Beispielsweise führt `fig.savefig("filename.pdf")` eine PDF mit dem Namen `filename.pdf` erzeugen. Dagegen liefert `fig.savefig("filename.pdf", format="png")` dazu, dass eine PNG mit dem Dateinamen `filename.pdf` geschrieben wird.

Sehen Sie sich auch die Dokumentation unter `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.figure.Figure.html?highlight=savefig#matplotlib.figure.Figure.savefig` an für weitere Details.

Speicherbedarf

In Python ist es sehr leicht, große Datenmengen zu generieren, und damit sogar zeitweise seinen Rechner komplett auszulasten. Dies sollte insbesondere bedacht werden, wenn mit Plots gearbeitet wird.

Die MatPlotLib speichert intern keine Bilder, sondern wirklich die Rohdaten. Das bedeutet, dass zu jedem Punkt einer Kurve sowohl X- als auch Y-Koordinate gespeichert werden müssen. Da diese Informationen i. d. R. als `float` vorliegen, sind dies also mindestens 16 Byte pro Datenpunkt. Ggf. kommen zusätzliche Daten für Formatierungen etc. hinzu. Eine Kurve mit 1 000 000 Datenpunkten (nicht ungewöhnlich für wissenschaftliche Simulationen) belegt also bereits 16 MB im Arbeitsspeicher, auch wenn das daraus errechnete Bild nur wenige 100 kB belegt.

Weiter liegen die Daten oft doppelt im Arbeitsspeicher vor. Betrachten Sie folgenden Code-Auszug:

Beispiel: Daten-Dopplung

```
import matplotlib.pyplot as plt

X = [x * .001 for x in range(1000000)]
Y = [0]
for x in X[1:] :
    Y.append( (Y[-1] ** 2 - .0005 * x) % 2 )

plt.plot(X, Y, ",")
plt.show()
```

Die `lists` `X, Y` werden durch den Befehl `plot` tatsächlich in einen von der MatPlotLib verwalteten Speicherbereich *kopiert*! Kurzfristig verbrauchen wir hier also 32 MB an Arbeitsspeicher für eine einzige Kurve.

Wenn wir verschiedene Parameter in unserer Simulation durchtesten, kann der Speicherbedarf schnell anwachsen. Anforderungen im Gigabyte-Bereich sind leicht erreicht.

Um dieses Problem zu umgehen, können wir versuchen, unsere Simulation so aufzubauen, dass sie Daten nicht am Stück generiert, sondern jeweils in kleineren Batches. Aus diesen Batches wird nur eine Untermenge ausgewählt, die tatsächlich geplottet wird; der Rest dagegen wird wieder verworfen:

Beispiel: Batch-Weise Berechnung und reduziertes Plotten

```
import matplotlib.pyplot as plt

Y = [0]
for batch in range(1000) :
    X = [x * .001 for x in range(batch * 1000, (batch + 1) * 1000)]
    for x in X[1:] :
        Y.append( (Y[-1] ** 2 - .0005 * x) % 2 )

    plt.plot(X[::50], Y[::50], "b,")

    Y = [(Y[-1] ** 2 - .0005 * x) % 2]
plt.show()
```

Im zweiten Beispiel sinkt der Speicherbedarf vom 32 MB auf nur knapp über 46 kB. Manchmal bringt ein solches Reduzieren der zu plottenden Datenmenge zusätzlich den Vorteil, dass die innere Struktur der Plots leichter erkennbar wird, wie die Gegenüberstellung der beiden Ausgaben zeigt:

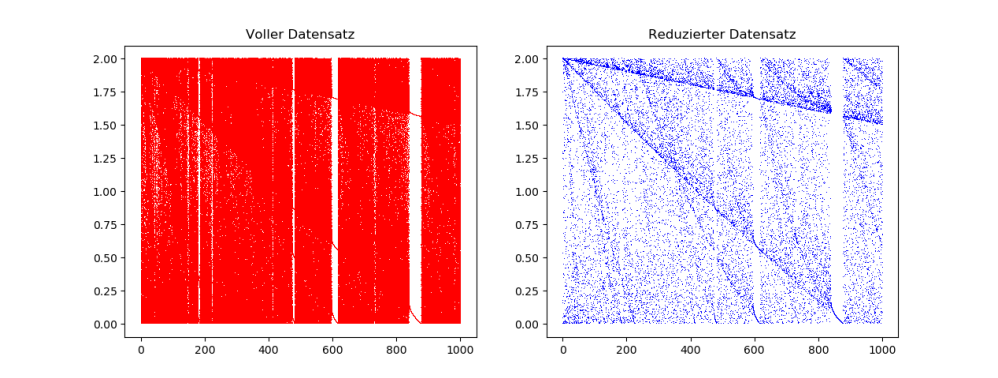


Abbildung 10.33.: Deterministisches Chaos: Voller Datensatz und Reduzierter Datensatz

Das Beispiel ist lose angelehnt an die Erkenntnisse, die ich aus dem Kurs *Chaos and Nonlinear Dynamics* an der Universität Regensburg gewonnen habe.

11. NumPy

> The only thing that remains unsolved is the resolution of the problem.
>
> Thomas Wells

NumPy ist ein Python-Modul mit vielen Einsatzgebieten. Die Dokumentation des Pakets[1] wird selbstbewusst eingeleitet mit:

NumPy is the fundamental package for scientific computing in Python.

Nicht nur werden Routinen für viele häufig auftauchende Aufgaben bereitgestellt; die Umsetzung dieser ist auch bedeutend schneller, als es in „Vanilla-Python“ möglich wäre, da die Kernroutinen in kompilierten C-Libraries vorliegt. Wir erhalten also nicht nur zusätzlichen Komfort, sondern auch einen massiven Speedup bei der Verwendung von NumPy-Funktionen.

Installation

Nicht auf allen Systemen ist NumPy sofort vorinstalliert. Um zu testen, ob Ihr System bereits die NumPy-Bibliothek unterstützt, können Sie diesen Code aufführen:

Installation von NumPy testen

```
1 try:
2     import numpy
3 except ImportError:
4     print("numpy is not installed")
5 else :
6     print("numpy is ready for use")
```

Sollte die Zeile `numpy is not installed` erscheinen, so können Sie dies einfach nachinstallieren. Rufen Sie hierzu entweder die Anaconda-Konsole auf (Windows, Mac) oder ein reguläres Terminal (Linux). Hier geben Sie die folgende Zeile (Windows, Mac) ein:

NumPy Nachinstallieren – Windows, Mac

```
pip install numpy
```

oder, wenn Sie unter Linux arbeiten:

NumPy Nachinstallieren – Linux

```
pip3 install numpy
```

[1] `https://numpy.org/doc/stable/user/whatisnumpy.html`

Hier kann nur auf die wichtigsten Features eingegangen werden; eine vollständige Referenz finden Sie unter `https://numpy.org/doc/stable/numpy-ref.pdf` (1812 Seiten, Stand Juni 2020).

11.1. Grundlegendes Objekt

Für die Arbeit mit NumPy muss natürlich zuerst das Modul geladen werden. Per Konvention geschieht dies mit dem Alias `np`:

Modul NumPy Laden

```
import numpy as np
```

Auch wenn nicht explizit in allen Code-Beispielen gezeigt, soll in diesem Kapitel immer davon ausgegangen werden, dass dieser `import` die erste Zeile aller Beispiele sein soll.

Zentrales Objekt in allen NumPy-Anwendungen ist das *Numpy-Array*. In erster Näherung verhält es sich wie eine Ihnen bereits bekannte `list`: NumPy-Arrays sind Container von geordneten Werten, d. h. ein Wert im Container kann durch seinen *Index* eindeutig identifiziert werden. Sie können sowohl eindimensionale als auch mehrdimensionale Objekte darstellen, also beispielsweise Tabellen abbilden. Über NumPy-Arrays kann auch iteriert werden, d. h. es ist möglich sie mit einer `for`-Schleife zu durchlaufen.

Im Gegensatz zu Python-`list`s müssen aber *alle Einträge* in einem Numpy-Array *denselben Datentyp* haben[2]. Bei Tabellen und Tensoren müssen alle Einträge angegeben werden; es ist also nicht möglich, eine Tabelle mit 5 Zeilen in der ersten Spalte und 10 Zeilen in der zweiten Spalte anzulegen.

Erzeugt werden NumPy-Arrays über ihren Konstruktor aus normalen Python-`list`s[3]. Wo versucht wird, die Einschränkungen auf gleichen Datentyp oder ausgefüllte Tabellen zu missachten, wendet NumPy verschiedene Strategien an, um ein kompatibles Objekt zu erzeugen:

Beispiel: NumPy-Arrays erzeugen

```
1  import numpy as np
2
3  npList   = np.array([1, 2, 3])
4  npMixed1 = np.array([1, 1.5])
5  npMixed2 = np.array([1, 1.0, "1.0"])
6
7  npTab      = np.array([[1, 2, 3], [4, 5, 6]])
8  npListList = np.array([[1], [2, 3]])
9
10 print(npList)
11 print(npMixed1)
12 print(npMixed2)
13 print(npTab)
14 print(npListList)
```

[2] Diese Einschränkung macht die zugründeliegende Speicherstruktur sehr viel einheitlicher und ist mit für den großen Geschwindigkeits-Gewinn verantwortlich, den wir mit NumPy erhalten. In der Praxis schränkt uns dies kaum ein, da wir NumPy ohnehin nur für ähnliche Daten verwenden.

[3] Genauer: Aus einem *Iterable*. Das bedeutet, dass auch `tuple`s, ... benutzt werden können, um NumPy-Arrays zu erzeugen.

Ausgabe: NumPy-Arrays erzeugen

```
[1 2 3]
[1.  1.5]
['1' '1.0' '1.0']
[[1 2 3]
 [4 5 6]]
[list([1]) list([2, 3])]
```

Wir sehen zunächst, dass die aus `int`s bestehende `list [1, 2, 3]` direkt als NumPy-Array übernommen werden kann (erste Zeile der Ausgabe). Für die aus einem `int` und einem `float` bestehende `list [1, 1.5]` rechnet NumPy beide Einträge zu `float`s um[4]. Die `list [1, 1.0, "1.0"]` enthält einen String; da nicht jeder String in eine Zahl umgewandelt werden kann, haben die Entwickler von NumPy sich dazu entschieden, dass in so einem Fall *alle Einträge* zu Strings umgewandelt werden.

Im Falle von Tabellen wie der `list [[1, 2], [3, 4]]` werden die verschachtelten Listen in das interne Schema der NumPy-Bibliothek heruntergebrochen. In diesem Format kann die Tabelle auch „schön“ als Tabelle ausgegeben werden – ein einziger `print`-Befehl erzeugt mehrere Zeilen auf dem Bildschirm.

Dagegen ist `list [[1], [2, 3]]` keine vollständig ausgefüllte Tabelle; NumPy fasst sie daher als Array von `list`s auf. Die Ausgabe spiegelt genau dies wieder.

11.1.1. Datentypen und Attribute des NumPy-Arrays

Der Speicherbedarf für eine Zahl wird in Python dynamisch angepasst. Für kleine Zahlen werden nur einige wenige Bytes verwendet, größere Zahlen beanspruchen einen langen zusammenhängenden Speicherbereich. Hinzu kommt ein *Descriptor*, also ein Speicherabschnitt, in dem deklariert wird, wie viel Speicherplatz zur Darstellung der Zahl verwendet wird, und wo dieser zu finden ist. Der `int 4` beansprucht z. B. 28 Bytes; für den `int 2**20` dagegen sind es 32 Bytes und für die Zahl `2**128` werden 44 Bytes belegt. Diese Flexibilität kommt zum Preis längerer Laufzeiten.

In NumPy hingegen wird für alle Zahlen derselbe Speicherplatz zur Verfügung gestellt, der sich aus dem Datentypen herleitet. Für einen `np.int64` – die häufigste Übersetzung für einen Python-`int` – werden *immer* 8 Bytes genutzt. Es ist offensichtlich, dass 8 Bytes schneller zu verarbeiten sind als 28. Dafür aber kann es beim Rechnen mit NumPy zu sogenannten *oveverflows* kommen: Das Ergebnis einer Berechnung kann zu groß sein, um im bereitgestellten Speicherbereich abgelegt werden zu können. Tabelle 11.1 enthält eine Übersicht der wichtigsten bereitgestellten Typen. Wie Sie an den aufgelisteten Wertebereichen sehen können, sind die Beschränkungen der Wertebereiche selten von Belang.

Unter `https://numpy.org/doc/stable/reference/arrays.dtypes.html` finden Sie weitere Details zum Umgang mit den Numpy-Datentypen.

Im Folgenden wollen wir immer dieselben Arrays in den Beispielen betrachten. Diese seien definiert durch:

Beispiel: Ausgangsarrays für die folgenden Beispiele

```
1  import numpy as np
2  import math
3
4  npList = np.array([1, 2, 3])
5  npTab  = np.array([[1.1, 2.2, 3.3], [4.4, 5.5, 6.6]])
```

[4] Genauer zu `np.float64`s – siehe etwas später dazu.

Übliche NumPy-Datentypen

NumPy-Typ	Beschreibung	Wertebereich	Entsprechung in C-Artigen Sprachen
`np.int8`	Ganzzahl	-128 bis +127	`char`
`np.int16`	Ganzzahl	-32 768 bis +32 767	`short`
`np.int32`	Ganzzahl	-2 147 483 648 bis +2 147 483 647	`long`
`np.int64`	Ganzzahl	-9 223 372 036 854 775 808 bis +9 223 372 036 854 775 807	`long long`
`np.uint8`	Ganzzahl	0 bis +255	`unsigned char`
`np.unit16`	Ganzzahl	0 bis +65 536	`unsigned short`
`np.unit32`	Ganzzahl	0 bis +4 294 967 295	`unsigned long`
`np.unit64`	Ganzzahl	0 bis +18 446 744 073 709 551 615	`unsigned long long`
`np.float32`	Fließkommazahl	ca. -3.4×10^{38} bis $+3.4\times10^{38}$, ca. 7 signifikante Ziffern	`float`
`np.float64`	Fließkommazahl	ca. -1.7×10^{308} bis $+1.7 \times 10^{308}$, ca. 15 signifikante Ziffern	`double`
`np.complex64`	Komplexe Zahl	Real und Imaginärteil jeweils wie `np.float32`	`float complex`
`np.complex128`	Komplexe Zahl	Real und Imaginärteil jeweils wie `np.float64`	`double complex`

Tabelle 11.1.: Datentypen in NumPy
Siehe auch https://numpy.org/doc/stable/user/basics.types.html für weitere Details.

