

---

# **Einführung in das Programmieren für Physiker und Materialwissenschaftler**

*Release 2017beta*

**Gert-Ludwig Ingold**

12.06.2017



<b>1</b>	<b>Einführung</b>	<b>1</b>
1.1	Warum Programmieren lernen? . . . . .	1
1.2	Warum Python? . . . . .	2
1.3	Einige Zutaten . . . . .	3
1.4	Verwendete Symbole . . . . .	5
1.5	Literatur . . . . .	5
1.6	Python-Version . . . . .	6
1.7	Danke an ... . . . .	6
<b>2</b>	<b>Eine Vorschau</b>	<b>7</b>
<b>3</b>	<b>Einfache Datentypen, Variablen und Zuweisungen</b>	<b>11</b>
3.1	Integers . . . . .	11
3.2	Gleitkommazahlen . . . . .	13
3.3	Funktionen für reelle Zahlen . . . . .	15
3.4	Komplexe Zahlen . . . . .	18
3.5	Variablen und Zuweisungen . . . . .	20
3.6	Wahrheitswerte . . . . .	22
3.7	Formatierung von Ausgaben . . . . .	23
<b>4</b>	<b>Kontrollstrukturen</b>	<b>27</b>
4.1	For-Schleife . . . . .	27
4.2	While-Schleife . . . . .	30
4.3	Verzweigungen . . . . .	32
4.4	Abfangen von Ausnahmen . . . . .	34
<b>5</b>	<b>Funktionen</b>	<b>35</b>
5.1	Funktionsdefinitionen . . . . .	35
5.2	Dokumentation von Funktionen . . . . .	37
5.3	Lokale und globale Variable . . . . .	38
5.4	Rekursive Funktionen . . . . .	39
5.5	Funktionen als Argumente von Funktionen . . . . .	39
5.6	Lambda-Funktionen . . . . .	40
5.7	Schlüsselworte und Defaultwerte . . . . .	41
<b>6</b>	<b>Zusammengesetzte Datentypen</b>	<b>43</b>
6.1	Listen . . . . .	43
6.2	Tupel . . . . .	48
6.3	Zeichenketten . . . . .	49
6.4	Dictionaries . . . . .	51
<b>7</b>	<b>Ein- und Ausgabe</b>	<b>57</b>
7.1	Eingabe über die Kommandozeile und die Tastatur . . . . .	57

7.2	Lesen und Schreiben von Dateien . . . . .	58
<b>8</b>	<b>Numerische Programmbibliotheken am Beispiel von NumPy/SciPy</b>	<b>65</b>
8.1	Installation . . . . .	65
8.2	Arrays und Anwendungen . . . . .	65
8.3	Numerische Integration . . . . .	68
8.4	Integration gewöhnlicher Differentialgleichungen . . . . .	68
<b>9</b>	<b>Objektorientiertes Programmieren</b>	<b>73</b>
9.1	Klassen, Attribute und Methoden . . . . .	73
9.2	Vererbung . . . . .	78
<b>10</b>	<b>Erstellung von Grafiken</b>	<b>81</b>
10.1	matplotlib . . . . .	81
10.2	PyX . . . . .	83
<b>A</b>	<b>64-Bit-Gleitkommazahlen nach IEEE-Standard 754</b>	<b>87</b>
<b>B</b>	<b>Unicode</b>	<b>89</b>

---

## Einführung

---

### 1.1 Warum Programmieren lernen?

Computer sind heutzutage allgegenwärtig, sei es im Auto oder in Form eines Smartphones, um nur zwei Beispiele zu nennen. Um Autofahren zu können, muss man dennoch keine Programmiersprache beherrschen. Man muss auch nicht unbedingt Apps für sein Smartphone programmieren, auch wenn dies eine wesentlich produktivere Tätigkeit sein kann als die Benutzung einer App, wie der amerikanische Präsident kürzlich in einer Ansprache betonte:

Don't just buy a new videogame; make one. Don't just download the latest app; help design it. Don't just play on your phone; program it. (B. H. Obama II, 8.12.2013 <sup>1</sup>)

Vielleicht ist es aber auch schon interessant, eine gewisse Vorstellung davon zu bekommen, was in der Steuerelektronik eines Autos oder in einem Smartphone vor sich geht.

Als Studentin oder Student der Naturwissenschaften wird man kaum das Studium absolvieren können, ohne einen Computer zu benutzen. Schließlich muss eine Abschlussarbeit erstellt werden, die vielleicht auch die eine oder andere Abbildung enthalten sollte. Natürlich wird man für diesen Zweck kaum ein eigenes Textverarbeitungsprogramm erstellen, aber bereits bei der Erstellung von Abbildungen nach den eigenen Vorstellungen können Programmierkenntnisse nützlich werden.

Bevor eine Abschlussarbeit geschrieben werden kann, sind Daten zu sammeln und zu analysieren. Dabei muss vielleicht die Kommunikation zwischen Messgerät und Computer programmiert werden. Oder man kann sich bei der Auswertung der Daten mit Hilfe eines selbst geschriebenen Programms viel Zeit und Mühe sparen. Bei einer theoretisch ausgerichteten Arbeit wird man vielleicht feststellen, dass das, was sich mit Papier und Bleistift rechnen lässt, schon lange von anderen gemacht wurde. Viele interessante Probleme erfordern daher den Einsatz des Computers, und dabei wird man ohne Programmierkenntnisse nicht weit kommen. Nicht ohne Grund stellt man heute neben die Experimentalphysik und die Theoretische Physik häufig die Numerische Physik oder *Computational Physics*. Genauso wie Physiker und Materialwissenschaftler mit Papier und Bleistift umgehen können müssen, müssen sie auch in der Lage sein, ein Programm zu schreiben, um die alltäglichen Aufgaben bei der Forschungsarbeit lösen zu können. Ebenso selbstverständlich wird diese Fähigkeit auch für viele Tätigkeiten in der Industrie vorausgesetzt.

Programmieren zu können ist aber nicht nur eine Notwendigkeit für Naturwissenschaftler, Programmieren kann auch Spaß machen. So wie das Knobeln an einer wissenschaftlichen Aufgabe (hoffentlich) Spaß macht, gilt dies auch für die Suche nach einer eleganten und effizienten Lösung eines Problems auf dem Computer <sup>2</sup>. Zugegeben: Es kann durchaus nerven, wenn der Computer ständig Fehler anmahnt und dabei äußerst starrsinnig ist. Allerdings hat er praktisch immer recht. Da hilft es nur, die Fehlermeldungen ernst zu nehmen und ein bisschen Gelassenheit an den Tag zu legen. Dafür beschwert sich der Computer auch nicht, wenn er uns lästige Arbeit abnimmt, indem er zum Beispiel zuverlässig immer wieder die gleichen Programmanweisungen abarbeitet.

---

<sup>1</sup> Anlässlich der Computer Science Education Week 2013 <http://www.youtube.com/watch?v=yE6IfCrqg3s>.

<sup>2</sup> Wer Spaß am Programmieren und am Lösen mathematischer Probleme hat oder einfach die gelernten Programmierfähigkeiten ausprobieren möchte, sollte einen Blick auf [projecteuler.net](http://projecteuler.net) werfen.

## 1.2 Warum Python?

Das Ziel dieser Vorlesung ist es in erster Linie, eine Einführung in das Programmieren für Studierende ohne oder mit nur wenig Programmiererfahrung anzubieten. Aus der Vielzahl von existierenden Programmiersprachen wird hierfür die Programmiersprache *Python* verwendet, wobei allerdings vermieden wird, spezifische Eigenschaften der Programmiersprache zu sehr in den Vordergrund zu rücken. Gründe für die Wahl von Python sind unter anderem<sup>3</sup>:

- Es handelt sich um eine relativ leicht zu erlernende Programmiersprache, die aber dennoch sehr mächtig ist.
- Python ist für die gängigen Betriebssysteme, insbesondere Linux, MacOSX und Windows, frei erhältlich.
- Es unterstützt das Schreiben gut lesbarer Programme.
- Als interpretierte Sprache erlaubt Python das schnelle Testen von Codesegmenten im Interpreter und unterstützt somit das Lernen durch Ausprobieren.
- Python besitzt eine umfangreiche Standardbibliothek (»Python comes with batteries included«), und es existieren umfangreiche freie Programmbibliotheken, u.a. auch für wissenschaftliche Anwendungen wie **NumPy/SciPy**, das wir im Kapitel *Numerische Programmbibliotheken am Beispiel von NumPy/SciPy* besprechen werden.

Python wird gelegentlich vorgeworfen, dass es im Vergleich mit einigen anderen Programmiersprachen relativ langsam ist. Inzwischen gibt es allerdings eine Reihe von effizienten Techniken, um diesem Problem zu begegnen. Im Rahmen dieser Einführung spielen jedoch Geschwindigkeitsaspekte ohnehin kaum eine Rolle.

Die folgenden Zitate zeigen, dass Python nicht nur zur Einführung in das Programmieren taugt, sondern auch in großen Projekten Anwendung findet<sup>4</sup>:

- Tarek Ziadé, Mozilla Service Team  
«The Python programming language supports many programming paradigms and can be put to productive use virtually anywhere. What's more, Python is not restricted to the web. For example, we also use Python for our packaging and build systems.  
  
The Python ecosystem is very rich and well-developed. Our developers can incorporate existing libraries into their projects and only need to develop the new functions that they need.  
  
Python's concise syntax is simple and yet highly productive. This means that new developers can very quickly get involved in our projects, even if they are not yet familiar with Python.»
- Peter Norvig, Google  
«Python has been an important part of Google since the beginning, and remains so as the system grows and evolves. Today dozens of Google engineers use Python, and we're looking for more people with skills in this language.»
- Cuong Do, YouTube.com  
«Python is fast enough for our site and allows us to produce maintainable features in record times, with a minimum of developers.»
- Benedikt Hegner, CERN  
«Most developers in the CMS experiment are physics students looking for new physics in the data. Usually they don't have any formal IT training. Python allows them to be productive from the very start and to dedicate most of their time on the research they want to carry out.»
- Tommy Burnette, Lucasfilm  
«Python plays a key role in our production pipeline. Without it a project the size of The Avengers would have been very difficult to pull off. From crowd rendering to batch processing to compositing, Python binds all things together.»