Der Datentyp der Elemente eines NumPy-Arrays kann über das Attribut `dtype` abgefragt werden:

Beispiel: Zugrundeliegender Datentyp

```
5  # ...
6  print(npList.dtype)
7  print(npTab.dtype)
```

Ausgabe: Zugrundeliegender Datentyp

```
int64
float64
```

Ebenso wie der Datentyp kann auch die *Form* eines NumPy-Arrays mit `shape` abgefragt werden. Die „Antwort“ ist en `tuple` mit der Zahl der Elemente in jeder Dimension:

Beispiel: Form eines NumPy-Arrays

```
8  # ...
9  print(npList.shape)
10 print(npTab.shape)
```

Ausgabe: Form eines NumPy-Arrays

```
(3,)
(2, 3)
```

Die *Gesamtzahl der Elemente* in einem NumPy-Array und die *Gesamtzahl der Dimensionen* sind in den `int`-Attributen `size` und `ndim` gespeichert. Alternativ könnte man diese Informationen auch aus `shape` gewinnen:

Beispiel: Anzahl Elemente & Dimensionen

```
11  # ...
12  print(npList.size, math.prod(npList.shape))
13  print(npTab .size, math.prod(npTab.shape))
14  print()
15  print(npList.ndim, len(npList.shape))
16  print(npTab .ndim, len(npTab.shape))
```

Anzahl Elemente & Dimensionen

```
3 3
6 6

1 1
2 2
```

Schließlich kann für die Anbindung an C-Bibliotheken auch die Speicheradresse in Erfahrung gebracht werden, wo die Daten des NumPy-Arrays abgelegt sind. Hierzu dient das Attribut `data`. Der Rückgabewert ist ein `MemoryView`-Objekt, das an dieser Stelle nicht näher besprochen werden soll.

Beispiel: Adresse eines Arrays

```
17  # ...
18  print(npList.data)
```

Ausgabe: Adresse eines Arrays

```
<memory at 0x7fa439743b80>
```

Bei komplexwertigen Arrays kann der Real- und Imaginärteil für alle Elemente als eigenes Array über die Attribute `real` und `imag` abgegriffen werden.

11.2. Indices und NumPy-Arrays

NumPy-Arrays können genauso indiziert werden wie Python-`list`s. Im Gegensatz zu diesen bietet NumPy aber einigen zusätzlichen Komfort.

Bei mehrdimensionalen Listen kann ein einzelnes Element angesprochen werden, indem ein `tuple` mit seinen Koordinaten übergeben wird. Für das NumPy-Array `a` ist also der Zugriff `a[i, j]` gleichbedeutend mit dem Zugriff `a[i][j]`. Ersteres dürfte den meisten ProgrammiererInnen als leichter zu Tippen und zu Lesen erscheinen.

Wie Python-`list`s unterstützen auch NumPy-Arrays Slicing. Dies kann auch mit der Multi-Index-Schreibweise kombiniert werden, d. h. Slices können durch Kommata voneinander getrennt werden, um einen Unter-Block des NumPy-Arrays anzugeben

Beispiel: Einfache Index-Zugriffe

```
1   npTab = np.array(
2       [[ 1,  2,  3],
3        [ 4,  5,  6],
4        [ 7,  8,  9],
5        [10, 11, 12]]
6   )
7
8   print("entire table:\n", npTab)
9   print()
10  print("element (0, 1):", npTab[0, 1])
11  print("last row:", npTab[-1])
12  print("column 1:", npTab[:,1])
13  print("upper left square:\n", npTab[0:2, 0:2])
```

Ausgabe: Einfache Index-Zugriffe

```
entire table:
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]

element (0, 1): 2
last row: [10 11 12]
column 1: [ 2  5  8 11]
upper left square:
 [[1 2]
 [4 5]]
```

Weiter kann auch ein NumPy-Array oder eine Python-`list` oder ein NumPy-Array als Index-Liste übergeben werden. Jedes Element des NumPy-Arrays wird dann als Index behandelt. Hat das Index-Array mehrere Dimensionen, so wird für jede Dimension eine eigene Ergebnis-Liste angelegt:

Beispiel: Zugriffe mit Index-Arrays

```
14  # ...
15
16  npIdcs = np.array([1, -1])
17  print(npTab[npIdcs])
18
19  npTup = np.array([[1, 2], [2, 1]])
20  print(npTab[npTup])
```

Ausgabe: Zugriffe mit Index-Arrays

```
[[ 4  5  6]
 [10 11 12]]
[[[4 5 6]
  [7 8 9]]

 [[7 8 9]
  [4 5 6]]]
```

Das Index-Array darf auch aus `bool`s bestehen. In diesem Fall müssen Index-Array und des NumPy-Array dieselbe Länge haben. Zurückgemeldet werden diejenigen Elemente des NumPy-Arrays, für die im Index-Array `True` eingetragen war. Man spricht auch von Indexmasken:

Beispiel: Indexmasken

```
22  # ...
23  mask = [True, False, False, True]
24  print(npTab[mask])
```

Ausgabe: Indexmasken

```
[[ 1  2  3]
 [10 11 12]]
```

Ein spezieller Index ist der Wert `np.newaxis`: Mit diesem kann eine neue Dimension in ein NumPy-Array eingeführt werden:

Beispiel: Neue Dimension Einführen

```
25  # ...
26
27  print( npList[np.newaxis] )
28  print( npList )
29  print()
30  print( npTab[np.newaxis] )
31  print( npTab[:,np.newaxis] )
```

Ausgabe: Neue Dimension Einführen

```
[[1 2 3]]
[1 2 3]

[[[ 1  2  3]
  [ 4  5  6]
  [ 7  8  9]
  [10 11 12]]]
[[[ 1  2  3]]

 [[ 4  5  6]]

 [[ 7  8  9]]

 [[10 11 12]]]
```

Ausgehend von der eindimensionalen `npList` erzeugt `npList[np.newaxis]` also ein zweidimensionales Objekt, also eine Tabelle mit einer Zeile und drei Spalten. Dieses Objekt wird neu erzeugt, d. h. das originale Objekt `npList` wird also nicht verändert. Bei mehrdimensionalen Objekten gibt die Position von `np.newaxis` in der Index-Liste an, „in welche Richtung“ die neue Dimension eingefügt wird. So ist `npTab[np.newaxis]` ein 1x4x3-Tensor und kann als „Liste von 4x3-Tabellen mit einem Eintrag“ interpretiert werden; dagegen ist `npTab[:,np.newaxis]` ein 4x1x3-Tensor und kann als „Liste von 1x3-Tabellen mit vier Einträgen“ interpretiert werden.

11.3. Methoden zum schnellen Erstellen von NumPy-Arrays

Bestimmte Formen von Arrays kehren immer wieder. Die wichtigsten davon seien hier zusammengefasst. Siehe auch `https://numpy.org/doc/stable/reference/routines.array-creation.html` für eine ausführliche Liste.

11.3.1. Gleichmäßig ansteigende Folgen

Von Python kennen wir bereits den Befehl `range`, der Folgen von `int`s erzeugt. NumPy erlaubt, auch Arrays anderer Datentypen zu erzeugen.

Die direkte Entsprechung von `range` ist der Befehl `arange`: Er kann entweder in der Form `np.arange(Start, Stop, Schrittweite)` oder `np.arange(Stop)` benutzt werden, und erzeugt ein NumPy-Array, das die Werte von `Start` bis ausschließlich `Stop` enthält. Im Falle der zweiten Syntax werden die Zahlen von `0` bis ausschließlich `Stop` in Abständen von `1` erzeugt, wie bei `range`. Anders als dort ist die Ausgabe aber kein Generator-Objekt, sondern ein NumPy-Array. Außerdem darf `Schrittweite` auch eine Nicht-Ganzzahl sein. Das Optionale Argument `dtype` kann außerdem dazu genutzt werden, um den Datentyp der Array-Elemente festzulegen:

Beispiel: `arange`

```
1  arr1 = np.arange(5)
2  arr2 = np.arange(5, 10, dtype=np.float64)
3  arr3 = np.arange(0, 2, 0.25)
```

```
print(arr1.dtype, arr1, sep='\t')
print(arr2.dtype, arr2, sep='\t')
print(arr3.dtype, arr3, sep='\t')
```

Ausgabe: `arange`

```
int64   [0 1 2 3 4]
float64 [5. 6. 7. 8. 9.]
float64 [0.   0.25 0.5  0.75 1.   1.25 1.5  1.75]
```

Ganz ähnlich arbeitet `linspace`. Im Gegensatz zur Schrittweite wird hier jedoch die *Anzahl der Elemente im Array* angegeben. Lässt man diese Angabe aus, geht NumPy automatisch von 50 Array-Elementen aus. Außerdem wird der Wert `Stop` hier *mit eingeschlossen*. Ein `Start`-Wert *muss* angegeben werden. Auch hier kann der optionale Parameter `dtype` angegeben werden. Standard-Typ ist `np.float64`.

Beispiel: `linspace`

```
arr1 = np.linspace(0, 4.90)
arr2 = np.linspace(0, 2, 11)

print(arr1.dtype, arr1, sep='\t')
print(arr2.dtype, arr2, sep='\t')
```

Ausgabe: `linspace`

```
float64 [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.  1.1 1.2 1.3 1.4 1.5 1.6 1.7
 1.8 1.9 2.  2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.  3.1 3.2 3.3 3.4 3.5
 3.6 3.7 3.8 3.9 4.  4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9]
float64 [0.  0.2 0.4 0.6 0.8 1.  1.2 1.4 1.6 1.8 2. ]
```

In manchen Situationen braucht man keine Werte, die durch einen konstanten Abstand gegeben sind, sondern durch einen konstanten Faktor. Zwischen einem Element `a[i]` und `a[i + 1]` soll also immer die Beziehung `a[i + 1] = factor * a[i]` gelten. Dies wird sehr leicht über die Befehle `geomspace` und `logspace` erreicht.

Wie auch bei `linspace` werden bei `geomspace` die Werte `Start`, `Stop` und `Anzahl` angegeben. Bei Auslassen von `Anzahl` geht NumPy auch wieder von `50` aus. Auch hier kann der optionale Parameter `dtype` angegeben werden.

Beispiel: `geomspace`

```
arr = np.geomspace(2, 1024, 10)
print(arr.dtype, arr)
```

Ausgabe: `geomspace`

```
float64 [   2.    4.    8.   16.   32.   64.  128.  256.  512. 1024.]
```

Der Befehl `logspace` arbeitet nach demselben Schema wie `geomspace`; Angegeben werden jedoch nicht der Start- und Stop*wert*, sondern die *Exponenten* sowie die Basis `base` mit dem Default-Wert `10.0`. Zu den Parametern gehören auch wieder `Anzahl=50` und `dtype=np.float64`

Beispiel: `logspace`

```
arr1 = np.logspace(0, 4, 5)
arr2 = np.logspace(0, 10, 11, base=2, dtype=np.int32)

print(arr1.dtype, arr1)
print(arr2.dtype, arr2)
```

Ausgabe: `logspace`

```
float64 [1.e+00 1.e+01 1.e+02 1.e+03 1.e+04]
int32 [   1    2    4    8   16   32   64  128  256  512 1024]
```

11.3.2. Spezielle Matrizen und Tensoren

Die NumPy-Funktionen `zeros`, `ones` und `full` erzeugen jeweils Listen, Matrizen oder Tabellen, in denen alle Einträge gleich sind. Wie zu erwarten, erzeugt `zeros` ein NumPy-Array aus Nullen, `ones` eines mit Einsen, und `full` eines, bei dem alle Einträge denselben, frei wählbaren Wert haben. Ihnen gleich ist der erste Parameter `shape`, der entweder ein `int` sein muss, und dann die Länge der zu erzeugenden Liste enhtält, oder ein `tuple` von `int`s, der jeweils die Ausmaße der einzelnen Dimensionen enthält. Bei `full` kommt als zweiter Parameter der Wert hinzu, der in das NumPy-Array eingetragen werden soll. Alle drei Befehle „verstehen“ den optionalen Parameter `dtype`, der dieselbe Funktion wie schon bei `arange` etc. hat.

Beispiel: `zeros`, `ones`, `full`

```
arr1 = np.zeros(5)
arr2 = np.ones((2, 2), dtype=np.int32)
arr3 = np.full((2, 3, 4), 5)

print(arr1.dtype, arr1, sep='\n', end='\n\n')
print(arr2.dtype, arr2, sep='\n', end='\n\n')
print(arr3.dtype, arr3, sep='\n', end='\n\n')
```

Ausgabe: `zeros`, `ones`, `full`

```
float64
[0. 0. 0. 0. 0.]

int32
[[1 1]
 [1 1]]
```

```
int64
[[[5 5 5 5]
  [5 5 5 5]
  [5 5 5 5]]

 [[5 5 5 5]
  [5 5 5 5]
  [5 5 5 5]]]
```

Die sogenannten *Identitäts*-Matrizen sind quadratische Matrizen (Zeilenzahl gleich Spaltenzahl), die an jeder Stelle außer der Hauptdiagonalen den Eintrag Null haben. Auf der Hauptdiagonale (die Elemente, bei denen Zeile gleich Spalte) dagegen stehen Einsen. Sie können mit NumPy durch den Befehl `identity` erzeugt werden. Erwartet wird die Zahl der Zeilen/Spalten. Wie schon zuvor kann auch wieder das optionale Argument `dtype` mit übergeben werden.

Eine Verallgemeinerung davon stellt der Befehl `eye` dar: Hier wird eine $N \times M$-Matrix erzeugt, also eine Matrix mit N Zeilen und M Spalten. Wieder stehen nur auf der Hauptdiagonale Einsen, ansonsten wird die Matrix mit Nullen gefüllt. Die Werte N und M dürfen hier verschieden sein. Der Befehl `eye` erwartet also die beiden Werte N und M; optional kann wieder `dtype` übergeben werden.

Beispiel: `identity`, `eye`

```
1  print(np.identity(1))
2  print()
3  print(np.identity(3,dtype=np.int64))
4  print()
5
6
7  print( np.eye(2, 4) )
```

Ausgabe: `identity`, `eye`

```
[[1.]]

[[1 0 0]
 [0 1 0]
 [0 0 1]]

[[1. 0. 0. 0.]
 [0. 1. 0. 0.]]
```

Schließlich kann `diag` benutzt werden, um aus einem Vektor eine *Diagonalmatrix* zu erstellen, d. h. eine Matrix, die nur aus Nullen besteht, außer auf der Hauptdiagonale. Die Einträge der Hauptdiagonale sind die Einträge des Vektors, der als Argument übergeben wird.

Alternativ kann `diag` aus einer Matrix auch die k-te Diagonale extrahieren:

Beispiel: `diag`

```
1  npList = np.array([1, 2, 3])
2  print(np.diag(npList)  , end='\n\n')
3
4  npTab = np.array(
5      [[ 1,  2,  3],
6       [ 4,  5,  6],
7       [ 7,  8,  9],
8       [10, 11, 12]]
9  )
10 print(np.diag(npTab)   , end='\n\n')
11 print(np.diag(npTab, 1), end='\n\n')
12 print(np.diag(npTab, -1) )
```

Ausgabe: `diag`

```
[[1 0 0]
 [0 2 0]
 [0 0 3]]

[1 5 9]

[2 6]

[ 4  8 12]
```

Übergeben wird also entweder eine *eindimensionales* Array, aus dem eine Diagonalmatrix erstellt werden soll, oder ein *zweidimensionales* Array, aus dem eine Diagonale extrahiert werden soll. Im letzteren

Fall kann ein optionaler `int`-Parameter mit übergeben werden, der angibt, wie viele Schritte von der Hauptdiagonalen entfernt die Extraktion starten soll.

11.3.3. Weitere nützliche Funktionen

Im Kapitel 10 wurde bereits die Funktion `meshgrid` angesprochen. Sie gibt *Koordinaten-Arrays zur vektorisierten Auswertung N-dimensionaler Arrays über einem Gitter* aus. Das bedeutet, dass ein `tuple` von NumPy-Arrays erzeugt wird, das die Koordinaten von Datenpunkten in einer mehrdimensionalen Struktur enthält.

Sehen Sie sich dazu das folgende Beispiel an:

$$\begin{array}{c} \\ 0.2 \\ 0.4 \\ 0.6 \\ 0.8 \\ 1.0 \end{array} \begin{array}{c} \begin{array}{ccccc} 2 & 4 & 6 & 8 & 10 \end{array} \\ \begin{pmatrix} 0 & 1 & 3 & 1 & 0 \\ 1 & 3 & 5 & 3 & 1 \\ 3 & 5 & 9 & 5 & 3 \\ 1 & 3 & 5 & 3 & 1 \\ 0 & 1 & 3 & 1 & 0 \end{pmatrix} \end{array}$$

Die Matrix in Klammern sei eine Menge an „Nutzdaten“, z. B. Datenpunkte für einen Plot. Diese müssen im *eindimensionalen* Arbeitsspeicher abgelegt werden. Für schnelle Algorithmen bietet es sich an, diese Nutzdaten zu *vektorisieren*, also als Liste aufzufassen:

$$0 \quad 1 \quad 3 \quad 1 \quad 0 \quad 1 \quad 3 \quad 5 \quad 3 \quad \ldots$$

Für den Anwendungszweck (z. B. Plotten) sei es aber auch notwendig, zu jedem Datenpunkt seine Koordinaten (die Werte außerhalb der Klammer) ausfindig zu machen. Theoretisch ist dies zwar aus dem Index der vektorisierten Daten alleine schon möglich; dies kostet aber einige Berechnungsschritte und kann bei großen Datenmengen viel Zeit in Anspruch nehmen. Stattdessen erzeugt man zwei zusätzliche Vektoren, die die zugehörigen Koordinaten bereits enthalten:

$$\begin{array}{ccccccccccc} 0 & 1 & 3 & 1 & 0 & 1 & 3 & 5 & 3 & \ldots \\ 2 & 4 & 6 & 8 & 10 & 2 & 4 & 6 & 8 & \ldots \\ 0.2 & 0.2 & 0.2 & 0.2 & 0.2 & 0.4 & 0.4 & 0.4 & 0.4 & \ldots \end{array}$$

Es wird also Speicherbedarf für Rechenzeitbedarf aufgewägt. Genau diese Koordinaten-Listen werden von `meshgrid` erzeugt. Zusätzlich versieht `meshgrid` diese Listen mit einer Struktur, die es für Menschen einfacher macht, die einzelnen Blöcke zu erkennen. Im Speicher aber bleibt die vektorisierte Form erhalten. Dies erlaubt es dem Prozessor, sehr einfach alle nötigen Informationen zu finden, da in jeder Liste mit demselben Index gearbeitet werden kann.

Das Beispiel oben zeigt also 2 Listen von Koordinaten ($0.2, 0.4, 0.6, \ldots$ und $2, 4, 6, \ldots$), die zusammen ein Gitter mit 25 Punkten aufspannen. Aus diesen beiden Listen generiert `meshgrid` zwei Listen der Länge 25, die die Koordinaten den vektorisierten Nutzdaten zuordnen. Diese zwei mal 25 Koordinaten-Werte werden uns Menschen aber weiterhin als Matrix dargestellt:

Beispiel: `meshgrid`

```
X, Y = np.meshgrid( np.linspace(0.2, 1.0, 5), np.linspace(2, 10, 5) )

print(X)
print(Y)
```

Ausgabe: `meshgrid`

```
[[0.2 0.4 0.6 0.8 1. ]
 [0.2 0.4 0.6 0.8 1. ]
 [0.2 0.4 0.6 0.8 1. ]
 [0.2 0.4 0.6 0.8 1. ]
 [0.2 0.4 0.6 0.8 1. ]]
[[ 2.  2.  2.  2.  2.]
 [ 4.  4.  4.  4.  4.]
 [ 6.  6.  6.  6.  6.]
 [ 8.  8.  8.  8.  8.]
 [10. 10. 10. 10. 10.]]
```

Dies Funktioniert für beliebige Anzahlen von Dimensionen, d. h. statt zwei Listen können auch drei, vier, ... Listen von Koordinatenachsen angegeben werden.

Wo nötig kann auch die Reihenfolge der Vektorisierung angepasst werden. Standardmäßig geht NumPy von *Row Major Vectorization* aus (d. h. die Nutzdaten werden *zeilenweise* im Speicher abgelegt). Es ist aber auch möglich, *Column Major Vectorization* zu implementieren, d. h. die Daten *spaltenweise* abzulegen. Um für so organisierte Daten geeignete Koordinatenlisten zu erzeugen kann dem optionale Argument `indexing` der Wert `'ij'` zugewiesen werden.

Siehe auch `https://numpy.org/doc/stable/reference/generated/numpy.meshgrid.html` für weitere Details.

Sind nicht die Koordinaten selbst, sondern ihre *Indices* gewünscht, so kann dies mit dem Befehl `indices` schnell erreicht werden. Übergeben wird ein `tuple` von `int`s, der die Ausmaße in jeder Dimension angibt. Der Rückgabewert ist wieder ein `tuple` von Index-Arrays:

Beispiel: `indices`

```
print( np.indices((5,)) )
print()
print( np.indices((3,2)) )
```

Ausgabe: `indices`

```
[[0 1 2 3 4]]

[[[0 0]
  [1 1]
  [2 2]]

 [[0 1]
  [0 1]
  [0 1]]]
```

Wie in Python üblich sind die Symbole, mit denen wir NumPy-Arrays ansprechen lediglich *Referenzen* auf Speicherstellen. Um Kopieen anzulegen brauchen wir also wieder spezielle Methoden. Das vorgestellte Modul `copy` funktioniert auch mit NumPy-Arrays; jedoch bietet das Modul `numpy` auch eine eigene Copy-Funktion, die die auf Effizienz ausgelegte Struktur von NumPy-Arrays ausnutzt und daher merklich schneller arbeitet.

Beispiel: `copy` – NumPy-Arrays

```
1  npData = np.array([1, 2, 3])
2  ref    = npData
3  cpy    = np.copy(npData)
4
5  ref[0] = -1
6
7  print(npData)
8  print(cpy)
```

Ausgabe: `copy` – NumPy-Arrays

```
[-1  2  3]
[1 2 3]
```

Daneben kann `np.copy` auch dazu eingesetzt werden, um „klassische“ Python-`list`s in das NumPy-Format zu übertragen, und dabei die Reihenfolge der Vektorisierung festzulegen. Dies geschieht über den optionalen Parameter `order`, der in der Praxis entweder `'C'` (für Row-Major; in der Programmiersprache C ist dies die konventionelle Anordnung von mehrdimensionalen Datenblöcken) oder `'F'` (für Column-Major; in Fortran ist dies der Standard) ist. Default-Wert ist `'C'`.

Da die *Funktion* `np.copy` zuerst unterscheiden muss, ob sie mit einer Python-`list` oder einem NumPy-Array arbeitet, gehen bei der gezeigten Methode einige Systemtakte an Zeit verloren. Bevorzugt sollte zum Kopieren von NumPy-Arrays daher die *Methode* `copy` verwendet werden. Im obigen Beispiel können wir also Zeile 3 auch ersetzen durch

```
cpy = npData.copy()
```

Der Effekt ist derselbe wie oben schon beschrieben, jedoch wird der Auftrag marginal schneller durchgeführt. Bei häufigem Kopieren kann dies einen nennenswerten Beitrag zur Effizienz leisten.

11.4. Rechenoperationen auf NumPy-Arrays

NumPy wurde zu dem Zweck entwickelt, (natur-)wissenschaftliche Berechnungen durchzuführen. Dazu wurden Konzepte so nah wie möglich an der gewohnten mathematischen Notation umgesetzt.

11.4.1. Rechenoperatoren

Der größte Unterschied zwischen Python-`list`s und NumPy-Arrays ist die Art, wie Rechenoperationen umgesetzt sind. Nämlich setzt NumPy Rechenbefehle *komponentenweise* um. Das Bedeutet, dass Addition und Subtraktion sich wie in der Mathematik bei Vektoren bzw. Matrizen und Tensoren verhalten. Analog dazu gibt es auch die Komponentenweise Multiplikation und Division. Voraussetzung hierfür ist natürlich, dass die beiden Operanden dieselbe Dimension haben:

Beispiel: Rechenoperationen mit Arrays

```
1  u = np.array([1, 2, 3])
2  v = np.array([3, 2, 1])
3
4  print(u + v)
5  print(u - v)
6  print(u * v)
7  print(u / v)
```

Ausgabe: Rechenoperationen mit Arrays

```
[4 4 4]
[-2  0  2]
[3 4 3]
[0.33333333 1.         3.        ]
```

Operationen mit *Skalaren* („einfachen Werten“) werden für jedes Element eines NumPy-Arrays ausgeführt:

Beispiel: Rechenoperationen mit Arrays und Skalaren

```
8  # ...
9  print(2 * u)
10 print(u + 1)
```

Ausgabe: Rechenoperationen mit Arrays und Skalaren

```
[2 4 6]
[2 3 4]
```

Hinzu kommt außerdem das Matrix-Produkt (bzw. Tensorprodukt), das sowohl zwischen Matrizen (Tensoren) und Vektoren als auch zwischen Matrizen und Matrizen (Tensoren und Tensoren) erklärt ist. Wie üblich wird es über den Operator `@` bewirkt.

Matrixprodukt

Das Produkt zweier Matrizen A und B ist die Matrix, die sich aus dem *Skalarprodukt* aller möglichen Zeilenvektoren von A und aller möglichen Spaltenvektoren von B ergibt. Dies verlangt also schon, dass die Spaltenzahl von A gleich der Zeilenanzahl von B sein muss. In Formeln lässt sich schreiben:

$$(AB)_{ij} = \sum_k A_{ik} B_{kj}$$

In einem Beispiel:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \cdot \begin{pmatrix} 0 & 5 \\ 1 & 0 \\ 2 & 2 \end{pmatrix} = \begin{pmatrix} 0\cdot 1 + 1\cdot 2 + 2\cdot 3 & 5\cdot 1 + 0\cdot 2 + 2\cdot 3 \\ 0\cdot 4 + 1\cdot 5 + 2\cdot 6 & 5\cdot 4 + 0\cdot 5 + 2\cdot 6 \end{pmatrix} = \begin{pmatrix} 8 & 11 \\ 17 & 36 \end{pmatrix}$$

Die Gedanken lassen sich in Analogie auch auf Tensoren verallgemeinern.

Beispiel: Matrixprodukt mit NumPy-Arrays

```
1  vector = np.array([1, 2])
2  matrix = np.array([[1,2], [3,4]])
3  tensor = np.array([[[1,2],[3,4],],  [[5,6],[7,8]]])
4
5  print('vector @ vector\n', vector @ vector)
6  print()
7  print('matrix @ vector\n', matrix @ vector)
8  print()
9  print('matrix @ matrix\n', matrix @ matrix)
10 print()
```

```
11  print('tensor @ vector\n', tensor @ vector)
12  print()
13  print('tensor @ matrix\n', tensor @ matrix)
14  print()
15  print('tensor @ tensor\n', tensor @ tensor)
```

Ausgabe: Matrixprodukt mit NumPy-Arrays

```
vector @ vector
 5

matrix @ vector
 [ 5 11]

matrix @ matrix
 [[ 7 10]
 [15 22]]

tensor @ vector
 [[ 5 11]
 [17 23]]

tensor @ matrix
 [[[ 7 10]
  [15 22]]

 [[23 34]
  [31 46]]]

tensor @ tensor
 [[[  7  10]
  [ 15  22]]

 [[ 67  78]
  [ 91 106]]]
```

Siehe auch `https://numpy.org/doc/stable/user/basics.broadcasting.html` für mehr Details zum Verhalten von Rechenoperationen in NumPy.

11.4.2. Transposition

Die Transpoierte eines NumPy-Arrays kann über das Attribut `T` abgefragt werden[5]. Bei der Transponierten handelt es sich um eine Version der Matrix (des Tensors), die um ihre Hauptdiagonale gespiegelt ist:

[5] Die Transponierte eines Tensors ergibt sich lediglich durch eine andere „Lese-Reihenfolge“, und muss daher nicht explizit berechnet werden, wozu man eine *Methode* verwenden würde. Das Attribut `T` legt also nur ein Interface auf dieselben Daten frei, in dem diese neue Lese-Reihenfolge implementiert ist.

Beispiel: Transponierte

```
16  # ...
17
18  print(vector.T, end="\n\n")
19  print(matrix.T, end="\n\n")
20  print(tensor.T, end="\n\n")
```

Ausgabe: Transponierte

```
[1 2]

[[1 3]
 [2 4]]

[[[1 5]
  [3 7]]

 [[2 6]
  [4 8]]]
```

Man kann also allgemein sagen: Wenn bei einem Tensor A ein Objekt mit den Indices $i_0, i_1, \ldots, i_{n-1}$ angesprochen wird (`A[i_0, i_1, ..., i_nMinus1] == x`), so hat dasselbe Objekt in der Transponierten A^T den Index $i_{n-1}, i_{n-2}, \ldots, i_0$ (`A.T[i_nMinus1, i_nMinus2, ..., i_0] == x`): Beim Transpoieren kehrt sich die Reihenfolge der Indices um.

Eine Verallgemeinerung hiervon bietet die Methode `transpose`: Mit übergeben wird eine Liste von Dimensions-IDs in der Reihenfolge, in der sie im neu generierten Objekt erscheinen sollen. Das heißt es gilt: `A[i_0, i_1, i_2] == A.transpose(2, 0, 1)[i_2, i_0, i_1]`.