---

<sup>3</sup> Siehe z.B. auch "Why teach Python?" <https://www.youtube.com/watch?v=X18jt2IVmJY>.

<sup>4</sup> Zitiert nach python, Case Studies & Success Stories, Vol. I (Python Software Foundation) <http://brochure.getpython.info/>

## 1.3 Einige Zutaten

Um auf einem Computer programmieren zu können, ist es im Allgemeinen zunächst erforderlich, die hierfür nötige Software zu installieren, sofern sie nicht ohnehin schon vorhanden ist. Zum einen muss man die Möglichkeit haben, dem Computer die eigenen Absichten mitzuteilen, also ein Programm einzugeben, und zum anderen muss der Computer wissen, wie er die Hardware des Computers dazu bringt, die Vorgaben des Programms umzusetzen.

Beginnen wir mit dem zweiten Punkt und nehmen wir an, dass wir den Programmcode, der in unserem Fall in der Programmiersprache Python geschrieben ist, bereits eingegeben haben. Damit der Computer hiermit etwas anfangen kann, bedarf es bei Python eines sogenannten Interpreters, der den Programmcode interpretiert und auf der Computerhardware ausführen lässt.

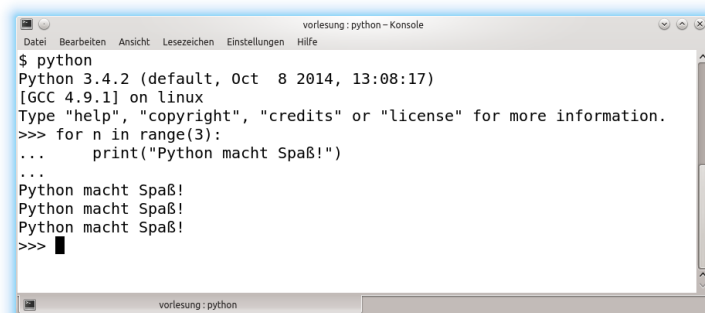
Grundsätzlich kann man die aktuellste Version des Python-Interpreters von der offiziellen Python-Webseite [www.python.org](http://www.python.org) herunterladen. Dabei ist zu beachten, dass gegenwärtig zwei Versionen von Python im Gebrauch sind, die sich in einigen Aspekten voneinander unterscheiden: Python 2 und Python 3. Beim Schreiben dieses Manuskripts waren die Versionen 2.7.11 und 3.5.1 aktuell. Im Rahmen der Vorlesung wird Python 3 besprochen, so dass auf die Installation von Python 2.7 verzichtet werden kann. Es stellt jedoch kein Problem dar, wenn Python 2 ebenfalls installiert ist, wie dies zum Beispiel bei gängigen Linuxsystemen der Fall sein wird. Für den Gebrauch im Rahmen dieser Vorlesung wird eine Python-Version ab 3.3 empfohlen.

Neben dem eigentlichen Python-Interpreter werden wir jedoch noch weitere Software benötigen, wie zum Beispiel die numerischen Programmbibliotheken **NumPy** und **SciPy** <sup>5</sup>, die wir im Kapitel *Numerische Programmbibliotheken am Beispiel von NumPy/SciPy* besprechen werden. Es ist daher am einfachsten, eine Python-Distribution, also eine Art Komplettpaket, zu installieren. Verzichtet man auf die Installation einer geeigneten Distribution, so müssen diese Programmbibliotheken zusätzlich installiert werden.

Es gibt eine Reihe von Python-Distributionen, die auf den Einsatz in der wissenschaftlichen Datenverarbeitung ausgerichtet sind. Hier sind vor allem **Canopy** von Enthought und **Anaconda** von Continuum Analytics zu nennen. Für die Installation von Python 3 ist gegenwärtig die Anaconda-Distribution zu empfehlen, die für die gängigen Betriebssysteme frei verfügbar ist. Unter Linux kann man auf die Installation einer speziellen Python-Distribution häufig verzichten, da sich die benötigte Software sehr einfach über die Paketverwaltung installieren lässt.

Kommen wir nun zum zweiten Punkt: Wie sage ich es meinem Computer? Offenbar muss man das Python-Programm in irgendeiner Form dem Computer mitteilen. Hierfür gibt es eine Reihe von Möglichkeiten, die je nach Anwendungszweck geeignet sind und im Folgenden kurz beschrieben werden sollen.

Manchmal hat man es nur mit kurzen Codestücken zu tun. Dies ist vor allem der Fall, wenn man schnell etwas ausprobieren möchte. Dann eignet sich die Python-Shell, in der man zeilenweise Python-Code eingeben kann, der anschließend ausgeführt wird. Sobald Python installiert ist, ist auch die zugehörige Python-Shell verfügbar. Allerdings ist die Python-Shell nicht sehr komfortabel. Da Tippfehler in den vorigen Zeilen nicht leicht korrigiert werden können, kann das Arbeiten mit der Python-Shell schnell lästig werden. Die folgende Abbildung zeigt die Python-Shell in einem Konsolenfenster unter Linux.



```

$ python
Python 3.4.2 (default, Oct 8 2014, 13:08:17)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> for n in range(3):
...     print("Python macht Spaß!")
...
Python macht Spaß!
Python macht Spaß!
Python macht Spaß!
>>>

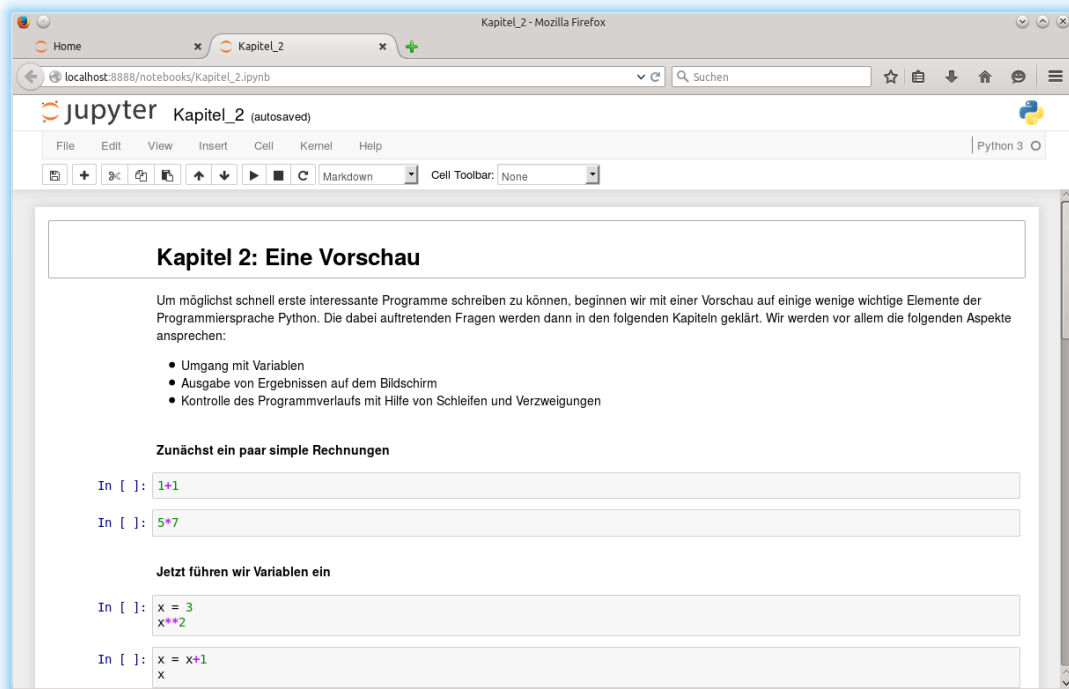
```

Eine erheblich komfortablere Variante der Python-Shell ist die IPython-Shell <sup>6</sup>, die beispielsweise bei der Anaconda-Distribution automatisch installiert wird. Hier kann man alte Eingaben zurückholen, Befehle ergänzen lassen und vieles mehr. Es ist daher sinnvoll, statt der Python-Shell grundsätzlich die IPython-Shell zu verwenden.

<sup>5</sup> [www.scipy.org](http://www.scipy.org)

<sup>6</sup> [ipython.org](http://ipython.org)

Interessant ist die Verwendung von so genannten IPython-Notebooks, die es erlauben, interaktiv mit Python in einem Browser zu arbeiten. Zur Vorlesung werden IPython-Notebooks zur Verfügung gestellt <sup>7</sup>, die den zu besprechenden Programmcode enthalten. Damit kann der Programmcode während der Vorlesung auf dem eigenen Rechner ohne lästiges Abtippen ausprobiert werden. Zudem ist es leicht möglich, den Programmcode zu ändern, um auftretende Fragen zu klären. Schließlich kann man auch eigene Anmerkungen im Notebook eintragen und auf diese Weise eine eigene Vorlesungsmitschrift erstellen. Ein Beispiel für ein Python-Notebook zeigt die folgende Abbildung.



Da Notebookzellen mehrzeiligen Code enthalten können, sind IPython-Notebooks prinzipiell auch dafür geeignet, etwas umfangreicheren Code damit zu entwickeln. Für größere Programmierprojekte bieten sich allerdings bessere Werkzeuge an. Eine Möglichkeit ist die Verwendung von Editoren wie **EMACS** <sup>8</sup> oder **Vim** <sup>9</sup>, die zur Erstellung jeglicher Art von Textdateien und damit auch zum Erstellen von Pythonprogrammen verwendbar sind. Diese Editoren sind sehr mächtig und erfordern daher etwas Einarbeitung, die sich aber auf längere Sicht durchaus lohnen kann. Wer die Mühe der Einarbeitung scheut, kann für die Programmentwicklung auch zu einer der verschiedenen verfügbaren graphischen Entwicklungsumgebungen greifen.