11.4.3. Funktionen auf Matrix-Elementen

Weiter bringt NumPy auch seine eigene Form der math-Library mit. Diese Form hat den Vorteil, speziell auf NumPy-Arrays abgestimmt zu sein. Auf diese Art kann „Batch-Weise“ mit einem einzigen Befehl eine Funktion für einen ganzen Vorrat von Werten ausgewertet werden. Die Funktionen haben die Ihnen bereits bekannten Namen, erhalten jedoch den Präfix `np.`. Ausgewertet wird also auch wieder Element-Weise:

Beispiel: NumPy-Arrays und Funktionen

```
1  angles = np.linspace(0, np.pi, 7)
2
3  pointsOnACircle = np.array(
4      [np.cos(angles),
5       np.sin(angles)]
6  ).T
7
8  print(pointsOnACircle)
```

Ausgabe: NumPy-Arrays und Funktionen

```
[[ 1.00000000e+00  0.00000000e+00]
 [ 8.66025404e-01  5.00000000e-01]
 [ 5.00000000e-01  8.66025404e-01]
 [ 6.12323400e-17  1.00000000e+00]
 [-5.00000000e-01  8.66025404e-01]
 [-8.66025404e-01  5.00000000e-01]
 [-1.00000000e+00  1.22464680e-16]]
```

Optimierte Funktionen von NumPy

Machen Sie sich an dieser Stelle mit dem Angebot von NumPy vertraut: geben Sie in die Python-Konsole ein: `dir(np)`, und durchsuchen Sie die angezeigte Liste nach Funktionen, die Sie bereits kenenn. Nutzen Sie auch die Gelegenheit, und machen sich mit neuen Funktionen bereit, indem Sie z. B. `help(np.choose)` eingeben.

11.4.4. Berechnung von Feldern

An dieser Stelle wird Ihnen vielleicht auch klar, weshalb die von `meshgrid` erzeugten Felder eine so aufwändige Struktur haben: Dank dieser kann mit einem ganzen Gitter genauso gerechnet werden wie mit den einzelnen Punkten darin!

Stellen Sie sich vor, sie wollen die Wellen einer Wasseroberfläche beschreiben. Sie haben herausgefunden, dass der Ausdruck

$$0.2 \cdot \sqrt{x^2 + y^2} \cdot \cos(x) \cdot \sin(y)$$

die Höhe der Wasseroberfläche am Punkt (x, y) gut beschreibt. Nun wollen Sie einen Plot ihres Modells der Wasseroberfläche erstellen. Sie müssen also den oben gezeigten Ausdruck *für alle x-Werte* und *für alle y-Werte* auswerten, die in Ihrem Plot vorkommen sollen. Natürlich lässt sich dies in Python mit einigen wenigen Schleifen erledigen. Die Verwendung von NumPy erlaubt dafür aber eine sehr viel kürzere Form:

Sie können in zwei Hilfs-Arrays auflisten, für welche x- und y-Werte Sie den Ausdruck auswerten wollen. Aus diesen beiden Arrays generierten Sie ein `meshgrid`, also zwei Arrays `X`, `Y`, die alle denkbaren Kombinationen aus Ihren Hilfsarrays enthält. Nun können Sie mit den Arrays `X`, `Y` arbeiten, als ob sie nur einen einzelnen Punkt (x, y) beschreiben wollten, und erhalten das Ergebnis für *alle* Punkte!

Beispiel: Wellen auf Wasser

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from mpl_toolkits.mplot3d import Axes3D
4
5  xVals = np.linspace(-2*np.pi, 2*np.pi, 201)      # Hilfsarrays: Werte x und y
6  yVals = np.linspace(-2*np.pi, 2*np.pi, 201)      # zwischen -2pi und +2pi
7  X, Y = np.meshgrid(xVals, yVals)                 # Meshgrid erstellen ...
8
9  Z = 0.2 * np.cos(X) * np.sin(Y) * np.sqrt(X**2 + Y**2)  # ... und damit rechnen!
10
11 fig = plt.figure()
12 drw = fig.add_subplot(projection='3d')
13 drw.plot_surface(X, Y, Z)
14 drw.view_init(60, 45)
15 plt.show()
```

Ausgabe: Wellen auf Wasser

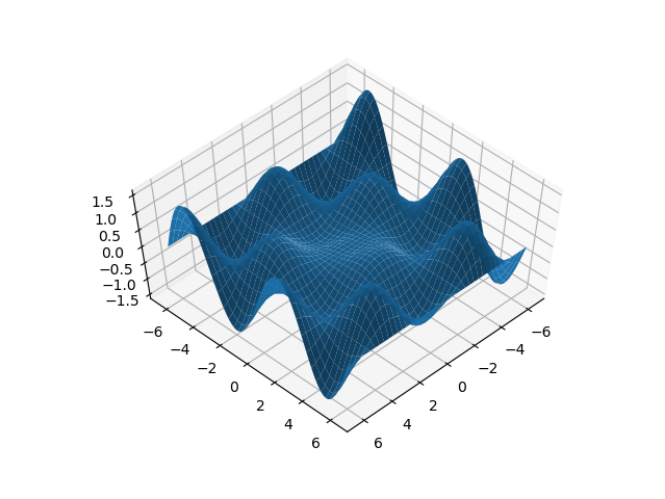


Abbildung 11.1.: Wellen auf Wasser

11.4.5. Vergleiche

Auch Relationen (`==`, `<`, etc.) sind auf NumPy-Arrays definiert; auch hier wird Elementweise gearbeitet. Ergebnis ist also ein Array von `bool`s, das markiert, welche Array-Elemente die angegebene Relation erfüllen. Es sei daran erinnert, dass Arrays von `bool`s wiederum zum „Filtern“ von NumPy-Arrays genutzt werden können[6].

Beispiel: Relationen auf NumPy-Arrays

```
1  import numpy as np
2
3  data = np.random.randint(-5, 5, 20))
4  positive = data > 0
5
6  print("all data         :", data)
7  print("mask             :", positive)
8  print("pos. data points:", data[positive])
```

Ausgabe: Relationen auf NumPy-Arrays

```
all data         : [-2 -2 -2  2  5  1  2  4  2 -1 -5  0  4  2  3  3  3  5  2 -1]
mask             : [False False False  True  True  True  True  True  True False
  False False  True  True  True  True  True  True  True False]
pos. data points: [2 5 1 2 4 2 4 2 3 3 3 5 2]
```

[6] Das Unter-Modul `numpy.random` wird in Abschnitt 11.8 besprochen. Im folgenden Beispiel wird die Funktion `randint(low, high, size)` verwendet, die ein NumPy-Array der Länge `size` mit `int`-Werten zwischen einschließlich `low` und ausschließlich `high` erzeugt. Ich denke, Sie können mit diesem Wissen bereits sehr gut verstehen, wie der Code funktioniert.

Um zu testen, ob zwei NumPy-Arrays als Ganzes gleich sind, kann `array_equal` genutzt werden. Es wird direkt die Gleichheit geprüft, d. h. `np.array_equal(A, B)` gibt genau dann `True` zurück, wenn `A.shape == B.shape` *und* wenn alle Einträge in `A` mit denen in `B` übereinstimmen. Andernfalls ist der Rückgabewert `False`.

Für andere Vergleiche als die Gleichheit können `all` und `any` benutzt werden. Beide erwarten ein NumPy-Array von `bool`s, über die gesammelt das logische `and` bzw. `or` ausgeführt wird. Als Mensch gesprochen: die Funktionen geben `True` zurück, wenn *alle* bzw. *mindestens ein Element* des Arrays schon `True` war. Mit dem optionalen Parameter `axis` können `all` und `any` dazu angewiesen werden, nur entlang einer Achse zu operieren.

Beispiel: `array_equal`, `all` und `any`

```
1  data = np.random.randint(-5, 5, (3, 3))
2  copy = data.copy()
3
4  print(data)
5  print()
6
7  print("Original and copy are identical:", np.array_equal(data, copy))
8  print("Alternative form of same test  :", np.all(data == copy))
9     # hierbei wird shape NICHT verglichen!
10 print()
11
12 for i, truth in enumerate( np.any(data < 0, axis=0) ) :
13     print(f"column {i} has{'' if truth else ' no'} negatie entries.")
14 print()
15
16 for i, truth in enumerate( np.any(data < 0, axis=1) ) :
17     print(f"row {i} has{'' if truth else ' no'} negatie entries.")
```

Ausgabe: `array_equal`, `all` und `any`

```
[[-3 -5 -1]
 [ 3  0  1]
 [ 0 -2  4]]

Original and copy are identical: True
Alternative form of same test  : True

column 0 has negatie entries.
column 1 has negatie entries.
column 2 has negatie entries.

row 0 has negatie entries.
row 1 has no negatie entries.
row 2 has negatie entries.
```

11.5. Reduktion

NumPy bietet viele Lösungen für Operationen auf Arrays. Wir kennen bereits Möglichkeiten, eine Berechnung für jedes Element eines Arrays auszuführen und dabei ein neues Array von Werten zu erhalten. Oft wollen wir aber die Werte nicht isoliert betrachten, sondern aus ihrer Gesamtheit einen einzigen Wert wie z. B.. den Durchschnitt generieren. Auch für solche *Reduktionen* bietet NumPy vorgefertigte Lösungen.

Natürlich kennen wir bereits Möglichkeiten, solche Reduktionen mit Python-Mitteln durchzuführen. Pythons `sum` und `len` geben Summe und Länge eines Arrays aus, und können auch auf NumPy-Arrays angewandt werden. Da diese Funktionen jedoch für sehr viele verschiedenartige Strukturen funktionieren sollen, konnten sie nicht weit optimiert werden. Aus diesem Grund bringt NumPy seine eigenen Funktionen mit, die die spezielle Struktur von NumPy-Arrays ausnutzen, und so eine Aufgabe sehr schnell lösen.

Exkursion: Zeitmessung mit `time`

Das Modul `time` erlaubt – wie es der Name schon suggeriert –, in Python mit Zeiten umzugehen. Neben Funktionen zur Formatierung von Zeitpunkten als Strings und Berechnung von Zeit-Abständen existiert die Funktion `time`, die das aktuelle Datum und Uhrzeit als Rückgabewert liefert. Diese Information wird in einer einzigen Fließkommazahl codiert, welche als Anzahl der Sekunden seit Mitternacht, erster Januar 1970 zu deuten ist. Vorteil dieser Konvention ist, dass solche *Zeitstempel* einfach voneinander subtrahiert werden können, um *Zeitabstände* zu erfahren.

Um den Zeitbedarf eines Algorithmus experimentell zu ermitteln hält man sich häufig an das folgende Schema:

Schema: Zeitbedarf ermitteln

```
import time

R = 1000

tic = time.time()
for run in range(R) :
    # Algorithmus
toc = time.time()

print(f"it took {toc - tic} second to run the algorithm {R} times")
```

Die sehr kleinen Zeiten, die ein Computeralgorithmus zur Durchführung braucht, sind oft starken Schwankungen unterworfen. Um diese Schwankungen auszugleichen, wird meist nicht der Zeitbedarf für *einen* Durchlauf ermittelt, sondern der *durchschnittliche* Zeitbedarf für *sehr viele* Durchläufe des Algorithmus. Daraus erklärt sich die Schleife über `run` im oben gezeigten Schema.

Ein kurzes Script illustriert, wie viel größer der Zeitaufwand ohne NumPy-Funktionen ist:

Beispiel: Zeitbedarf Array-Summation mit NumPy und Python

```
import numpy as np
import time

N = 10000
R =  2000
X = np.linspace(0, 100, N)

print(f"taking the time for summation of {N} values, Python-Style...",
      end="", flush=True)
tic = time.time()
for run in range(R) :
  s = sum(X)
toc = time.time()
print("done")
tPython = toc - tic

print(f"taking the time for summation of {N} values, NumPy-Style...",
      end="", flush=True)
tic = time.time()
for run in range(R) :
  s = np.sum(X)
toc = time.time()
print("done")
tNumpy = toc - tic

print()
print(f"Measured {tPython*1000/R:5.3f} milliseconds using the Python-Method")
print(f"Measured {tNumpy *1000/R:5.3f} milliseconds using the NumPy-Method")
print(f"NumPy is {tPython/tNumpy:5.1f} times faster than native Python")
```

Ausgabe: Zeitbedarf Array-Summation mit NumPy und Python

```
taking the time for summation of 10000 values, Python-Style...done
taking the time for summation of 10000 values, NumPy-Style...done

Measured 1.589 milliseconds using the Python-Method
Measured 0.008 milliseconds using the NumPy-Method
NumPy is 211.5 times faster than native Python
```

Bei mehrdimensionalen Arrays wird standardmäßig über alle Einträge summiert. Optional kann aber auch der `int`-Parameter `axis` mitgegeben werden. Dieser gibt an, über die wievielte Dimension summiert werden soll. Auch der Rückgabetyp des hierbei entstehenden NumPy-Arrays kann über das optionale Argument `dytpe` bestimmt werden.

Beispiel: Summation in mehr Dimensionen

```
import numpy as np

npTab = np.array([[1, 2], [3, 4]])

print( np.sum(npTab) )
print( np.sum(npTab, axis=0) )
print( np.sum(npTab, axis=1) )
```

Ausgabe: Summation in mehr Dimensionen

```
10
[4 6]
[3 7]
```

Das eben zu `np.sum` gesagte gilt so gleichermaßen auch für `np.prod` (Produkt der Werte eines NumPy-Arrays).

Der Mittelwert eines NumPy-Arrays kann über `mean` ermittelt werden. Auch hier können optional `axis` und `dtype` mit angegeben werden. Eine Erweiterung von `mean` stellt `average` dar. Diesem kann ein optionaler Parameter `weights` mitgegeben werden. Wie der Name nahelegt, handelt es sich hierbei um Gewichte für die einzelnen Werte des Daten-Arrays. Der Aufruf `np.average(a, weights=w)` berechnet also

$$\frac{\sum_i a_i w_i}{\sum_i w_i}$$

Der Nenner in diesem Ausdruck – die sogenannte *Zustandssumme* oder *Partition Function* – kann mit demselben Befehl in Erfahrung gebracht werden. Wird das optionale Argument `returned=True` mit übergeben, so ist der Rückgabewert ein `tuple` aus gewichtetem Durchschnitt und Zustandssumme[7]:

Beispiel: Durchschnittgeschwindigkeit in Wasserstoff (Statistische Physik)