Python stellt eine relativ einfache Entwicklungsumgebung namens IDLE zur Verfügung. Daneben gibt es eine Reihe von freien wie auch von kostenpflichtigen Entwicklungsumgebungen. In der Anaconda-Distribution wird die graphische Entwicklungsumgebung Spyder <sup>10</sup> mitgeliefert, die im nächsten Bild gezeigt ist. Unter Linux kann Spyder auch über die jeweilige Paketverwaltung installiert werden.

Das Spyder-Fenster besteht aus verschiedenen Teilfenstern. Rechts unten erkennen wir ein IPython-Fenster, in dem, wie zu Beginn beschrieben, kürzere Codesegmente getestet werden können. Dies ist besonders hilfreich, wenn man beim Programmieren nicht sicher ist, ob ein bestimmter Programmcode wirklich das Beabsichtigte leistet. Das große Fenster links dient dazu, umfangreichere Programmtexte zu schreiben. Die Ausgabe solcher Programme erfolgt dann wiederum im IPython-Fenster. Ein weiteres Fenster dient zur Anzeige von Dokumentation oder zur Fehlersuche. Je nach Bedarf lässt sich das Spyder-Fenster anpassen, aber zumindest zu Beginn sollte die Standardeinstellung angemessen sein. Wir können an dieser Stelle keine ausführliche Einführung in Spyder geben und verweisen stattdessen auf die zugehörige Dokumentation, die bei <http://pythonhosted.org/spyder/> zu finden ist.

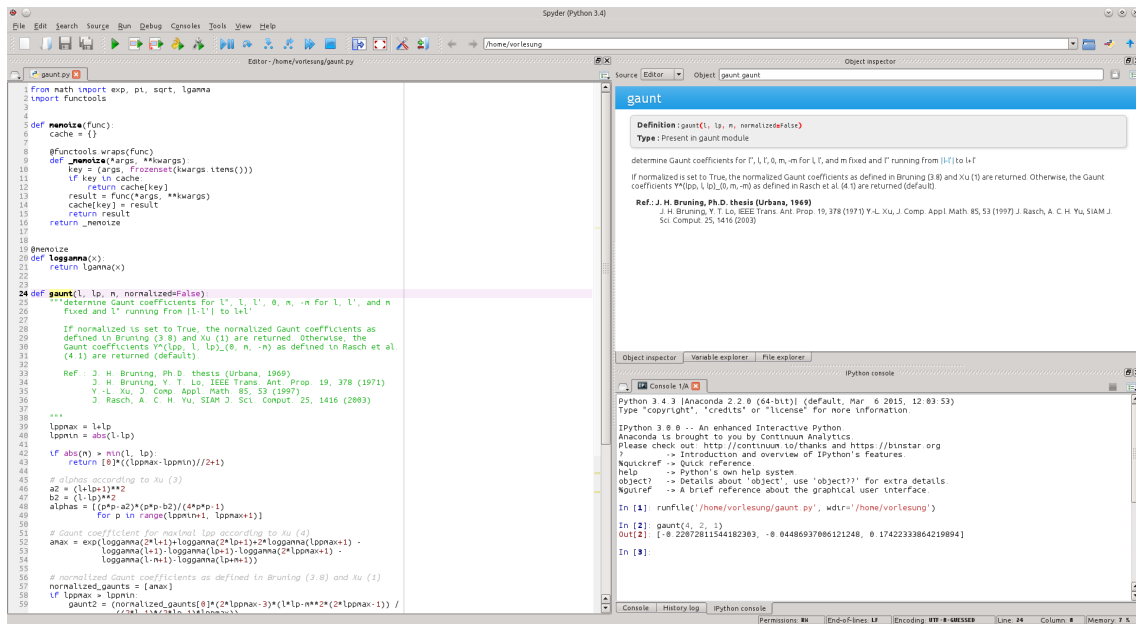
<sup>7</sup> Im Laufe der Entwicklung von IPython wurde auch das Format, in dem Notebookinhalte abgespeichert werden, weiterentwickelt. Zur Zeit ist die Version 4 des Notebookformats aktuell. Falls Sie eine Fehlermeldung bekommen, die darauf hinweist, dass das Notebookformat nicht lesbar ist, liegt dies vermutlich an einer veralteten IPython-Version. Es ist dann sinnvoll, auf eine neuere Version umzusteigen.

<sup>8</sup> [www.gnu.org/software/emacs](http://www.gnu.org/software/emacs)

<sup>9</sup> [www.vim.org](http://www.vim.org)

<sup>10</sup> Spyder steht für «Scientific PYthon Development EnviRonment»





## 1.4 Verwendete Symbole

>>> stellt den Prompt des Python-Interpreters dar. Darauf folgt der einzugebende Text. Die Folgezeilen enthalten gegebenenfalls die Ausgabe des Interpreters.

. . . wird im Python-Interpreter als Prompt verwendet, wenn die Eingabe fortzusetzen ist, zum Beispiel im Rahmen einer Schleife. Diese Art der Eingabe kann sich über mehrere Zeilen hinziehen. Zum Beenden wird die EINGABE-Taste ohne vorhergehende Eingabe von Text verwendet.

\$ steht für den Prompt in der Kommandozeile eines Terminalfensters.

! Dieses Symbol weist auf eine wichtige Anmerkung hin.

? Hier folgt eine Frage, die zum Testen des Verständnisses oder als Vertiefung bearbeitet werden kann.

+ Dieses Symbol kennzeichnet weiterführende Anmerkungen, die sich unter anderem auf speziellere Aspekte der Programmiersprache Python beziehen.

## 1.5 Literatur

- Gert-Ludwig Ingold, Vorlesungsmanuskript »Einführung in das Programmieren für Physiker und Materialwissenschaftler«

Das Vorlesungsmanuskript, das Sie gerade lesen, ist online unter [gertingold.github.io/eidprog](https://gertingold.github.io/eidprog) verfügbar.

- [www.python.org](https://www.python.org)

Dies ist die offizielle Python-Webseite. Dort gibt es z.B. die Software zum Herunterladen, eine umfangreiche Dokumentation der Programmiersprache sowie ihrer Standardbibliothek, Verweise auf einführende Literatur und einiges mehr.

- Hans Petter Langtangen, *A Primer on Scientific Programming with Python* (Springer, 2012)
- Michael Weigend, *Python GE-PACKT* (MITP-Verlag, 2015)

Dieses Buch eignet sich als kompaktes Nachschlagewerk. Die 6. Auflage berücksichtigt die Python-Version 3.4.

## 1.6 Python-Version

Dieses Manuskript bezieht sich auf die Python-Version 3.3 oder eine neuere Version. Ein Großteil des behandelten Stoffes ist auch auf Python 2.7 anwendbar. Wichtige Ausnahmen sind das unterschiedliche Verhalten bei der Integer-Division, die Klammerung bei der `print`-Funktion und die Verwendung von Unicode in Zeichenketten.

## 1.7 Danke an ...

- ... die Hörerinnen und Hörer der Vorlesung „Einführung in das Programmieren für Physiker und Materialwissenschaftler“, deren Fragen bei der Überarbeitung des Manuskripts ihren Niederschlag fanden;
- Michael Hartmann für Kommentare und Verbesserungsvorschläge zu diesem Manuskript;
- Michael Drexel für Hinweise auf Tippfehler.

---

## Eine Vorschau

---

Um möglichst schnell erste interessante Programme schreiben zu können, beginnen wir mit einer Vorschau auf einige wenige wichtige Elemente der Programmiersprache Python. Die dabei auftretenden Fragen werden dann in den folgenden Kapiteln geklärt. Wir werden vor allem die folgenden Aspekte ansprechen:

- Umgang mit Variablen
- Ausgabe von Ergebnissen auf dem Bildschirm
- Kontrolle des Programmverlaufs mit Hilfe von Schleifen und Verzweigungen

Da wir nur kurze Codesegmente ausprobieren werden, benutzen wir die Python- oder IPython-Shell und versuchen zunächst einmal, eine ganz einfache Rechnung auszuführen.

```
>>> 1+1
2
>>> 5*7
35
```

Es wird uns auf Dauer jedoch nicht genügen, Rechnungen mit fest vorgegebenen Zahlen auszuführen, sondern wir werden auch Variablen benötigen.

```
>>> x = 3
>>> x**2
9
>>> x = x+1
>>> x
4
```

Wir können also einer Variable, die hier `x` genannt wurde, eine Zahl zuweisen und anschließend mit dieser Variable Rechnungen ausführen. Beispielsweise können wir die Zahl quadrieren. Allerdings muss man sich davor hüten, das Gleichheitszeichen in seiner üblichen mathematischen Bedeutung zu verstehen, denn dann wäre die Anweisung `x = x+1` in der vierten Zeile ein mathematischer Widerspruch. Python interpretiert das Gleichheitszeichen im Sinne einer Zuweisung, bei der die rechte Seite ausgewertet und der linken Seite zugewiesen wird.

Führen wir eine Division aus, so zeigt sich, dass Python nicht nur ganze Zahlen beherrscht, sondern auch Gleitkommazahlen.

```
>>> 5/2
2.5
```

In der Shell wird das Ergebnis einer Anweisung auf dem Bildschirm ausgegeben, sofern das Ergebnis nicht einer Variable zugewiesen wird. Weiter oben hatten wir schon gesehen, dass der Inhalt einer Variable ausgegeben werden kann, indem man den Namen der Variable in der Shell eingibt. Speichert man ein Skript in einer Datei ab und führt diese aus, so erhält man auf diese Weise jedoch keine Ausgabe. Wir speichern die folgenden Anweisungen in einer Datei namens `f00.py` ab <sup>1</sup>:

---

<sup>1</sup> Der Name `f00` wird häufig als Platzhalter in Beispielprogrammen verwendet. Im Normalfall sollten natürlich aussagekräftigere Bezeichnungen gewählt werden.

```
x = 5*8
x = x+2
x
```

Führen wir diese Datei im Python-Interpreter aus, so erhalten wir keine Ausgabe:

```
$ python foo.py
$
```

In den meisten Fällen möchte man jedoch das Ergebnis eines Programms wissen. Dazu verwendet man die `print`-Anweisung. Im einfachsten Fall kann dies folgendermaßen aussehen:

```
x = 5*8
x = x+2
print(x)
```

Speichern wir diese Anweisungen wiederum in der Datei `foo.py` ab und führen sie im Python-Interpreter aus, so erhalten wir eine Ausgabe:

```
$ python foo.py
42
$
```

Eine etwas informativere Ausgabe kann man folgendermaßen erhalten. Das Programm `foo.py`

```
x = 5*8
x = x+2
print("Das Ergebnis lautet:", x)
```

führt zu folgender Ausgabe:

```
$ python foo.py
Das Ergebnis lautet: 42
$
```

Dabei wurden Anführungszeichen benutzt, um dem Python-Interpreter anzuzeigen, dass sich dazwischen auszugebender Text befindet. Die Darstellung der Ausgabe lässt sich sehr viel genauer spezifizieren. Wie das geht, werden wir im Kapitel [Zeichenketten](#) sehen.