```
kB = 1.380649e-23          # Boltzmann constant
T  = 300                   # Temperature in Kelvin
m  = 2 * 1.6735575e-27     # hydrogen molecule mass in kg

velocities = np.linspace(0, 5000, 50000)
energies   = m/2 * velocities**2
w = np.exp(-energies / (kB * T))

vMean, Z = np.average(velocities, weights=w, returned=True)

print("weighted average:", vMean, "m/s")
print("partition function:", Z)
```

Ausgabe: Durchschnittgeschwindigkeit in Wasserstoff (Statistische Physik)

```
weighted average: 887.5169329420108 m/s
partition function: 13942.18222351434
```

Eng mit dem Mittelwert verknüpft sind der *Median* und die *Standardabweichung*.

[7] Ich *hasse* Thermodynamik.

Median und Standardabweichung

Der Median wird oft auch *Mittenwert* (im Gegensatz zum *Mittelwert*) genannt. Wenn man die Werte einer Stichprobe aufsteigend ordnet, so handelt es sich um den Wert „in der Mitte der geordneten Liste“. Bei einer Liste mit gerader Anzahl von Werten ist es das arithmetische Mittel (der Mittelwert) aus den beiden Werten in der Mitte der Liste.

Beispiel 1:
Seien die folgenden Werte gegeben: $1, 5, 8, 5, 2, 1, 4, 3, 1$
Sortiert man diese Liste, so erhält man: $1, 1, 1, 2, 3, 4, 5, 5, 8$
der Median dieser Liste aus 9 Zahlen ist also der fünfte Wert der sortierten Liste, also die 3.

Beispiel 2:
Seien die folgenden Werte gegeben: $1, 5, 8, 5, 2, 4, 3, 1$
Sortiert man diese Liste, so erhält man: $1, 1, 2, 3, 4, 5, 5, 8$
der Median dieser Liste aus 8 Zahlen ist also der Mittelwert aus dem vierten und fünften Wert der sortierten Liste, also $\frac{3+4}{2} = 3.5$.

Die *Standardabweichung* kann als Streuung um den Mittelwert verstanden werden. Es stellt sich heraus, dass die meisten elementaren Prozesse in der echten Welt einer *Normalverteilung* folgen. Gegeben eine Menge von N Werten x_i mit Mittelwert μ, finden wir diese Streuung σ nach der Formel:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \mu)^2}{N}}$$

Die Aussage dieses Wertes σ ist anschaulich, dass die überwiegende Mehrhet der Werte[a] innerhalb von $\mu \pm \sigma$ liegen.

[a] etwa 68%

Median und Standardabweichung können mit den Befehlen `median` und `std` berechnet werden. Wie schon zuvor erlaubt der optionale Parameter `axis` es, beispielsweise Zeilen- oder Spaltenweise zu arbeiten; über `dtype` kann der Datentyp des Ausgabe-Arrays bestimmt werden.

Der Vollständigkeit halber sei auch auf die NumPy-Versionen von `min` und `max` hingewiesen werden, die derselben Logik folgen und Minimun und Maximum eines (Sub-)Arrays ermitteln. Wo nicht das Minimum/Maximum selbst relevant ist, sondern seine Position im Array, können die Befehle `argmin` und `argmax` eingesetzt werden:

Beispiel: Minimum und Maximum

```
1  npTab = np.array([[1, 2], [3, 4]])
2
3  print( "min:", np.min(npTab), "at index", np.argmin(npTab) )
4  print( "max:", np.max(npTab), "at index", np.argmax(npTab) )
5  print( "min (axis=0):", np.min(npTab, axis=0), "at", np.argmin(npTab, axis=0) )
6  print( "max (axis=0):", np.max(npTab, axis=0), "at", np.argmax(npTab, axis=0) )
```

Ausgabe: Minimum und Maximum

```
min: 1 at index 0
max: 4 at index 3
min (axis=0): [1 2] at [0 0]
max (axis=0): [3 4] at [1 1]
```

11.6. Umformungen an NumPy-Arrays

Oft erhalten wir die Daten, mit denen wir arbeiten wollen, nicht im für die weiteren Schritte günstigsten Form. Es kann sein, dass unsere Quelldaten umsortiert, besonders angeordnet, ergänzt oder verschoben werden müssen, bevor wir sinnvoll damit arbeiten können. Die in diesem Abschnitt vorgestellten Befehle helfen uns dabei. Sie stellen eine Unterauswahl der Befehle dar, die auf `https://numpy.org/doc/stable/reference/routines.array-manipulation.html` aufgelisteten sind.

11.6.1. Arrays Verbinden und Trennen

Auf Python-`lists` ist die Addition als Hintereinanderhängen der Listen definiert. Bei NumPy-Arrays dagegen handelt es sich um die Elementweise Addition der Werte. Um die Verkettung von NumPy-Arrays zu erreichen können wir verschiedene Befehle benutzen; das Allzweck-Tool zum Verketten ist der Befehl `concatenate`. Übergeben wird ein `tuple`[8] von NumPy-Arrays, die zu einem einzigen Array zusammengesetzt werden sollen. Optional kann der Parameter `axis` übergeben werden, der bestimmt, „in welcher Richtung“ ein zusätzliches Array angehängt wird. Der Wert von `axis` kann auch `None` sein; in diesem Fall wird aus den Arrays eine eindimensionale Liste erstellt. Alle zu verkettenden Arrays müssen in der gewählten Dimension dieselbe Anzahl von Einträgen haben. Der Default-Wert für `axis` ist `0`.

Beispiel: `concatenate`

```
1  import numpy as np
2
3  a = np.array([[1, 2], [3, 4]])
4  b = np.array([[5, 6]])
5  c = np.array( [7, 8] )
6
7  print( np.concatenate((a, b))             , end="\n\n" )
8  print( np.concatenate((a, b)  , axis=0)   , end="\n\n" )
9  print( np.concatenate((a, b.T), axis=1)   , end="\n\n" )
10 print( np.concatenate((a, b)  , axis=None), end="\n\n" )
```

Ausgabe: `concatenate`

```
[[1 2]
 [3 4]
 [5 6]]

[[1 2]
 [3 4]
 [5 6]]

[[1 2 5]
 [3 4 6]]

[1 2 3 4 5 6]
```

Um beim Verketten eine neue Dimension einzuführen können wir den Befehl `stack` benutzen. Prinzipiell gilt dasselbe, das schon zu `concatenate` gesagt wurde. Während `concatenate` aus zwei N-dimensionalen

[8] beachten Sie daher das zusätzliche Paar von Klammern ()!

Objekten wieder ein N-dimensionales Objekt erzeugt, ist das Erbebnis von `stack` $N+1$-dimensional. Entsprechend darf der Wert von `axis` hier auch zwischen 0 und N liegen.

Beispiel: `stack`

```
10  # ...
11
12  print( np.stack((c, c))        , end="\n\n" )
13  print( np.stack((c, c), axis=0), end="\n\n" )
14  print( np.stack((c, c), axis=1), end="\n\n" )
15  print( np.stack((a, a))        , end="\n\n" )
```

Ausgabe: `stack`

```
[[ 0 -1]
 [ 0 -1]]

[[ 0 -1]
 [ 0 -1]]

[[ 0  0]
 [-1 -1]]

[[[1 2]
  [3 4]]

 [[1 2]
  [3 4]]]
```

Ebenso, wie NumPy-Arrays zusammengefügt werden können, ist es möglich, diese auch zu zerteilen. Hierzu dient der Befehl `split`. Übergeben werden muss zum einen das zu zerteilende Array, sowie eine Information, wie es zerschnitten werden soll. Dies kann entweder in Form eines `int`s oder eines eindimensionalen `int`-Containers (`list`, `tuple`, NumPy-Array, ...) geschehen.

Wird ein `int` übergeben, so legt dieser die Anzahl der entstehenden Blöcke fest. NumPy versucht, das Array in Blöcke gleicher Größe zu zerschneiden. Ist es nicht möglich, eine Array restlos zu zerteilen (z. B.: ein Array mit 5 Elementen soll in 3 Teile zerschnitten werden), wird eine Fehlermeldung ausgelöst.

Bei einer Liste dagegen versteht NumPy die Listen-Einträge als Punkte, an denen geschnitten werden soll. `[2, 4]` teilt ein Array `A` beispielsweise in drei Blöcke: `A[:2]`, `A[2:4]` und `A[4:]`.

Wie üblich kann auch wieder über `axis` festgelegt werden, in welcher Richtung geschnitten werden soll.

Beispiel: `split`

```
1  data = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 0]])
2
3  print( np.split(data, 2) )
4  print( np.split(data, (2, 4), axis=1) )
```

Ausgabe: `split`

```
[array([[1, 2, 3, 4, 5]]), array([[6, 7, 8, 9, 0]])]
[array([[1, 2],
       [6, 7]]), array([[3, 4],
       [8, 9]]), array([[5],
       [0]])]
```

`np.split(data, 2)`

$$\begin{pmatrix}1&2&3&4&5\\6&7&8&9&0\end{pmatrix} \rightarrow \begin{pmatrix}1&2&3&4&5\\6&7&8&9&0\end{pmatrix},$$

`np.split(data, (2, 4), axis=1)`

$$\begin{pmatrix}1&2&3&4&5\\6&7&8&9&0\end{pmatrix} \rightarrow \begin{pmatrix}1&2\\6&7\end{pmatrix}, \begin{pmatrix}3&4\\8&9\end{pmatrix}, \begin{pmatrix}5\\0\end{pmatrix}$$

11.6.2. Elemente Einfügen und Löschen

Mit dem Befehl `insert` können in NumPy-Arrays zusätzliche Spalten eingefügt werden. Es folgt der Syntax

```
np.insert(array, where, what, axis=0)
```

Dabei ist...

- `array` das NumPy-Array, in das eine Spalte eingefügt werden soll
- `where` der Index oder die Indices der Spalten, die eingefügt werden sollen. Es kann sich hierbei also um einen `int` oder einen Container von `int`s handeln.
- `what` das Element oder die Elemente, die eingefügt werden sollen. Es kann sich hierbei um einen einzelnen Wert handeln, der dann in jedes Feld der eingefügten Spalten geschrieben wird, oder ein Container, der alle zu schreibenden Werte enthält.
- `axis` in seiner gewohnten Bedeutung zu verstehen.

Beispiel: `insert`

```
1  data = np.array([0, 1, 2, 3], [4, 5, 6, 7])
2  idx = (1, 3)
3  print( np.insert(data, idx, 999, axis=1) )
4  print()
5
6  data = np.array([[1, 1], [2, 2], [3, 3]])
7  print( np.insert(data, 1, 5) )
8  print()
9
10 print( np.insert(data, 1, 5, axis=1) )
11 print( np.insert(data, [1], [[1],[2],[3]], axis=1) )
```

Ausgabe: `insert`

```
[[  0 999   1   2 999   3]
 [  4 999   5   6 999   7]]

[1 5 1 2 2 3 3]

[[1 5 1]
 [2 5 2]
 [3 5 3]]

[[1 1 1]
 [2 2 2]
 [3 3 3]]
```

Ebenso können Spalten aus NumPy-Arrays mit `delete` gelöscht werden. Analog zu `insert` lautet dabei die Syntax:

```
np.delete(array, where, axis=0)
```

d. h. dieselbe Logik wie für `insert` lässt sich auch auf `delete` anwenden.

11.6.3. Größe und Form Ändern

Die *Methode* `resize` erlaubt es, die Größe eines Arrays nach der Erstellung zu ändern. Übergeben wird ein `tuple`, der die neue Größe des Arrays angibt. Vergrößert sich das Array beim Aufruf von `resize`, so werden Nullen eingefügt; bei Verkleinerungen werden einfach Werte „abgeschnitten“. In beiden Fällen behält NumPy die *vektorisierte Datenmenge* des Arrays bei:

Die Matrix:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

liegt vektorisiert vor als

$$1 \quad 2 \quad 3 \quad 4$$

Nach einem Aufruf von `A.resize((2, 3))`[9] werden zwei Nullen angefügt, um insgesamt 6 Zahlen im Datenpuffer zu erzeugen; wir finden also

$$1 \quad 2 \quad 3 \quad 4 \quad 0 \quad 0$$

die gelesen werden als

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 0 & 0 \end{pmatrix}$$

Ebenso erhalten wir (ausgehen von der 2x2-Matrix) nach Aufruf von `A.reshape((2, 1))` die Matrix:

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

Die Methode `reshape` funktioniert genauso wie `resize`, ändert aber nicht die Gesamt-Datenmenge. Stattdessen wird eine Fehlermeldung erzeugt, wenn die neue Form mehr oder weniger Daten als das

[9] beachten Sie, dass *zwei* Klammern nötig sind, da die `shape` als `tuple` übergeben werden muss!

ursprüngliche Array enthält. Anders als `resize` wird die neue Größe aber nicht direkt auf das Array angewandt, sondern gilt nur für eine *Kopie*. Ist `A` eine 2×3-Matrix, so ist nach dem Aufruf von `A.reshape((3, 2))` immer noch von der Form 2×3. Um die Änderung auf `A` anzuwenden, muss entweder `A = A.reshape((3, 2))` oder `np.reshape(A, (3, 2))` benutzt werden.

Beispiel: `reshape`

```
1  data = np.arange(8).reshape((2,4))
2  print(data)
3
4  data.reshape((4, 2))      # ohne Effekt!
5  print( data )
6
7  data.reshape((3, 3))      # löst Fehlermeldung aus
```

Ausgabe: `reshape`

```
[[0 1 2 3]
 [4 5 6 7]]
[[0 1 2 3]
 [4 5 6 7]]
Traceback (most recent call last):
  File "<stdin>", line 9, in <module>
ValueError: cannot reshape array of size 8 into shape (3,3)
```

Die Methode `flatten` liefert immer ein eindimensionales NumPy-Array, das der vektorisierten Form entspricht, in die Matrix bzw. der Tensor im Speicher vorliegt. Optional kann der Parameter `order` übergeben werden, und so mit `'C'` eine Vektorisierung in Row-Major-Form oder mit `'F'` eine solche in Column-Major-Form erzeugt werden:

Beispiel: `flatten`

```
1  data = np.arange(8).reshape((2,4))
2  data.flatten()  # ohne Effekt
3
4  print(data)
5  print(data.flatten())
6  print(data.flatten('F'))
```

Ausgabe: `flatten`

```
[[0 1 2 3]
 [4 5 6 7]]
[0 1 2 3 4 5 6 7]
[0 4 1 5 2 6 3 7]
```

11.6.4. Pattern aus Arrays

Manchmal finden wir uns in der Situation, dass Teile einer Matrix oder eines Tensors sich wiederholen. In diesem Fall können wir aus den Sub-Pattern den gesamten Tensor mit dem Befehl `repeat` erstellen. Er folgt der Syntax