Ein großer Vorteil von Computern liegt unter anderem darin, dass auch vielfache Wiederholungen der gleichen Aufgabe zuverlässig ausgeführt werden. Dazu kann man Schleifen verwenden. Betrachten wir ein einfaches Beispiel, das wir in einer Datei `summe.py` abspeichern:

```
1  summe = 0
2  for n in range(5):
3      print("Schleifendurchgang", n)
4      summe = summe+1
5  print("Summe =", summe)
```

Bevor wir dieses Skript genauer ansehen, wollen wir uns davon überzeugen, dass es vernünftig funktioniert:

```
$ python summe.py
Schleifendurchgang 0
Schleifendurchgang 1
Schleifendurchgang 2
Schleifendurchgang 3
Schleifendurchgang 4
Summe = 5
$
```

Wie entsteht diese Ausgabe? In Zeile 1 belegen wir zunächst die Variable `summe` mit dem Wert 0. Später wird dieser Wert beim Durchlaufen der Schleife jeweils um Eins erhöht. In Zeile 2 beginnt die eigentliche Schleife. Wie der Wert der `range`-Anweisung angibt, soll die Schleife fünfmal durchlaufen werden. Dabei nimmt die Variable `n` nacheinander die Werte 0, 1, 2, 3 und 4 an. Anschließend folgt in den Zeilen 3 und 4 ein eingerückter Bereich. Die Einrückung gibt den Teil des Codes an, der im Rahmen der Schleife wiederholt ausgeführt wird. Dabei muss

immer gleich weit eingerückt werden. Es empfiehlt sich, immer vier Leerzeichen zur Einrückung zu verwenden. Beim Schleifendurchlauf wird nun zunächst angegeben, der wievielte Durchlauf gerade stattfindet. Anschließend wird in Zeile 4 die Variable `summe` inkrementiert. In Zeile 5 wurde die Schleife bereits wieder verlassen und es wird das Ergebnis der fünffachen Inkrementierung ausgegeben.

**?** Rücken Sie im Skript `summe.py` Zeile 5 ebenfalls ein. Überlegen Sie sich zunächst, welche Ausgabe Sie erwarten, und überprüfen Sie Ihre Überlegung, indem Sie das Skript anschließend ausführen.

Sehen wir uns noch an, wie die Eingabe der Schleife in der Python-Shell ablaufen würde:

```
>>> summe = 0
>>> for n in range(5):
...     print("Schleifendurchgang", n)
...     summe = summe+1
...
Schleifendurchgang 0
Schleifendurchgang 1
Schleifendurchgang 2
Schleifendurchgang 3
Schleifendurchgang 4
>>> print("Summe", summe)
Summe 5
```

Nach der Eingabe der zweiten Zeile erkennt der Python-Interpreter, dass gerade eine Schleife begonnen wurde. Dies wird durch Änderung der Eingabeaufforderung von `>>>` nach `...` angedeutet. Will man die Schleife beenden, so verzichtet man auf eine Eingabe und drückt direkt die Eingabetaste.

Bei der Eingabe können verschiedene Dinge schiefgehen. Betrachten wir zwei Beispiele.

```
>>> summe = 0
>>> for n in range(5)
      File "<stdin>", line 1
        for n in range(5)
                        ^
      SyntaxError: invalid syntax
```

Der `SyntaxError` weist darauf hin, dass die Eingabe nicht die Form hat, die der Python-Interpreter erwartet. In diesem Fall fehlt der Doppelpunkt am Ende der Zeile – ein beliebiger Fehler. Kein Problem, man nimmt einfach einen zweiten Anlauf.

```
>>> for n in range(5):
...     print("Schleifendurchgang", n)
      File "<stdin>", line 2
        print("Schleifendurchgang", n)
        ^
      IndentationError: expected an indented block
```

Hier wurde der Doppelpunkt eingegeben, aber in der nächsten Zeile fehlt die Einrückung, worauf die Fehlermeldung `expected an indented block` deutlich hinweist.

Abschließend wollen wir noch Verzweigungen ansprechen, die es erlauben, abhängig davon, ob eine bestimmte Bedingung erfüllt ist, unterschiedlichen Programmcode auszuführen. Betrachten wir wieder ein Beispiel:

```
1 temperatur = 25
2 if temperatur < 28:
3     print("Ich rechne meine Übungsaufgaben.")
4 else:
5     print("Ich gehe lieber in's Freibad.")
6 print("Das war's.")
```

Liest man das Programm und übersetzt die einzelnen Befehle ins Deutsche, so hat man eigentlich schon eine gute Vorstellung davon, was dieses Programm machen wird. Wenn (`>if<`) die Temperatur unter 28 Grad ist, rechne ich meine Übungsaufgaben oder genauer: Das Programm gibt aus, dass ich meine Übungsaufgaben rechnen werde. Andernfalls (`>else<`) ist es zu warm, und ich gehe lieber in's Freibad.

Um zu testen, ob wir den Code richtig interpretiert haben, führen wir das Programm aus und erhalten die folgende Ausgabe:

```
Ich rechne meine Übungsaufgaben.  
Das war's.
```

Setzt man dagegen die Variable `temperatur` auf einen Wert von 28 oder größer, so lautet die Ausgabe:

```
Ich gehe lieber in's Freibad.  
Das war's.
```

Der wichtige Teil dieses Programm befindet sich in den Zeilen 2 bis 5. Ist die Bedingung in der Zeile 2 erfüllt, so wird der danach befindliche, eingerückte Block ausgeführt. Dieser kann durchaus auch mehr als eine Zeile umfassen. Ist die Bedingung in Zeile 2 dagegen nicht erfüllt, so wird der Block nach der `else:-`Zeile ausgeführt. Wichtig sind hier, wie schon bei der Schleife, die Doppelpunkte nach der `if`-Bedingung und nach `else` sowie die Einrückung der zugehörigen Codeblöcke. Die nicht mehr eingerückte Anweisung in Zeile 6 wird unabhängig davon ausgeführt, ob die Bedingung in Zeile 2 erfüllt ist oder nicht.

## Einfache Datentypen, Variablen und Zuweisungen

### 3.1 Integers

Wir hatten bereits im Kapitel *Eine Vorschau* gesehen, dass Python mit ganzen Zahlen umgehen kann und man mit diesen auch Rechenoperationen ausführen kann. Doch wie groß dürfen die ganzen Zahlen werden? Probieren wir es einfach aus, indem wir die 200-ste Potenz von 2 nehmen.

```
>>> 2**200
1606938044258990275541962092341162602522202993782792835301376
```

Im Prinzip kann Python mit beliebig großen Zahlen umgehen. Außerdem können diese natürlich auch ein Vorzeichen besitzen.

```
>>> 5-7
-2
```

Es ist keineswegs selbstverständlich, dass eine Programmiersprache mit beliebig großen ganzen Zahlen umgehen kann. Zudem gibt es Programmiersprachen, die vorzeichenlose ganze Zahlen als Datentyp zur Verfügung stellen. Daher unterscheiden wir im Folgenden zwischen dem mathematischen Begriff »ganze Zahl« und dem Datentyp »Integer«. Dieser Unterschied kommt in Python bei der Addressierung von Listen und Zeichenketten zum Tragen, wo der Index begrenzt ist. Den zulässigen Maximalwert kann man folgendermaßen herausfinden<sup>1</sup>:

```
>>> import sys
>>> sys.maxsize
9223372036854775807
>>> 2**63-1
9223372036854775807
```

Dieses Ergebnis wurde auf einem System mit einem 64-Bit-Prozessor erhalten. Wie lässt sich diese Beschränkung des Wertebereichs von Integers erklären? Im Computer werden sämtliche Daten binär durch Bits dargestellt, wobei jedes Bit den Wert 0 oder 1 annehmen kann. In einem 64-Bit-Prozessor kann ein Integer, der aus maximal 64 Bit besteht, auf einmal verarbeitet werden. Ein Bit wird allerdings für das Vorzeichen benötigt, wie man an den folgenden beiden Beispielen sieht:

6	C	D	8	9	3	2	F
0	1	1	0	1	1	0	0
1	1	0	1	1	0	1	1
1	0	0	0	1	0	0	1
0	0	1	0	0	1	1	0
0	0	1	0	1	0	0	1
1	0	0	1	1	0	0	1
0	0	1	1	0	1	1	1

= 1826132783

9	3	2	7	6	C	D	1
1	0	0	1	0	0	1	1
0	0	1	0	0	1	1	0
0	0	1	0	0	1	1	0
0	1	1	1	0	1	1	0
1	0	1	1	0	1	1	0
1	1	0	1	0	1	1	0
1	0	0	0	1	1	0	1

= -1826132783

Für positive Zahlen ist das erste Bit gleich Null, für negative Zahlen dagegen gleich Eins. Die Viererblöcke stellen jeweils Hexadezimalzahlen dar, wobei nach 0-9 mit A-F oder auch a-f weitergezählt wird, wie die folgende Tabelle zeigt.