```
np.repeat(what, repeats, axis=None)
```

wobei `what` das zu wiederholende Subpattern ist. `repeats` kann ein `int` oder ein `tuple` von `int`s, der angibt, wie oft die Elemente von `what` wiederholt werden sollen. `axis` steuert hier wieder, in welcher Richtung die Wiederholungen stattfinden.

Beispiel: `repeat`

```
print(np.repeat(3, 4))

x = np.array([[1,2],[3,4]])

print(np.repeat(x, 2))
print(np.repeat(x, 3, axis=1))
print(np.repeat(x, [1, 2], axis=0))
```

Ausgabe: `repeat`

```
[3 3 3 3]
[1 1 2 2 3 3 4 4]
[[1 1 1 2 2 2]
 [3 3 3 4 4 4]]
[[1 2]
 [3 4]
 [3 4]]
```

Ähnlich wie `repeat` funktioniert auch `tile`: Auch hier wird ein Sub-Pattern wiederholt, kann aber auch in mehrere Richtungen zugleich ausgedehnt werden. Die Syntax lautet:

```
np.tile(what, repeats)
```

Auch hier wieder ist `what` das zu wiederholende Sub-Pattern. `repeats` ist wieder ein `int` oder `tuple`, der jetzt aber die Wiederholungen *je Richtung* angibt:

Beispiel: `tile`

```
# ...

print( np.tile(x, 2) )
print( np.tile(x, (2, 1)) )
```

Ausgabe: `tile`

```
[[1 2 1 2]
 [3 4 3 4]]
[[1 2]
 [3 4]
 [1 2]
 [3 4]]
```

Manche interessante Matrizen bestehen aus Komponenten, die sich durch Verschiebung eines einfacheren Elements ergeben. Betrachten Sie die folgende Matrix[10]:

$$\frac{1}{2}\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \\ 1 & 0 & 0 & 0 & -1 & 0 \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} + \frac{1}{2}\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 \end{pmatrix}$$

Die Zerlegung in eine Summe erinnert an Diagonalmatrizen, weicht von diesen jedoch durch die beiden Einträge in der rechten oberen bzw. linken unteren Ecke von diesen ab. Wir erhalten die Zerlegung, indem wir eine Diagonalmatrix um eine Spalte nach links bzw. rechts verschieben; „überstehende“ Spalten werden dabei am anderen Ende wieder angehängt:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Genau diese Funktion erfüllt der Befehl `roll`. Erwartet wird, um wie viele Schritte die gegebene Matrix verschoben werden soll; optional kann außerdem über den Parameter `axis` angegeben werden, in Richtung

[10] Es handelt sich hier um eine Darstellung der *Ableitung mit zyklischen Randbedingungen* in der *Finite Elemente Methode*.

welcher Achse die Matrix verschoben werden soll. Die obige Matrix lässt sich damit in Python so sehr schnell erstellen:

Beispiel: `roll`

```
import numpy as np

D = 5
derivative = 0.5 * ( np.roll(np.diag(np.ones(D)),  1, axis=1) - \
                     np.roll(np.diag(np.ones(D)), -1, axis=1) )
print(derivative)
```

Ausgabe: `roll`

```
[[ 0.   0.5  0.   0.  -0.5]
 [-0.5  0.   0.5  0.   0. ]
 [ 0.  -0.5  0.   0.5  0. ]
 [ 0.   0.  -0.5  0.   0.5]
 [ 0.5  0.   0.  -0.5  0. ]]
```

11.6.5. Sortieren und Suchen

Wie Sie sich vorstellen können, hat NumPy auch seine eigene, optimierte Sortierfunktion. Diese kann sowohl über den *Befehl* `sort` als auch über die *Methode* `sort` aufgerufen werden. Unterschied der beiden ist, dass der Befehl `sort` eine sortierte Kopie des NumPy-Arrays erstellt, während die Methode *in place* arbeitet, also das Array selbst verändert. Der Befehl arbeitet also wie `sorted`, während die Methode so arbeitet wie `list.sort`.

Für Mehrdimensionale Objekte kann wieder der optionale Parameter `axis` mitgegeben werden, der angibt, ob Zeilen-, Spalten-, Schichten-, ...-weise sortiert wird. Per Default wird entlang der letzten Achse sortiert. Ist `axis=None`, so ordnet NumPy die vektorisierten Daten des Arrays um, ungeachtet der zugrundeliegenden Struktur.

Ein Sortierschlüssel kann in NumPy leider nicht mitgegeben werden[11]. Für die üblichen Anwendungsfälle ist dies aber auch nicht nötig – NumPy hat seine Stärken im Umgang mit Zahlen, die nur selten in einer anderen als der „natürlichen“ Reihenfolge sortiert werden müssen.

[11] Genauer: NumPy erlaubt zwar, Arrays von Listen nach ausgewählten Feldern zu sortieren. Das Anwenden einer Funktion auf die Array-Elemente dagegen ist nicht möglich. Auf diese Möglichkeit soll hier der Übersichtlichkeit halber aber nicht eingegangen werden. Lesen Sie sich hierzu den Eintrag von `help(np.sort)` durch, oder sehen Sie sich die Online-Doku unter `https://numpy.org/doc/stable/reference/generated/numpy.sort.html` an, falls Sie mehr erfahren wollen.

Beispiel: `sort`

```
data = np.array([[1,4], [3,1]])

print( np.sort(data) )   # Sortiere nach letzter Dimension (Spaltenweise)
print(         data  )   # Sortieren hatte keinen Einfluss auf data selbst
print( data.sort()   )   # Sortieren nach letzter Dimension, in place
print(         data  )   # Effekt jetzt sichtbar
print()

print( np.sort(data, axis=None) )    # geplättetes Array sortieren
print( np.sort(data, axis=0)    )    # Zwelenweise sortieren
print(derivative)
```

Ausgabe: `sort`

```
[[1 4]
 [1 3]]
[[1 4]
 [3 1]]
None
[[1 4]
 [1 3]]

[1 1 3 4]
[[1 3]
 [1 4]]
```

Wo es tatsächlich nötig ist, ein NumPy-Array nach einem Kriterium f zu sortieren, wie das mit Pythons `sorted(array, key=f)` möglich wäre, kann auch der Befehl `argsort` verwendet werden. Dieser gibt ein Array von *Indices* aus, das die Information enthält, in welcher Reihenfolge die Elemente angeordnet werden müssten, um ein sortiertes Array zu erhalten. Auch hier kann optional der Parameter `axis` verwendet werden wie oben beschrieben.

Beispiel: `argsort`

```
data = np.array([[1,4], [3,1]])
print( np.argsort(data) )
```

Ausgabe: `argsort`

```
[[0 1]
 [1 0]]
```

Der Befehl `where` kann als NumPy-Variante des Ternären Operators verstanden werden. Abhängig von einem `bool`-Array werden entweder Elemente eines `Yes`-Arrays oder eines `No`-Arrays zurückgegeben:

```
result = np.where(boolArray, yesArray, noArray)
```

Üblicherweise wird das `boolArray` direkt aus einem NumPy-Zahlen-Array über die Vergleichsoperatoren erstellt:

Beispiel: `where`

```
1  import numpy as np
2  data = np.arange(10).reshape(5, 2)
3  print( np.where(data > 5,
4                  data,
5                  data * 10) )
```

Ausgabe: `where`

```
[[ 0 10]
 [20 30]
 [40 50]
 [ 6  7]
 [ 8  9]]
```

11.7. Lineare Algebra

Ein eigenes Teilgebiet der Mathematik mit vielen Anwendungen im „Alltäglichen“ ist die *lineare Algebra*. Sie beschäftigt sich mit Problemen in mehreren Unbekannten, die aber alle nur in *linearer Form* auftauchen (d. h. nicht im Quadrat oder anderen Potenzen, und auch nicht multipliziert miteinander). Das Gleichungssystem:

$$\begin{aligned} x + 2y - 5z &= 7 \\ 8x - 3y + 2z &= 1 \\ 9y - 9z &= -2 \end{aligned}$$

(bzw. der implizite Auftrag: *finde x, y, z so, dass das obige Gleichungssystem nur aus wahren Aussagen besteht*) ist beispielsweise eine Problem aus der Linearen Algebra. Dagegen stellt

$$\begin{aligned} x^2 &= 2 \\ x \cdot y &= 5 \end{aligned}$$

ein *nichtlineares* Gleichungssystem dar, da sowohl x^2 als auch $x \cdot y$ darin vorkommen[12].

NumPy bietet ein eigenes Submodul `np.linalg`, mit dem solche Probleme gelöst werden können. Eine ausführliche Übersicht über Funktionen sowie deren Beschreibungen finden Sie unter `https://numpy.org/doc/stable/reference/routines.linalg.html`.

11.7.1. Lösung von Gleichungssystemen

Vielleicht erinnern Sie sich daran, dass ein lineares Gleichungssystem durch eine Matrixgleichung beschrieben werden kann. Nötig hierzu ist das Aufstellen der *Koeffizientenmatrix* und der *Inhomogenität*.

Koeffizientenmatrix und Inhomogenität

Die *Koeffizientenmatrix* ist eine Matrix, die aus den Vorfaktoren der Variablen des Gleichungssystems besteht. Sie werden dabei so angeordnet, dass die Zeilen der Matrix den Zeilen des Gleichungssystems und die Spalten den Variablen des Systems zugeordnet werden können. Die Koeffizientenmatrix des oben gezeigten Beispiels lautet also

$$A = \begin{pmatrix} 1 & 2 & -5 \\ 8 & -3 & 2 \\ 0 & 9 & -9 \end{pmatrix}$$

[12] Die Lösung $x = \pm\sqrt{2}, y = \pm\frac{5}{\sqrt{2}}$ lässt sich natürlich einfach mit anderen Mitteln der Mathematik finden. Mein innerer Perfektionist konnte dieses Gleichungssystem nicht ungelöst stehen lassen.

Die *Inhomogenität* ist ein Vektor, deren Einträge aus den „rechten Seiten“ des Gleichungssystems besteht. Im Beispiel oben handelt es sich also um den Vektor

$$b = \begin{pmatrix} 7 \\ 1 \\ -2 \end{pmatrix}$$

für eine Lösung

$$v = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

gilt dann die Gleichung

$$Av = b$$

Wenn Sie den Ausdruck Av „ausrechnen“, werden Sie feststellen, dass Sie exakt das Gleichungssystem erhalten, das Sie beschreiben wollten.

Diese Darstellung hat den Vorteil, dass alle Zahlen in der Koeffizientenmatrix A und der Inhomogenität b untergebracht sind, damit also gut für einen Computer dargestellt werden können. Die Variablen sind in v „verstaut“, und zur Beschreibung des eigentlichen Problems nicht nötig. Wenn wir aber einen Vektor v finden, der $Av = b$ löst, haben wir auch die Lösung zu unserem Gleichungssystem.

Der Befehl `linalg.solve` erwartet genau diese Koeffizientenmatrix und Inhomogenität, und gibt die Lösung des Systems[13] in Form des Lösungsvektors v zurück.

Beispiel: `linalg.solve`

```
coeffs = np.array( [[1,  2, -5],
                    [8, -3,  2],
                    [0,  9, -9]] )
inhom = np.array([7, 1, -2])

print( np.linalg.solve(coeffs, inhom) )
```

Ausgabe: `linalg.solve`

```
[-0.28019324 -2.79710145 -2.57487923]
```

[13] sofern eine solche existiert... siehe dazu mehr weiter unten.

Unter- und überbestimmte Gleichungssysteme

Vielleicht können Sie sich denken, dass nicht alle linearen Gleichungssysteme lösbar sind. Eine erste Anforderung an ein Gleichungssystem ist, dass es *ausreichend bestimmt* ist: Ein Gleichungssystem muss mindestens so viele Zeilen haben, wie Variablen darin vorkommen, andernfalls ist es *unterbestimmt* und kann unter keinen Umständen eindeutig gelöst werden. Ein *überbestimmtes System* hat mehr Zeilen als Variablen darin vorkommen. Für ein solches System kann eine Lösung existieren, die Existenz ist aber nicht garantiert.

Ist ein Gleichungssystem weder über- noch unterbestimmt, so ist es im Allgemeinen[a] lösbar. Ein solches Gleichungssystem heißt *wohlbestimmt* und wird durch eine *quadratische Koeffizientenmatrix* beschrieben.

[a] d. h. es gibt noch eine weitere Einschränkung, die weiter unten erklärt wird.

Für `linalg.solve` können nur *wohlbestimmte* Gleichungssysteme angegeben werden. Wird keine quadratische Koeffizientenmatrix angegeben, so löst NumPy die Exception `numpy.linalg.LinAlgError` aus.

Überbestimmte Gleichungssysteme lösen

Ein überbestimmtes Gleichungssystem lässt sich immer in mehrere wohlbestimmte Systeme zerlegen. Beispielsweise kann

$$\begin{aligned} x + 2y &= 1 \\ 3x + 4y &= 0 \\ 5x + 6y &= -1 \end{aligned}$$

als zwei getrennte Probleme aufgefasst werden; das erste Problem ist dabei durch die *ersten* beiden Zeilen gegeben, das zweite durch die *letzten* beiden Zeilen. (Andere Zerlegungen sind ebenso möglich.) Wir finden also zwei Koeffizientenmatrizen und Inhomogenitäten:

$$A^{(1)} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \qquad b^{(1)} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad A^{(2)} = \begin{pmatrix} 3 & 4 \\ 5 & 6 \end{pmatrix} \qquad b^{(2)} = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

Diese beiden Unterprobleme können nun unabhängig voneinander gelöst werden. Stimmen die beiden Teillösungen überein, so lösen sie auch das ganze, überbestimmte System. Ergeben sich dagegen unterschiedliche Lösungen, so ist das überbestimmte System *gar nicht* lösbar. Wichtig für diesen Test ist nur, dass jede Zeile des Gleichungssystems mindestens einmal verwendet wird. Für stark überbestimmte Gleichungssysteme brauchen Sie möglicherweise mehr als zwei Untersysteme. Sprechen Sie mit dem/der MathematikerIn Ihres Vertrauens über weitere Details hierzu.

Im gezeigten Beispiel haben sowohl $A^{(1)}, b^{(1)}$ als auch $A^{(2)}, b^{(2)}$ die Lösung $x = -2, y = 1.5$. Das System ist also lösbar.

Selbst ein wohlbestimmtes Gleichungssystem ist nicht immer lösbar. Bildlich gesprochen gibt es bestimmte Koeffizientenmatrizen, die nicht genug Informationen enthalten, um eine Lösung daraus abzuleiten. Dies ist dann der Fall, wenn die Zeilen der Koeffizientenmatrix voneinander *linear abhängig* sind.

Lineare Abhängigkeit

Betrachten Sie das folgende Gleichungssystem:

$$\begin{aligned} x + 2y &= 2 \\ 2x + 4y &= 4 \end{aligned}$$

wie Sie sehen, ist die zweite Zeile exakt das doppelte der ersten Zeile. Wenn wir die beiden Zeilen jeweils nach y auflösen, finden wir aus beiden Zeilen dieselbe Aussage:

$$y = 1 - \frac{x}{2}$$

Das bedeutet, dass *für jeden Wert* x ein y gefunden werden kann, das das Gleichungssystem löst.

Das Problem liegt jedoch nicht in der Gleichung als Ganzes, sondern nur in der Koeffizientenmatrix. Betrachten Sie nun das folgende System:

$$\begin{aligned} x + 2y &= 2 \\ 2x + 4y &= 5 \end{aligned}$$

Wieder ist die zweite Zeile der Koeffizientenmatrix das Doppelte der ersten Zeile; die Inhomogenität folgt diesem Zusammenhang jedoch nicht. Wenn wir diese beiden Zeilen nach y auflösen, finden wir zwei Gleichungen:

$$\begin{aligned} y &= 1 - \frac{x}{2} \\ y &= \frac{5}{4} - \frac{x}{2} \end{aligned}$$

und daher

$$1 - \frac{x}{2} = \frac{5}{4} - \frac{x}{2} \qquad \Leftrightarrow \qquad 1 = \frac{5}{4}$$

was offensichtlich falsch ist.

Wann immer also zwei oder mehr Zeilen der Koeffizientenmatrix sich nur durch einen gemeinsamen Faktor unterscheiden, existiert also entweder *gar keine* Lösung, oder es sind unendlich viele Lösungen. Die Zeilen der Matrix sind *linear abhängig*.

Ob die Zeilen einer Matrix Linear abhängig sind, lässt sich leicht über die *Determinante* einer Matrix bestimmen. Es handelt sich dabei um eine Zahl, die aufwändig zu berechnen ist und im allgemeinen eine schwer zu verstehende Bedeutung trägt. Ungeachtet der mathematischen Feinheiten reicht es für Sie, zu wissen, dass eine Matrix mit linear abhängigen Zeilen die Determinante 0 hat. Man sagt, die Matrix ist *singulär*.

Mit NumPy können Sie die Determinante einer quadratischen Matrix einfach berechnen mit dem Befehl `linalg.det`.

Beispiel: `linalg.det`