<sup>1</sup> Die Bedeutung der `import`-Anweisung werden wir später noch genauer kennenlernen.

dezimal	binär	oktal	hexadezimal
0	0000	00	0
1	0001	01	1
2	0010	02	2
3	0011	03	3
4	0100	04	4
5	0101	05	5
6	0110	06	6
7	0111	07	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Mit den 63 Bit, die für positive Zahlen zur Verfügung stehen, lassen sich die ganzen Zahlen von 0 bis  $2^{63} - 1$  darstellen, womit sich obiges Ergebnis für `sys.maxsize` erklärt.

**?** Wie lässt sich das Zustandekommen des Bitmusters im Falle negativer Zahlen verstehen? Warum ist diese Wahl sinnvoll? (Hinweis: Betrachten Sie die Addition einer positiven und einer negativen Zahl.) Warum ergibt sich ein asymmetrischer Wertebereich für Integers?

In Python kann man auch mit Binär-, Oktal- und Hexadezimalzahlen arbeiten, die durch die Präfixe `0b` oder `0B`, `0o` oder `0O` bzw. `0x` oder `0X` gekennzeichnet werden.

```
>>> 0x6cd8932f
1826132783
>>> 0b11001
25
>>> 0o31
25
```

Die Umwandlung in das Binär-, Oktal- oder Hexadezimalformat erfolgt mit Hilfe der Funktionen `bin`, `oct` bzw. `hex`:

```
>>> bin(25)
'0b11001'
>>> oct(25)
'0o31'
>>> hex(25)
'0x19'
```

Bei der Division von zwei Integers muss man je nach Programmiersprache aufpassen, da die Division möglicherweise einen Rest ergibt. Das Ergebnis könnte daher entweder eine Gleitkommazahl sein oder aber ein Integer, wobei der entstandene Rest ignoriert wird.

In Python 3 ergibt die Division mit einem einfachen Schrägstrich immer eine Gleitkommazahl, selbst wenn bei der Division kein Rest entsteht <sup>2</sup>.

```
>>> 1/2
0.5
>>> 15/3
5.0
```

Dies dürfte in den meisten Fällen das erwünschte Verhalten sein. Es kann aber durchaus sein, dass man tatsächlich eine Integerdivision benötigt. Diese lässt sich mit einem doppelten Schrägstrich erhalten.

<sup>2</sup> In Python 2 bedeutet der einfache Schrägstrich eine echte Integerdivision. Ein Verhalten wie in Python 3 kann ab Python 2.2 mit der Befehlszeile `from __future__ import division` erzwungen werden.



```
>>> 1//2
0
>>> 15//7
2
>>> -15//7
-3
```

? Was macht der `//`-Divisionsoperator tatsächlich, vor allem vor dem Hintergrund des letzten Beispiels? <sup>3</sup>

In anderen Sprachen und auch in Python 2, in denen der einfache Schrägstrich eine Integerdivision bedeutet, kann man eine Gleitkommadivision erzwingen, indem man dafür sorgt, dass das erste Argument nicht ein Integer, sondern eine Gleitkommazahl ist. Wie dies geht, wird im Kapitel *Gleitkommazahlen* erklärt.

Ein wichtiger Punkt, der nicht nur für Integers von Bedeutung ist, ist die Reihenfolge, in der Operationen ausgeführt werden. Dies sei an einem Beispiel illustriert:

```
>>> 2+3*4
14
>>> (2+3)*4
20
```

Die Multiplikation hat also offenbar Vorrang vor der Addition. In der folgenden, für Python gültigen Tabelle haben die höher stehenden Operatoren Vorrang vor den tiefer stehenden <sup>4</sup>:

Operatoren	Beschreibung
<code>**</code>	Exponentiation
<code>+x</code> , <code>-x</code>	Positives und negatives Vorzeichen
<code>*</code> , <code>/</code>	Multiplikation, Division
<code>+</code> , <code>-</code>	Addition, Subtraktion

Wird `**` direkt von einem Plus oder Minus gefolgt, bindet letzteres stärker:

```
>>> 2**-0.5
0.7071067811865476
```

Stehen Operatoren auf der gleichen Stufe, so wird der Ausdruck von links nach rechts ausgewertet. Gegebenenfalls müssen Klammern verwendet werden, um die gewünschte Reihenfolge sicherzustellen. Es spricht auch nichts dagegen, im Zweifelsfall oder zur besseren Lesbarkeit Klammern zu setzen, selbst wenn diese nicht zur korrekten Abarbeitung des Ausdrucks erforderlich sind.

? Was ergibt `-2*4+3**2`? Was ergibt `6**4/2`?

## 3.2 Gleitkommazahlen

Wichtiger als Integers sind in der numerischen Physik die Floats, also Gleitkommazahlen. Man kann sie unter anderem durch Umwandlung mit Hilfe der Funktion `float()` erhalten. <sup>5</sup>

```
>>> type(2)
<class 'int'>
>>> float(2)
2.0
>>> type(float(2))
<class 'float'>
```

Eine Umwandlung von Floats in Integers ist mit der Funktion `int()` möglich, wobei der Nachkommaanteil abgeschnitten wird:

<sup>3</sup> Die Hintergründe kann man in einem [Blog-Artikel](#) von Guido van Rossum nachlesen.

<sup>4</sup> Eine vollständige Liste, die auch noch nicht besprochene Operatoren umfasst, findet man unter Punkt 6.15 *Operator precedence* in der Python-Dokumentation.

<sup>5</sup> In IPython kann die Ausgabe abweichen. Mit `%pprint off` sollte man aber die hier angegebene Ausgabe erhalten.

```
>>> int(2.5)
2
```

Bereits das Anhängen eines Punktes genügt, damit Python die Zahl als Float interpretiert:

```
>>> type(2.)
<class 'float'>
```

Im Gegensatz zu vielen anderen Programmiersprachen ist es nicht notwendig, den Typ explizit festzulegen. Man spricht in diesem Zusammenhang auch von *duck typing*: »If it looks like a duck and quacks like a duck, it must be a duck.«<sup>6</sup>

Für Floats gibt es zwei mögliche Schreibweisen. Dies ist zum einen die Dezimalbruchschreibweise unter Verwendung eines Dezimalpunkts. Stehen vor oder nach dem Dezimalpunkt keine Ziffern, so wird der entsprechende Anteil gleich Null gesetzt.

```
>>> 5.
5.0
>>> 0.25
0.25
>>> .25
0.25
>>> .
File "<stdin>", line 1
      .
      ^
SyntaxError: invalid syntax
```


Wie das letzte Beispiel zeigt, muss aber mindestens vor oder nach dem Dezimalpunkt eine Ziffer stehen. Andernfalls zeigt Python einen Syntaxfehler an.

Für sehr kleine oder sehr große Zahlen ist statt der Dezimalbruchschreibweise die Exponentialschreibweise besser geeignet. Die Zahl wird dabei mit Hilfe einer Mantisse, die nicht zwingend einen Dezimalpunkt enthalten muss, und einem ganzzahligen Exponenten, der ein Vorzeichen enthalten darf, dargestellt. Zwischen Mantisse und Exponenten muss dabei ein *e* oder ein *E* stehen.

```
>>> 1e-2
0.01
>>> 1.53e2
153.0
>>> 1E-5
1e-05
```

Da Dezimalzahlen im Allgemeinen keine endliche Binärdarstellung besitzen, kommt es bei der Umwandlung in die Binärdarstellung zu Rundungsfehlern, die gegebenenfalls bei der Beurteilung der Genauigkeit einer Rechnung zu beachten sind.<sup>7</sup> Das Auftreten von Rundungsfehlern wird an folgendem Beispiel deutlich.

```
>>> 0.1+0.1+0.1-0.3
5.551115123125783e-17
```

 Zeigen Sie, dass die Dezimalzahl 0.1 die Binärdarstellung  $0.\overline{00011}$  besitzt.

Informationen über die Eigenschaften von Floats auf dem verwendeten System kann man folgendermaßen erhalten:

```
>>> import sys
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308,
min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15,
mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

---

<sup>6</sup> [docs.python.org/glossary.html#term-duck-typing](https://docs.python.org/glossary.html#term-duck-typing). Kritiker halten dem entgegen, dass sich auch ein Drache wie eine Ente verhalten kann.

<sup>7</sup> In diesem Zusammenhang kann das Modul `decimal` nützlich sein, das das Rechnen in der Dezimaldarstellung erlaubt. Ferner gibt es das Modul `mpmath`, das eine Rechnung mit einer vorgebbaren, im Prinzip beliebig hohen Genauigkeit ermöglicht.

`sys.float_info.max` ist der maximale Wert, den ein Float darstellen kann, während `sys.float_info.min` die kleinste normalisierte Zahl größer als Null ist, die ein Float darstellen kann. Obwohl man noch kleinere Zahlen darstellen kann, besteht um die Null herum eine Lücke. `sys.float_info.epsilon` ist die Differenz zwischen der kleinsten Zahl größer als Eins, die mit dem gegebenen Float-Typ darstellbar ist, und Eins selbst.

**?** Können Sie die Werte für `max`, `min` und `epsilon` erklären? Hinweis: Es handelt sich hier um ein Double im Sinne des IEEE754-Standards<sup>8</sup> mit einem 11-Bit-Exponenten, einem Vorzeichenbit und einer Mantisse von 52 Bit. Welches ist die kleinste streng positive Zahl, die Sie mit einem Float darstellen können?

Im Gegensatz zu Integers können Gleitkommazahlen also nicht beliebig groß werden, sondern sind auf einen allerdings recht großzügig bemessenen Bereich bis etwas über  $10^{308}$  beschränkt. Werden Gleitkommazahlen zu groß, so erhält man ein vorzeichenbehaftetes Unendlich:

```
>>> 1e400
inf
>>> -1e400
-inf
>>> 1e400 - 1e401
nan
```

Lässt sich mit unendlichen Größen nicht sinnvoll rechnen, wird `nan` ausgegeben, das für »not a number« steht. Eine Division durch Null führt nicht etwa zu `inf`, sondern zu einem Fehler:

```
>>> 1.5/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: float division
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: float division
```

Hierbei wird eine Ausnahme (Exception) ausgelöst, die man geeignet behandeln kann, wie wir im Abschnitt *Abfangen von Ausnahmen* noch sehen werden.

### 3.3 Funktionen für reelle Zahlen

In physikalischen Anwendungen wird man häufig mathematische Funktionen auswerten wollen. Der Versuch, z.B. eine Exponentialfunktion auszuwerten, führt zunächst nicht zum Erfolg:

```
>>> exp(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'exp' is not defined
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'exp' is not defined
```

Es muss vielmehr zunächst das Modul `math` geladen werden:

```
>>> import math
>>> math.exp(2)
7.38905609893065
```

Dieser Schritt ist auch in vielen anderen Sprachen erforderlich. Eine wichtige Ausnahme stellt die Programmiersprache Fortran dar, deren Name ursprünglich als Abkürzung für *Formula Translation* stand und deren Hauptzweck in der Lösung numerischer Probleme besteht. Dort werden mathematische Funktionen als Bestandteile der Sprache direkt zur Verfügung gestellt.

Zum Vergleich mit Python betrachten wir den folgenden Code, der die Verwendung einer mathematischen Funktion in der Programmiersprache C illustriert:

<sup>8</sup> Detailliertere Informationen zu diesem Standard sind im Anhang *64-Bit-Gleitkommazahlen nach IEEE-Standard 754* zu finden.