```
coeffs = np.array( [[1, 2],
                    [2, 4]] )

print( np.linalg.det(coeffs) )
```

Ausgabe: `linalg.solve`

```
0
```

Numerische Rundungsfehler

Die Berechnung der Determinante ist, wie angesprochen, ein aufwändiger Prozess. Wenn mit Fließkommazahlen gearbeitet wird, entstehen hier in den einzelnen Teilschritten zwangsweise Rundungsfehler, die sich ansammeln. Es kann somit vorkommen, dass der vom Computer berechnete Wert der Determinanten von null verschieden ist, auch wenn er eigentlich gleich null sein sollte. In diesem Fall wird der berechnete Wert der Determinanten *fast* gleich null sein, also beispielsweise `1.2342E-15`.

Um diesem Problem zu begegnen, prüfen wir in der Praxis *nie*, ob `np.linalg.det(A) == 0`; stattdessen legen wir einen Grenzwert `epsilon` fest, und prüfen:

```
np.abs(np.linalg.det(A)) < epsilon
```

Üblicherweise hat `epsilon` Werte zwischen `1E-7` und `1E-15`.

Zu einer nicht-singulären quadratischen Matrix existiert auch eine *Inverse*.

Inverse einer Matrix

Für einfache Zahlen a, x, b ließe sich die Gleichung

$$ax = b$$

ganz leicht nach x auflösen:

$$x = \frac{1}{a}b$$

Wie wir gesehen haben, ist dies für Matrix-Gleichungen wie $Av = b$ nicht mehr ganz so leicht; a priori steht nicht einmal fest, ob so ein Objekt nach der Art von $\frac{1}{A}$ überhaupt existiert. Wo es aber möglich ist, so eine Matrix zu finden, bezeichnen wir diese als *Inverse*, und schreiben dafür A^{-1}.

Wenn ein Gleichungssystem beschrieben wird durch

$$Av = b$$

dann erhalten wir die Lösung auch über die Inverse von A:

$$v = A^{-1}b$$

Diese lässt sich mit dem Befehl `linalg.inv` ermitteln, sofern sie existiert:

Beispiel: `linalg.inv`

```
coeffs = np.array( [[1,  2, -5],
                    [8, -3,  2],
                    [0,  9, -9]] )

print( np.linalg.inv(coeffs) )
```

Ausgabe: `linalg.solve`

```
[[-0.04347826  0.13043478  0.0531401 ]
 [-0.34782609  0.04347826  0.20289855]
 [-0.34782609  0.04347826  0.09178744]]
```

Gründe gegen die Berechnung der Inversen

Im Allgemeinen sollte die Inverse *nicht* benutzt werden, um ein lineares Gleichungssystem zu lösen. Der Aufwand, die Inverse zu bestimmen ist ungleich höher und anfälliger für numerische Rundungsfehler.

Lohnend kann die Berechnung der Inversen dann sein, wenn Lösungen für *viele* Inhomogenitäten nötig sind. Die Berechnung eines Matrix-Vektor-Produkts ist weniger aufwändig als die Lösung eines linearen Gleichungssystems[a]. Im Allgemeinen sollten Sie es aber vermeiden, die Inverse einer Matrix zu berechnen oder in Ihren Algorithmen zu verwenden.

[a] Selbst in diesem Fall wird eigentlich davon abgeraten. Die Teilschritte, die zur Lösung eines Gleichungssystems nötig sind, lassen sich „recyclen“, d. h. können unabhängig von b durchgeführt werden. Dies erfordert tiefere Kenntnisse, wie Sie in der Vorlesung Numerik der Fakultät Mathematik gezeigt werden. Eine Auflistung von Gründen gegen die Inverse und Techniken, die hier besser zu verwenden sind, finden Sie unter `https://www.johndcook.com/blog/2010/01/19/dont-invert-that-matrix/`

Die Methode `linalg.solve` kann ebenso auch für Tensorgleichungen genutzt werden. Ergebnis ist dann eine Matrix bzw. ein Tensor $N-1$-ter Stufe, wenn A ein Tensor N-ter Stufe ist.

11.7.2. Eigenwerte und Eigenvektoren

Die Multiplikation einer Matrix und eines Vektors ergibt wieder einen Vektor. Für quadratische Matrizen kann es sich dabei um einen *Eigenvektor* mit einem zugehörigen *Eigenwert* handeln.

Eigenwerte und Eigenvektoren

Im Allgemeinen gilt für eine Matrix A und einen Vektor x:

$$Ax = y$$

Dieses y wird im allgemeinen in eine ganz andere Richtung zeigen als x.

Für besondere x kann es aber sein, dass die Multiplikation Ax den Vektor x nur um einen Faktor λ streckt oder staucht:

$$Ax = \lambda x$$

In diesem Fall nennen wir x einen *Eigenvektor von* A. Der Zugehörige Wert λ ist ein *Eigenwert*.

Sofern ein Eigenvektor x existiert, gibt es unendlich viele Eigenvektoren. Denn: Wenn x die Eigenwertgleichung $Ax = \lambda x$ löst, so kann man auch einen beliebigen Faktor α wählen, und damit ein $x' = \alpha x$ erzeugen, so dass gilt:

$$Ax' = A\alpha x = \alpha Ax = \alpha\lambda x = \lambda\alpha x = \lambda x'$$

Das heißt, nachdem ein Eigenvektor skaliert wurde, ist es immer noch ein Eigenvektor. Üblicherweise nennt man daher nur solche Eigenvektoren, die eine Länge von 1 haben.

Unabhängig von dieser Skalierungsfreihet hat eine Matrix normalerweise trotzdem mehrere *linear unabhängige* Eigenvektoren. Im Allgemeinen hat eine $N \times N$-Matrix auch N unabhängige Eigenvektoren und zugehörige Eigenwerte.

Eigenwerte und Eigenvektoren können mit dem einzigen Befehl `linalg.eig` gefunden werden. Einziger Parameter ist ein NumPy-Array, das die Matrix A enthält[14].

Zurück gegeben wird dann ein `tuple` aus zwei NumPy-Arrays: der erste Eintrag hiervon ist ein NumPy-Array, das die Eigenwerte der Matrix (bzw. des Tensors) enthält; der zweite Eintrag ist eine Matrix (oder ein Tensor) *zeilenweise* bestehend aus allen Eigenvektoren (Eigenmatrizen, Eigentensoren, ...). Die ersten Indices von Eigenwert- und Eigenvektor-Array gehören jeweils zusammen:

Beispiel: `linalg.eig`

```
1  pauliX = np.array([[0,1], [1,0]])
2
3  eigenstuff = np.linalg.eig(pauliX)
4
5  for i, eigVal in enumerate(eigenstuff[0]) :
6      print(f"Eigenvalue {i}: {eigVal}, with eigenvector {eigenstuff[1].T[i]}")
```

Ausgabe: `linalg.eig`

```
Eigenvalue 0: 1.0, with eigenvector [0.70710678 0.70710678]
Eigenvalue 1: -1.0, with eigenvector [-0.70710678  0.70710678]
[0.70710678 0.70710678]
[ 0.70710678 -0.70710678]
```

11.7.3. Weitere Nützliche Befehle

Der Befehl `linalg.matrix_power` kann zum Berechnen der Potenz einer Matrix benutzt werden. Er folgt der simplen Syntax:

```
np.linalg.matrix_power(A, n)
```

und berechnet A^n.

[14] Auch hier sind tensorielle Versionen möglich, d. h. statt den Eigenvektoren zu einer Matrix können die Eigenmatrizen zu einem Tensor oder die Eigentensoren zu einem höheren Eigentensor gefunden werden.

Potenz einer Matrix

Das Ergebnis der Matrixmultiplikation einer $N \times N$-Matrix mit sich selbst ist wieder eine $N \times N$-Matrix. Das bedeutet, dass man eine beliebig lange Kette

$$\underbrace{A \cdot A \cdot \ldots \cdot A}_{n \text{ mal}}$$

schreiben und berechnen kann. In Kurzschreibweise wird dies auch als A^n notiert, und naheliegenderweise *Matrixpotenz* genannt.

In vielen Aussagen, die aus der Linearen Algebra hervor gehen, kommt die sogegannte *Spur* (englisch: *trace*) einer Matrix vor.

Spur einer Matrix

Die Spur einer Matrix ist die Summe über ihre Diagnoalelemente:

$$\operatorname{tr} A = \sum_i A_{ii}$$

Beispielsweise ist

$$\operatorname{tr} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = 1 + 5 + 9 = 15$$

Für Tensoren höherer Ordnung ist die Spur die Summe über die Diagonale einer „Schicht“; für alle Indices außer zwei werden also konkrete Werte angegeben. Die verbleibenden, nicht-festen Indices beschreiben dann also wieder eine Matrix, deren Spur wie oben beschrieben berechnet wird.

Der Befehl `trace` berechnet genau diesen Wert. Optional können die Achsen, über die summiert werden sollen, mit den Parametern `axis1` und `axis2` festgelegt werden. Für Matrizen müssen diese die Werte `0` bzw. `1` haben; bei Tensoren der Ordnung N kann dies ein `int` zwischen `0` und `N-1` sein. Ergebnis ist ein Tensor der Ordnung $N - 2$.

Real Maths Ahead

Der Rest dieses Abschnitts beschäftigt sich mit Themen höherer Mathematik. Wenn Sie die Begriffe *Norm, Kondition und Rang einer Matrix* noch nie gehört haben, können Sie den Abschnitt auch überspringen und bei Abschnitt 11.8 weiter lesen. Außerhalb der Mathemathiknahen Naturwissenschaften kommen Sie vermutlich seltenst in Berührung mit Fragestellungen, die diese Begriffe verlangen. Sprechen Sie den/die MathematikerIn Ihres Vertrauens darauf an, wenn Sie eine Erklärung der Zusammenhänge interessiert. Die Vorlesungen *Lineare Algebra* und *Numerik* behandeln diese Themen ausführlich.

Die Norm ist eine weitere Kenngröße aus der Linearen Algebra. Für Vektoren kann man die Norm als „Länge“ auffassen; für Matrizen und Tensoren ist eine anschauliche Notation der größtmögliche Streckungsfaktor, den eine Multiplikation mit der Matrix (dem Tensor) hervorbringt. Diese Umschreibung ist jedoch nur grob richtig, und hängt genauer von der Wahl der Norm ab.

In NumPy sind die Frobenius-Norm, die p-Norm, die Zeilen- und Spalten-Norm und die *Ky Fan*-Norm[15]

[15] Im Englischen heißt diese *nuclear norm* ... Mathe auf Englisch ist einfach so Heavy Metal!

implementiert. Siehe `https://de.wikipedia.org/wiki/Matrixnorm#Wichtige_Matrixnormen` für Definitionen hierzu. Zur Berechnung dient der Befehl `linalg.norm`, der der Syntax

```
np.linalg.norm(A, ord=None, axis=None)
```

folgt. Dabei ist `A` die Matrix (bzw. der Vektor oder Tensor), deren Norm berechnet werden soll. `ord` beschreibt, welche Art von Norm berechnet werden soll. Erlaubte Werte sind in Tabelle 11.2 aufgeführt. Wie üblich wählt `axis` wieder, bezüglich welcher Richtung die Norm berechnet werden soll. Der Wert `None` behält die ursprüngliche Strutkr von `A` bei.

Parameter `ord` und berechnete Norm

Wert für `ord`	Norm für Matrizen	Norm für Vektoren
`None`	Frobenius-Norm: $\sqrt{\sum_{i,j} A_{i,j}^2}$	2-Norm: $\sqrt{\sum_i A_i^2}$
`'fro'`	Frobenius-Norm	–
`'nuc'`	Ky Fan-Norm: $\mathrm{tr}\left(\sqrt{A^* A}\right)$	–
`np.inf`	Maximum der Zeilen-Summen: $\max_i(\sum_j A_{i,j})$	Maximums-Norm: $\max_i(\lvert A_i\rvert)$
`-np.inf`	Minimum der Zeilen-Summen: $\min_i(\sum_j A_{i,j})$	Minimums-Norm: $\min_i(\lvert A_i\rvert)$
`0`	–	Anzahl der von null verschiedenen Einträge
`1`	Maximum der Spalten-Summen: $\max_j(\sum_i A_{i,j})$	p-Norm: $(\sum_i x_i^p)^{1/p}$ mit `p=ord`
`-1`	Minimum der Spalten-Summen: $\min_j(\sum_i A_{i,j})$	p-Norm
`2`	2-Norm (größter Singulärwert)	p-Norm
`-2`	kleinster Singulärwert	p-Norm
andere	–	p-Norm

Tabelle 11.2.: in NumPy implementierte Matrix-Normen

Eine Maßzahl in der Numerik ist die *Kondition* einer Matrix. Sie gibt die „Störungsanfälligkeit“ einer Matrix an und sollte im Optimalfall klein sein (kleiner als 100). Berechnet wird sie über die Norm der Matrix mit dem Befehl `linalg.cond`. Dieser folgt der Syntax