```
#include <stdio.h>
#include <math.h>

int main(void) {
    double x = 2;
    printf("Die Wurzel von %f ist %f\n", x, sqrt(x));
}
```

Speichert man diesen C-Code in einer Datei und verwendet hierfür beispielsweise den Dateinamen `bsp_math.c`, so lässt sich mit Hilfe des Kommandozeilenbefehls `cc -o bsp_math bsp_math.c -lm` die lauffähige Datei `bsp_math` erzeugen. Hierbei wird das Programm kompiliert und entsprechend der Option `-lm` mit der Mathematikbibliothek gelinkt. Das Resultat ist eine Datei in Maschinencode, die vom Rechner ausgeführt werden kann.

Dieses Codebeispiel zeigt einige Unterschiede zwischen den Programmiersprachen Python und C. Während Python das Programm direkt interpretiert, ist in C ein Kompilationsschritt und das Hinzuladen von Bibliotheken erforderlich. Zum anderen zeigt der C-Code, dass der Datentyp von Variablen deklariert werden muss. In diesem Beispiel wird `x` als doppelt genaue Gleitkommazahl definiert. Der Vorteil besteht darin, dass der resultierende Maschinencode im Allgemeinen deutlich schneller ausgeführt werden kann.

Doch kommen wir zurück zu Python. Nach dem Import des `math`-Moduls kann man Informationen über die zur Verfügung stehenden Funktionen durch Eingabe von `help(math)` im Python-Interpreter erhalten. Von der Ausgabe ist im Folgenden nur ein kleiner Ausschnitt gezeigt:

```
>>> import math
>>> help(math)
```

Help on module math:

NAME  
math

MODULE REFERENCE  
<http://docs.python.org/3.5/library/math>

The following documentation is automatically generated from the Python source files. It may be incomplete, incorrect or include features that are considered implementation detail and may vary between Python implementations. When in doubt, consult the module reference at the location listed above.

#### DESCRIPTION

This module is always available. It provides access to the mathematical functions defined by the C standard.

#### FUNCTIONS

`acos(...)`  
`acos(x)`

Return the arc cosine (measured in radians) of `x`.

`acosh(...)`  
`acosh(x)`

Return the inverse hyperbolic cosine of `x`.

Häufig ist es zu umständlich, den Modulnamen beim Funktionsaufruf immer explizit anzugeben. Stattdessen kann man einzelne Funktionen des Moduls einbinden:

```
>>> from math import sin, cos
>>> sin(0.5)**2+cos(0.5)**2
1.0
```

Alternativ kann man sämtliche Objekte eines Moduls auf einmal einbinden:

```
>>> from math import *
>>> log(10)
2.302585092994046
```

Dieses Vorgehen ist allerdings nicht ganz unproblematisch, da man auf diese Weise einen unter Umständen großen Namensraum einbindet und damit potentiell unabsichtlich Funktionen definiert oder undefiniert wodurch die Funktionsweise des Programms fehlerhaft sein kann.

Die nachfolgende Tabelle gibt die Funktionen des Moduls `math` an.

Name	Funktion
<code>ceil(x)</code>	kleinste ganze Zahl größer oder gleich $x$
<code>copysign(x, y)</code>	ergibt $x$ mit dem Vorzeichen von $y$
<code>fabs(x)</code>	Absolutwert von $x$
<code>factorial(x)</code>	Fakultät, nur für positive ganze Argumente
<code>floor(x)</code>	größte ganze Zahl kleiner oder gleich $x$
<code>fmod(x, y)</code>	Modulofunktion für Gleitkommazahlen
<code>frexp(x)</code>	ergibt Mantisse $m$ und Exponent $e$ für Basis 2
<code>fsum(z)</code>	Summe über $z$ , $z$ ist iterierbarer Datentyp
<code>gcd(a, b)</code>	größter gemeinsamer Teiler der ganzen Zahlen $a$ und $b$ (ab Python 3.5)
<code>isclose(a, b)</code>	überprüft ob $a$ und $b$ nahezu gleich sind (ab Python 3.5) <sup>9</sup>
<code>isfinite(x)</code>	überprüft ob $x$ weder unendlich noch <code>nan</code> (not a number) ist (ab Python 3.2)
<code>isinf(x)</code>	überprüft ob $x$ unendlich ist
<code>isnan(x)</code>	überprüft ob $x$ <code>nan</code> (not a number) ist
<code>ldexp(x, i)</code>	inverse Funktion zu <code>frexp</code> , gibt $x \cdot (2^i)$ zurück
<code>modf(x)</code>	gibt Vor- und Nachkommaanteil als Gleitkommazahl zurück
<code>trunc(x)</code>	schneidet Nachkommaanteil ab
<code>exp(x)</code>	Exponentialfunktion
<code>expm1(x)</code>	Exponentialfunktion minus 1 (ab Python 3.2)
<code>log(x[, base])</code>	Logarithmus, ohne Angabe der Basis: natürlicher Logarithmus
<code>log1p(x)</code>	natürlicher Logarithmus von $x+1$
<code>log2(x)</code>	binärer Logarithmus (ab Python 3.3)
<code>log10(x)</code>	dekadischer Logarithmus
<code>pow(x, y)</code>	$x^y$
<code>sqrt(x)</code>	Quadratwurzel
<code>acos(x)</code>	Arkuskosinus (im Bogenmaß)
<code>asin(x)</code>	Arkussinus (im Bogenmaß)
<code>atan(x)</code>	Arkustangens (im Bogenmaß)
<code>atan2(y, x)</code>	Arkustangens von $y/x$ (im Bogenmaß)
<code>cos(x)</code>	Kosinus ( $x$ im Bogenmaß)
<code>hypot(x, y)</code>	Wurzel aus der Summe der Quadrate von $x$ und $y$
<code>sin(x)</code>	Sinus ( $x$ im Bogenmaß)
<code>tan(x)</code>	Tangens ( $x$ im Bogenmaß)
<code>degrees(x)</code>	Umwandlung von Bogenmaß nach Grad
<code>radians(x)</code>	Umwandlung von Grad nach Bogenmaß
<code>acosh(x)</code>	Areakosinus Hyperbolicus
<code>asinh(x)</code>	Areasinus Hyperbolicus
<code>atanh(x)</code>	Areatangens Hyperbolicus
<code>cosh(x)</code>	Kosinus Hyperbolicus
<code>sinh(x)</code>	Sinus Hyperbolicus
<code>tanh(x)</code>	Tangens Hyperbolicus
<code>erf(x)</code>	Fehlerfunktion <sup>10</sup>

Fortsetzung auf der nächsten Seite

<sup>9</sup> Standardmäßig wird ein relativer Fehler von  $10^{-9}$  zugelassen. Die Dokumentation von `isclose` beschreibt, wie man den relativen und absoluten Fehler selbst setzen kann.

<sup>10</sup> Die Fehlerfunktion ist das normierte Integral über die Gaußfunktion von Null bis zu dem durch das Argument gegebenen Wert. Das

Tabelle 3.1 – Fortsetzung der vorherigen Seite

Name	Funktion
<code>erfc(x)</code>	Komplement der Fehlerfunktion <sup>10</sup>
<code>gamma(x)</code>	Gammafunktion <sup>11</sup>
<code>lgamma(x)</code>	natürlicher Logarithmus des Betrags der Gammafunktion <sup>11</sup>

Außerdem werden die Kreiszahl  $\pi=3.14159\dots$  und die eulersche Zahl  $e=2.71828\dots$  definiert:

```
>>> from math import sin, pi, degrees
>>> sin(0.5*pi)
1.0
>>> degrees(pi)
180.0
>>> from math import log, e
>>> log(e)
1.0
```

Falls `e` bereits als Bezeichner für andere Zwecke benötigt wird, können Sie auch einen anderen Namen vergeben:

```
>>> from math import e as euler_zahl
>>> euler_zahl
2.718281828459045
```

Ab Python 3 sind schließlich noch `math.inf` für positiv Unendlich und `math.nan` für »not a number« definiert.

## 3.4 Komplexe Zahlen

Neben reellen Zahlen benötigt man immer wieder auch komplexe Zahlen. Dabei erzeugt man einen Imaginärteil durch Anhängen des Zeichens `j` oder `J`, das Ingenieure häufig statt des in der Physik üblichen `i` verwenden. Alternativ kann man die Funktion `complex()` verwenden:

```
>>> (1+0.5j)/(1-0.5j)
(0.6+0.8j)
>>> complex(1, 0.5)
(1+0.5j)
```

Möchte man aus den Werten zweier Variablen eine komplexe Zahl konstruieren, geht dies mit der zweiten der gerade genannten Methoden sehr einfach

```
>>> x = 1
>>> y = 2
>>> z = complex(x, y)
>>> z
(1+2j)
```

Falls man die Funktion `complex()` nicht verwenden möchte, muss man beachten, dass die folgenden beiden Wege nicht zum Ziel führen:

```
>>> z = x+yj
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'yj' is not defined
>>> z = x+y*j
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'j' is not defined
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'j' is not defined
```

Komplement der Fehlerfunktion ist die Differenz zwischen 1 und dem Wert der Fehlerfunktion.

<sup>11</sup> Für positive ganze Zahlen entspricht die Gammafunktion der Fakultät des um Eins verminderten Arguments.

In diesem Fall geht Python davon aus, dass es sich bei `yj` bzw. `j` um eine Variable handelt, die jedoch bis jetzt noch nicht definiert wurde. Vielmehr muss die imaginäre Einheit explizit als `1j` geschrieben werden:

```
>>> z = x+y*1j
>>> z
(1+2j)
```

**?** Zeigen Sie, dass das Ergebnis einer Rechnung, die komplexe Zahlen enthält, selbst dann als komplexe Zahl dargestellt wird, wenn das Ergebnis reell ist.

Hat man eine komplexe Zahl einer Variablen zugewiesen (dies wird im Kapitel *Variablen und Zuweisungen* genauer diskutiert), so lassen sich Real- und Imaginärteil wie folgt bestimmen:

```
>>> x = 1+0.5j
>>> x.real
1.0
>>> x.imag
0.5
>>> x.conjugate()
(1-0.5j)
```

Die Unterschiede in den Aufrufen ergeben sich daraus, dass in den ersten beiden Fällen auf Attribute der komplexen Zahl zugegriffen wird, während im letzten Fall eine Methode aufgerufen wird. Diese Zusammenhänge werden im Kapitel *Objektorientiertes Programmieren* klarer werden.

Natürlich wollen wir auch für komplexe Zahlen mathematische Funktionen auswerten. Das Modul `math` hilft hier aber nicht weiter:

```
>>> from math import exp, pi
>>> exp(0.25j*pi)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't convert complex to float; use abs(z)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't convert complex to float; use abs(z)
```

Als Argument wird hier nur eine reelle Zahl akzeptiert. Stattdessen muss man das Modul `cmath` laden:

```
>>> from cmath import exp, pi
>>> exp(0.25j*pi)
(0.7071067811865476+0.7071067811865475j)
```

Dabei ist das Ergebnis immer eine komplexe Zahl. Daher kann es wünschenswert sein, sowohl das Modul `math` als auch das Modul `cmath` zu importieren:

```
>>> import math, cmath
>>> math.exp(2)
7.38905609893065
>>> cmath.exp(0.25j*math.pi)
(0.7071067811865476+0.7071067811865475j)
```

Eine andere Möglichkeit wäre

```
>>> from math import exp, pi
>>> from cmath import exp as cexp
>>> exp(2)
7.38905609893065
>>> cexp(0.25j*pi)
(0.7071067811865476+0.7071067811865475j)
```

**?** Welche Funktion wird verwendet, wenn man nacheinander die Funktion `exp()` aus dem Modul `math` und aus dem Modul `cmath` importiert?

## 3.5 Variablen und Zuweisungen

In einem Beispiel des letzten Abschnitts haben wir bereits eine Zahl einer Variablen zugewiesen. Da dies in einem Programm der Normalfall ist, müssen wir wissen, welche Namen für Variablen zugelassen sind. Ein Variablenname oder allgemein ein Bezeichner besteht aus einer beliebigen Zahl von Zeichen, wobei Buchstaben, der Unterstrich (`_`) und Ziffern zugelassen sind. Das erste Zeichen darf jedoch keine Ziffer sein. Der Unterstrich zu Beginn und am Ende eines Bezeichners impliziert üblicherweise eine spezielle Bedeutung, auf die wir später noch zurückkommen werden. Daher sollte man es sich zur Regel machen, den Unterstrich höchstens innerhalb eines Bezeichners zu verwenden, sofern man nicht bewusst den Unterstrich in anderer Weise einsetzt.

Viel interessanter als Unterstriche sind Buchstaben. Diese umfassen zunächst einmal die Großbuchstaben `A-Z` und Kleinbuchstaben `a-z`. Wie sieht es aber mit Umlauten oder gar mit Buchstaben aus anderen Schriftsystemen, beispielsweise griechischen Buchstaben aus? In diesem Zusammenhang stellt sich die Frage, wie Zeichen im Rechner überhaupt in einer binären Form dargestellt werden. Es gibt hierfür zahlreiche Standards, unter anderem den ASCII-Standard, der noch nicht einmal Umlaute kennt, den ISO-8859-1-Standard, der diesen Mangel behebt, aber dennoch im Umfang sehr beschränkt ist, bis hin zum Unicode-Standard, der mehr als hunderttausend Zeichen umfasst. Für den Unicode-Standard gibt es wiederum verschiedene Codierungen, insbesondere die in der westlichen Welt sinnvolle UTF-8-Kodierung. Etwas mehr Details zu diesem Thema sind im Anhang [Unicode](#) zu finden.

Aus dem vorigen Abschnitt ergibt sich vielleicht der Eindruck, dass die Kodierung von Zeichen ein komplexeres Thema ist, und dieser Eindruck trügt nicht. Die gute Nachricht ist allerdings, dass zum einen immer mehr Computerbetriebssysteme die UTF-8-Kodierung verwenden und für Python-3-Skripte standardmäßig die UTF-8-Kodierung angenommen wird. In Python 3 muss man sich, im Gegensatz zu Python 2, über die Codierung in vielen Fällen keine großen Gedanken mehr machen, sofern man nicht zum Beispiel eine Ausgabe in einer anderen Codierung haben möchte.

Die Verwendung der UTF-8-Kodierung impliziert, dass Buchstaben in Bezeichnern alle Zeichen sein können, die im Unicode-Standard als Buchstaben angesehen werden, also neben Umlauten zum Beispiel auch griechische Zeichen. Ob es wirklich sinnvoll ist, Buchstaben von außerhalb des Bereichs `A-Z` und `a-z` zu verwenden, sollte man sich im Einzelfall gut überlegen. Man muss sich nur vor Augen halten, was es für Folgen hätte, wenn man ein Programm analysieren müsste, das unter Verwendung von chinesischen Schriftzeichen geschrieben wurde. Dennoch ist zum Beispiel der folgende Code für Python 3 kein Problem:

```
>>> from math import pi as π
>>> Radius = 2
>>> Fläche = π*Radius**2
>>> print(Fläche)
12.5663706144
```

Es ist nicht selbstverständlich, dass solche Variablennamen in anderen Programmiersprachen ebenfalls zugelassen sind.

Bei einer Programmiersprache ist immer die Frage zu klären, ob zwischen Groß- und Kleinschreibung unterschieden wird. Python tut dies, so dass `var`, `Var` und `VAR` verschiedene Variablen bezeichnen und für Python nichts miteinander zu tun haben. Auch hier stellt sich im Einzelfall die Frage, ob es sinnvoll ist, in einem Programm Variablennamen gleichzeitig in Groß- und Kleinschreibung zu verwenden. Es ist jedoch wichtig zu wissen, dass eine Fehlfunktion des Programms ihren Ursprung in einem Tippfehler haben kann, bei dem Groß- und Kleinschreibung nicht beachtet wurden.

Es ist für die Verständlichkeit des Programmcodes angebracht, möglichst aussagekräftige Bezeichner zu verwenden, auch wenn diese im Allgemeinen etwas länger ausfallen. Dabei ist es häufig sinnvoll, einen Bezeichner aus mehreren Wörtern zusammenzusetzen. Um die einzelnen Bestandteile erkennen zu können, sind verschiedene Varianten üblich. Man kann zur Trennung einen Unterstrich verwenden, z.B. `sortiere_liste`. Alternativ kann man neue Worte mit einem Großbuchstaben beginnen, wobei der erste Buchstabe des Bezeichners groß oder klein geschrieben werden kann, z.B. `sortiereListe` oder `SortiereListe`. Im ersten Fall spricht man von *mixedCase*, im zweiten Fall von *CamelCase*, da die Großbuchstaben an Höcker eines Kamels erinnern. Details zu den in Python empfohlenen Konventionen für Bezeichner finden Sie im Python Enhancement Proposal [PEP 8](#) mit dem Titel »Style Guide for Python Code« im Abschnitt *Naming Conventions*.

Die folgenden Schlüsselwörter sind in Python als Sprachelemente reserviert und dürfen nicht für Bezeichner



verwendet werden <sup>12</sup>:

False	assert	del	for	in	or	while
None	break	elif	from	is	pass	with
True	class	else	global	lambda	raise	yield
and	continue	except	if	nonlocal	return	
as	def	finally	import	not	try	

**!** Da griechische Buchstaben in der Physik relativ häufig sind, ist insbesondere darauf zu achten, dass `lambda` reserviert ist. Den Grund hierfür werden wir im Kapitel *Lambda-Funktionen* diskutieren.

Variablen kann nun ein Wert zugeordnet werden, wie folgende Beispiele zeigen:

```
>>> x = 1
>>> x = x + 1
>>> print(x)
2
```

Aus der zweiten Zeile wird klar, dass es sich hier trotz des Gleichheitszeichens nicht um eine Gleichung handelt. Vielmehr wird die rechte Seite ausgewertet und der auf der linken Seite stehenden Variablen zugewiesen. Die zweite Zeile müsste also eigentlich als  $x \rightarrow x+1$  gelesen werden.