```
np.linalg.cond(A, ord)
```

wobei `A` die Matrix ist, deren Norm berechnet werden soll, und `ord` die Norm identifiziert, mit der gearbeitet werden soll. Erlaubt sind dieselben Werte wie auch für `linalg.norm`, also die in Tabelle 11.2 aufgelisteten Optionen.

Den Rang einer Matrix können Sie mit dem Befehl `linalg.matrix_rank` ausfindig machen.

11.8. Zufallswerte

Wie auch die Lineare Algebra ist die Stochastik ein eigenständiges Teilgebiet der Mathematik[16]. Es ist daher wenig überraschend, dass NumPy ein eigenes Submodul für den Umgang mit (Pseudo-) Zufallszahlen anbietet.

[16] das niemand mag

Sie finden eine ausführliche Beschreibung des Moduls unter `https://numpy.org/doc/stable/reference/random/index.html`.

Da der Präfix `np.random` unangenehm lang ist, wollen wir für dieses Kapitel davon ausgehen, dass es über die Zeile

Einbindung des Submoduls

```
from numpy import random
```

verfügbar gemacht wurde. Dies kann auch parallel zur schon zu Anfang des Kapitels besprochenen Konvention `import numpy as np` möglich. Die Präfixe `np` und `random` verweisen also jeweils auf Objekte aus dem Modul `numpy`.

11.8.1. Zufallsgeneratoren

Das Python-eigene Modul `random` verwendet als Zufallsgenerator den *Mersenne Twister* (oft auch MT19937 genannt, da seine Periode $2^{19937} - 1 \approx 4.3 \cdot 10^{6001}$ beträgt – sehr viel also). Dieser Algorithmus aus dem Jahr 1997 ist in vielen Bereichen der Informatik der Stand der Technik, weist aber einige für Statistik unerwünschte Eigenschaften auf, und hat zudem einen nennenswerten Zeitbedarf. NumPy erlaubt, neben dem MT19937 auch die Wahl anderer Zufallsgeneratoren.

Zufallsgeneratoren

Computer arbeiten *deterministisch*. Das bedeutet, dass das Ergebnis jedes Algorithmus exakt vorherbestimmbar ist (sofern man die Parameter kennt, mit denen der Algorithmus betrieben wird). Das steht natürlich im Widerspruch zur Anforderung an einen *Zufallsgenerator*, der eben gerade ein *zufälliges* Ergebnis liefern soll.

Ein Ausweg aus diesem Dilemma sind *Pseudo*-Zufallsgeneratoren. Diese erzeugen Zahlenfolgen nach einer mathematischen Vorschrift. Die Folgen in ihrer Gesamtheit sind also ebenso berechenbar wie alles andere, das ein Computer leistet. Jedoch kann die Rechenvorschrift so gewählt werden, dass die einzelnen Zahlen dieser Folgen „wild umherspringen“, d. h. wie tatsächlich zufällige Werte wirken.

Den Generatoren dieser Zahlenreihen ist gemein, dass sie rekursiv definiert sind, d. h. die nächste Zufallszahl wird jeweils aus ihrer Vorgängerin berechnet. Für die Zufallszahlen x_i gilt also:

$$x_{i+1} = f(x_i)$$

wobei f eine mathematische Funktion mit günstigen Eigenschaften ist.

Effektiv bedeutet das, dass die erste Zahl einer Zufallsfolge schon die gesamte Folge bestimmt. Diese erste Zahl, üblicherweise *Seed* genannt, sollte daher bei jeder Programmausführung einen anderen Wert haben. Üblich ist, als Seed eine an die Uhrzeit gekoppelte Zahl zu verwenden (z. B. die Zahl der Sekunden seit Mitternacht).

Charakterisiert werden Zufallsgeneratoren (also die oben angesprochene Funktion f) nach drei Kenngrößen: Periode, Zeitbedarf und Korrelation. Mit Periode ist die Anzahl der Zufallszahlen, nach denen sich die Zufallszahlen wiederholen. Zeitbedarf meint der Rechen- bzw. Zeitaufwand, der nötig ist, um eine neue Zahl der Folge zu finden. Die Korrelation ist ein Maß dafür, „wie zufällig“ eine Zahlenreihe ist (also beispielsweise, wie wahrscheinlich es ist, dass auf eine kleine Zahl eine zweite kleine Zahl folgt).

Der Standard-Generator von NumPy ist der PCG64, der zwar eine bedeutend kürzere Periode hat als der MT19937 ($2^{128} \approx 3.4 \cdot 10^{38}$), dafür aber auch schneller berechnet werden kann und einige statistische Vorteile bietet.

Der Link `https://numpy.org/doc/stable/reference/random/performance.html` listet die von NumPy unterstützten Zufallsgeneratoren sowie deren Zeitbedarf auf.

Abgebildet sind die Zufallsgeneratoren als Klassen, deren Methoden Zufallszahlen nach verschiedenen *Verteilungen* berechnen.

Verteilungsfunktionen

Zufall bedeutet noch nicht *gleiche Chancen für alle Ergebnisse*. So kann ein Zufallsprozess z. B. *normalverteilt*, *gleichverteilt* oder einer beliebigen anderen Verteilung folgen. Eine Verteilungsfunktion (Englisch: *distribution*) ist dabei eine Funktion, die beschreibt, wie wahrscheinlich eine Zahl aus einem bestimmten Intervall „gezogen“ wird[a]. In der bekannten *Normalverteilung* ist es beispielsweise sehr viel wahrscheinlicher, einen Wert nahe der 0 zu erhalten; bei der *Gleichverteilung* dagegen ist tatsächlich jedes Ergebnis gleich wahrscheinlich.

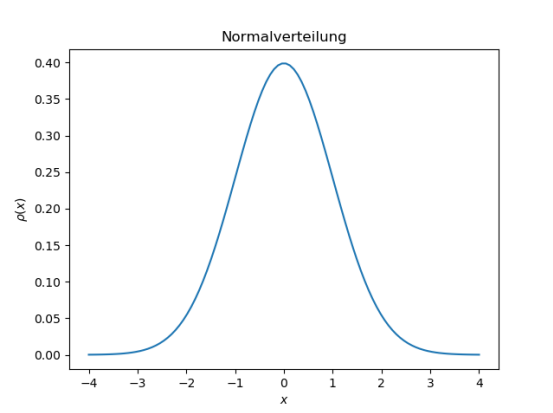


Abbildung 11.2.: Die Normalverteilung

[a] und die außerdem MathematikerInnen und PhysikerInnen viel Anlass zu ~~Streit~~ respektvoller Diskussion über saubere Notation und Verwendung von Fachbegriffen gibt ... Delta-*Funktion* ;)

Wir erhalten einen Zufallsgenerator von NumPy über die Funktion `Generator` aus den bereitgestellten Konstruktoren:

Zufallsgenerator explizit vorbereiten

```
1  from numpy import random
2  import time
3
4  seed = time.time()
5
6  rng = random.Generator(random.MT19937(seed))
7
8  print( rng.uniform() )     # Erzeugt eine zufällige Zahl zwischen 0 und 1
```

Dieses Vorgehen mag Ihnen unnötig aufwändig erscheinen. Für den – von den EntwicklerInnen von NumPy empfohlenen – Generator PCG64 ist das Generator-Objekt schon vorbereitet und kann über *Funktionen* von NumPy direkt angesprochen werden:

Standard-Zufallsgenerator verwenden

```
from numpy import random
print( random.uniform() )     # Erzeugt eine zufällige Zahl zwischen 0 und 1
```

Im Weiteren wird von Funktionen des Moduls `numpy.random` gesprochen. Behalten Sie aber auch im Kopf, dass (wo nichts gegenteiliges gesagt wird) ebenso auch Methoden mit gleichem Namen und Verhalten existieren, die auf solchen explizit gewählten Zufallsgeneratoren aufsetzen.

11.8.2. Verteilungen

Auf die Feinheiten der Stochastik kann hier leider nicht näher eingegangen werden. Stattdessen sollen die wichtigsten Verteilungen mit einigen nützlichen Eigenschaften vorgestellt werden. Eine Übersicht aller von NumPy bereitgestellten Distributionen samt ausführlicher Beschreibung finden Sie unter `https://numpy.org/doc/stable/reference/random/generator.html#distributions`.

Die Funktion `random` erzeugt gleichverteilte Zufallszahlen im Intervall $[0, 1)$, also zwischen 0 (eingeschlossen) und 1 (ausgeschlossen). Ausgegeben wird diese Zahl als `float`. Optional kann auch der Parameter `size` übergeben werden, um ein NumPy-Array zu erzeugen. `size` kann dabei entweder ein `int` sein, und beschreibt dann die Länge eines eindimensionalen NumPy-Arrays, oder ein `tuple`, der die Ausmaße des mehrdimensionalen Arrays in jeder Richtung beschreibt. Jeder Eintrag des Arrays wird dabei neu gezogen:

Beispiel: `random`

```
from numpy import random
print( random.random() )
print( random.random(3) )
print( random.random((3, 3)) )
```

Ausgabe: `random`

```
0.7451593799574697
[0.74561109 0.85833608 0.00708435]
[[0.08326234 0.05691519 0.52119071]
 [0.39721293 0.18348375 0.29456297]
 [0.19200617 0.93908539 0.88486547]]
```

Das Verhalten des optionalen Parameters `size` gilt in dieser Form auch für alle anderen Funktionen, die von `numpy.random` zur Verfügung gestellt werden.

Werden nur Ganzzahlen benötigt, so können diese über die Funktion `randint` bzw. die Methode `integers` erzeugt werden. Erwartet werden die Parameter `low` und `high`; optional kann auch `size` übergeben werden. Erzeugt werden so Ganzzahlen im Intervall [`low`, `high`). Für Fließkommazahlen zwischen vorgegebenen Grenzen kann in gleicher Art `uniform` benutzt werden:

Beispiel: `uniform` und `randint`

```
print(random.uniform(2, 5, size=3))
print(random.randint(2, 5, size=3))
```

Ausgabe: `uniform` und `randint`

```
[3.23220015 4.14602223 3.11928896]
[4 2 4]
```

Die meisten Verteilungen haben zwei Parameter `loc` und `scale`. Dabei beschreibt `loc` den *Lageparameter* (meist der Erwartungswert oder der häufigste Wert der Verteilung; bei symmetrischen Verteilungen ist dies derselbe Wert) und `scale`, wie „eng“ die Verteilung ist, d. h. wie groß die Streuung ist.

In Abb. 11.3 beispielsweise ist die Laplace-Verteilung um den gemeinsamen Erwartungswert 0 aber mit verschiedenen Streuungen σ aufgezeichnet. Siehe auch `https://de.wikipedia.org/wiki/Laplace-Verteilung` für Details zur Laplace-Verteilung.

Die Funktionen `normal` und `laplace` folgen diesem Schema. Sie erzeugen jeweils eine um `loc` zentrierte Normal- bzw. Laplace-Verteilung mit Streuung `scale`.

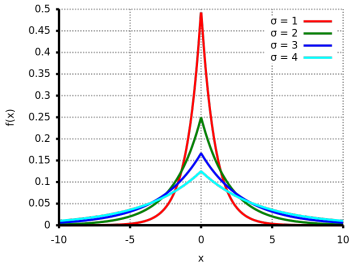


Abbildung 11.3.: Laplace-Verteilung mit verschiedenen Streuungen σ

Beispiel: `normal` und `laplace`

```
1  from numpy import random
2  import matplotlib.pyplot as plt
3
4  dataGauss   = random.normal (0,  1 , size=50000)
5  dataLaplace = random.laplace(0, 0.3, size=50000)
6
7  fig = plt.figure()
8  drw = fig.add_subplot()
9
10 drw.hist(dataGauss  , bins=100, histtype='step', label='normal distribution')
11 drw.hist(dataLaplace, bins=100, histtype='step', label='Laplace distribution')
12 drw.legend()
13
14 plt.show()
```

Ausgabe: `normal` und `laplace`

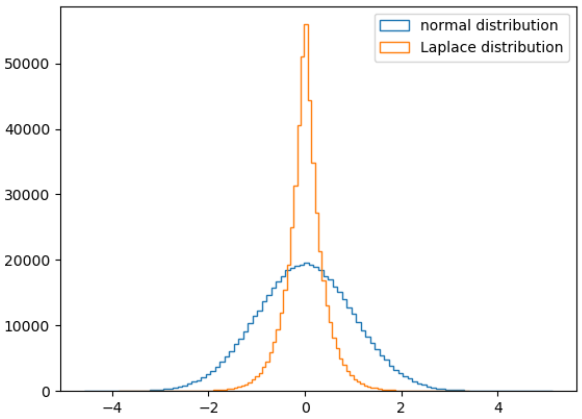


Abbildung 11.4.: Histogramme von Zufallszahlen mit Normal- und Laplace-Verteilung

11.8.3. Weitere Funktionen

Schließlich kann `choice` genutzt werden, um zufällige Elemente aus einem Array auszuwählen. Die Wahrscheinlichkeit für jedes Element ist hier gleich.

`shuffle` kann genutzt werden, um ein bestehendes Array zu durchmischen. Mit `permutation` kann eine durchmischte Kopie eines Arrays erstellt werden, ohne die Originaldaten zu verändern.

Beispiel: `shuffle` und `permutation`

```
1  import numpy as np
2  from numpy import random
3
4  A = np.arange(20).reshape(4, 5)
5  B = random.permutation(A)
6
7  print(A)
8  print(B)
9
10 random.shuffle(A)
11 print(A)
```

Ausgabe: `shuffle` und `permutation`

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
[[10 11 12 13 14]
 [ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [15 16 17 18 19]]
[[ 5  6  7  8  9]
 [15 16 17 18 19]
 [ 0  1  2  3  4]
 [10 11 12 13 14]]
```

12. SciPy

https://docs.scipy.org/doc/scipy/reference/tutorial/ https://xkcd.com/2048/

Appendices

A. Begriffe

B. Tabellen

B.1. Magic Keywords (Dunders)

https://rszalski.github.io/magicmethods/ https://docs.python.org/3/reference/datamodel.html

B.2. Format Strings

https://pyformat.info/

B.3. Escape Sequences

Take from C script

Abbildungsverzeichnis

Tabellenverzeichnis