```
1 >>> x = y = 1
2 >>> x, y
3 (1, 1)
4 >>> x, y = 2, 3
5 >>> x, y
6 (2, 3)
7 >>> x, y = y, x
8 >>> x, y
9 (3, 2)
```

In Python ist es möglich, mehreren Variablen gleichzeitig einen Wert zuzuordnen (Zeile 1) oder mehreren Variablen in einer Anweisung verschiedene Werte zuzuordnen (Zeile 4). Statt des Tupels auf der rechten Seite von Zeile 4 könnte auch eine Liste mit zwei Elementen stehen. Tupel und Liste sind Datentypen, die mehrere Elemente enthalten und die wir im Kapitel *Zusammengesetzte Datentypen* noch genauer ansehen werden. Zeile 7 zeigt, wie man elegant die Werte zweier Variablen vertauschen kann. Dieses Verfahren ist so nicht in jeder Programmiersprache möglich. Dann muss man darauf achten, nicht einen der beiden Werte zu verlieren:

```
1 >>> x, y = 1, 2
2 >>> x = y
3 >>> y = x
4 >>> x, y
5 (2, 2)
6 >>> x, y = 1, 2
7 >>> tmp = x
8 >>> x = y
9 >>> y = tmp
10 >>> x, y
11 (2, 1)
```

In Zeile 2 wird der Wert von `x` überschrieben und geht somit verloren. In Zeile 7 wird dieser Wert dagegen in der Variablen `tmp` zwischengespeichert und kann somit in Zeile 9 der Variablen `y` zugewiesen werden.

In den Codebeispielen haben wir vor und nach dem Gleichheitszeichen ein Leerzeichen gesetzt. Dies ist nicht zwingend notwendig, verbessert aber die Lesbarkeit des Codes und wird daher auch im bereits weiter oben erwähnten Python Enhancement Proposal **PEP 8** empfohlen. Eine weitere Empfehlung lautet, eine Zeilenlänge von 79 Zeichen nicht zu überschreiten. Bei überlangen Zeilen kann man mit einem Backslash (`\`) am Zeilenende eine Fortsetzungszeile erzeugen. In gewissen Fällen erkennt der Python-Interpreter, dass eine Fortsetzungszeile folgen muss, so dass dann der Backslash entfallen kann. Dies ist insbesondere der Fall, wenn in einer Zeile eine Klammer geöffnet wird, die dann erst in einer Folgezeile wieder geschlossen wird. Es wird häufig empfohlen, den Backslash zur Kennzeichnung einer Fortsetzungszeile zu vermeiden. Stattdessen sollte eine Klammerung verwendet werden,

<sup>12</sup> Bei Bedarf kann die Liste der Schlüsselwörter mit Hilfe des `keyword`-Moduls und Verwendung der Anweisung `keyword.kwlist` erhalten werden.

selbst wenn diese ausschließlich zur impliziten Markierung von Fortsetzungszeilen dient. Im folgenden Beispiel wird deutlich, wie man die Klammerung einsetzen kann, auch wenn man die Addition kaum über zwei Zeilen verteilen wird:

```
>>> 4+
      File "<stdin>", line 1
        4+
        ^
SyntaxError: invalid syntax
>>> (4+
... 5)
9
      File "<stdin>", line 1
        4+
        ^
SyntaxError: invalid syntax
```

Beim ersten Versuch kann Python nicht erkennen, dass eine Fortsetzungszeile folgen soll, und beschwert sich entsprechend über die unvollständige Addition. Im zweiten Versuch behebt die Klammerung das Problem.

## 3.6 Wahrheitswerte

Im Kapitel *Eine Vorschau* hatten wir schon gesehen, dass man den Ablauf eines Programms in Abhängigkeit davon beeinflussen kann, dass eine bestimmte Bedingung erfüllt ist oder nicht. In diesem Zusammenhang spielen Wahrheitswerte oder so genannte boolesche Variable eine Rolle.

Mit einem Wahrheitswert kann die Gültigkeit einer Aussage mit »wahr« oder »falsch« spezifiziert werden. Mögliche Werte in Python sind entsprechend `True` und `False`, wobei die Großschreibung des ersten Zeichens wichtig ist. Für die bis jetzt behandelten Datentypen (Integer, Float, Complex) gilt, dass eine Null dem Wert `False` entspricht, während alle anderen Zahlenwerte einem `True` entsprechen. Dies lässt sich durch eine explizite Umwandlung mit Hilfe der Funktion `bool()` überprüfen:

```
>>> bool(0)
False
>>> bool(42)
True
```

Wichtige logische Operatoren sind `not` (Negation), `and` (logisches Und) und `or` (logisches Oder):

```
>>> x = True
>>> y = False
>>> not x
False
>>> x and y
False
>>> x or y
True
```

Logische Ausdrücke werden in Python von links nach rechts ausgewertet und zwar nur so weit, wie es für die Entscheidung über den Wahrheitswert erforderlich ist. Dies wird in dem folgenden Beispiel illustriert:

```
>>> x = True
>>> y = 0
>>> x or 1/y
True
>>> x and 1/y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Bei der `or`-Verknüpfung ist schon aufgrund der Tatsache, dass `x` den Wert `True` hat, klar, dass der gesamte Ausdruck diesen Wert besitzt. Die Division wird daher nicht mehr ausgewertet. Bei der `and`-Verknüpfung reicht die Information über `x` dagegen nicht aus. Zusätzlich wird die Division ausgeführt, die hier zu einer `ZeroDivisionError`-Ausnahme führt.

Wahrheitswerte sind häufig das Ergebnis von Vergleichsoperationen, die in der folgenden Tabelle zusammengefasst sind.

Operator	Bedeutung
<code>&lt;</code>	kleiner
<code>&lt;=</code>	kleiner oder gleich
<code>&gt;</code>	größer
<code>&gt;=</code>	größer oder gleich
<code>!=</code>	ungleich
<code>==</code>	gleich

```
>>> 5 != 2
True
>>> 5 > 2
True
>>> 5 == 2
False
```

❗ Ein beliebter Fehler besteht darin, beim Test auf Gleichheit nur eines statt zwei Gleichheitszeichen zu verwenden.

Bei Gleitkommazahlen ist es normalerweise nicht sinnvoll, auf Gleichheit zu prüfen, da Rundungsfehler leicht dazu führen können, dass das Ergebnis nicht wie erwartet ausfällt.

```
>>> x = 3*0.1
>>> y = 0.3
>>> x == y
False
```

In solchen Fällen ist es besser zu überprüfen, ob die Differenz von zwei Gleitkommazahlen eine vertretbare Schwelle unterschreitet.

```
>>> eps = 1e-12
>>> abs(x-y) < eps
True
```

## 3.7 Formatierung von Ausgaben

Wenn man ein Programm ausführt, möchte man in den meisten Fällen eine Ausgabe haben, die im einfachsten Fall auf dem Bildschirm erfolgen kann. Im Folgenden soll auf die Formatierung, insbesondere von Zahlen, eingegangen werden. In Python 3 erfolgt die Ausgabe auf dem Bildschirm mit Hilfe der `print`-Funktion, die wir schon im Kapitel *Eine Vorschau* erwähnt hatten.

```
>>> x = 1.5
>>> y = 3.14159
>>> print(x)
1.5
>>> print(x, y)
1.5 3.14159
>>> print("Dies ist eine Näherung für die Kreiszahl:", y)
Dies ist eine Näherung für die Kreiszahl: 3.14159
```

Wie diese Beispiele zeigen, kann man in einer Zeile mehrere Variablenwerte ausgeben, unter denen auch Zeichenketten sein können. Im Moment genügt es zu wissen, dass man Zeichenketten zum Beispiel durch umschließende Anführungszeichen kennzeichnen kann. Zeichenketten werden wir detaillierter im Kapitel *Zeichenketten* besprechen. Die letzten beiden `print`-Aufrufe zeigen, dass beim Ausdruck mehrerer Variablen automatisch ein Leer-

zeichen eingefügt wird. Wenn man zwischen Zeichenketten das Komma weglässt, kann man dieses Leerzeichen unterdrücken:

```
>>> print("Dies ist eine Näherung für die Kreiszahl:", y)
Dies ist eine Näherung für die Kreiszahl: 3.14159
```

Dies ist besonders bei langen Zeichenketten nützlich, da diese damit problemlos über mehrere Zeilen geschrieben werden können. Zu beachten ist, dass das Weglassen des Kommas nur zwischen Zeichenketten erlaubt ist. Das noch vorhandene Komma ist also erforderlich.

In Python ist es, wie in vielen anderen Programmiersprachen, möglich, die Darstellung der Variablenwerte genauer festzulegen. Auch wenn die Details von der Programmiersprache abhängig sind, gibt man typischerweise eine Zeichenkette an, die neben Text auch Platzhalter für die auszugebenden Variablen beinhaltet. Mit Hilfe dieser Platzhalter kann man auch genauer spezifizieren, wie die Ausgabe aussehen soll. Beim Übergang von Python 2 zu Python 3 wurde hierzu eine `format`-Methode eingeführt, die eine sehr flexible Formatierung erlaubt. Wir beschränken uns daher im Folgenden auf diese Art der Formatierung. Einige der damit verbundenen Möglichkeiten werden wir im weiteren Verlauf noch kennenlernen.

Bevor wir uns mit den Formatierungsmöglichkeiten von Integers und Gleitkommazahlen beschäftigen, müssen wir uns zunächst die grundsätzliche Funktionsweise der `format`-Methode ansehen.

Im einfachsten Fall hat man eine gewisse Anzahl von Variablen, die man ausgeben möchte und die im Formatierungsausdruck durch in geschweifte Klammern eingeschlossene Nummern angegeben werden.

```
>>> x = 2
>>> power = 3
>>> print("{0}**{1} = {2}".format(x, power, x**power))
2**3 = 8
>>> print("{0}**{1} = {2}. Ja, das Ergebnis ist {2}!".format(x, power, x**power))
2**3 = 8. Ja, das Ergebnis ist 8!
```

Wie die letzte Eingabe zeigt, können Platzhalter auch wiederholt werden. Ist eine Wiederholung nicht gewünscht und gibt man die Variablen in der richtigen Reihenfolge an, so kann auf die Nummerierung verzichtet werden.

```
>>> print("{}**{} = {}".format(x, power, x**power))
2**3 = 8
```

In unübersichtlichen Situationen oder wenn man die Reihenfolge später noch auf einfache Weise ändern möchte kann man auch Namen vergeben.

```
>>> print("{basis}**{exponent} = {ergebnis}".format(basis=x,
                                                    exponent=power,
                                                    ergebnis=x**power))
2**3 = 8
```

Die drei Argumente stehen hier nur in eigenen Zeilen um den langen Ausdruck geeignet umzubereiten. Wir erinnern uns daran, dass nach einer geöffneten Klammer, also der Klammer vor `basis`, bis zur schließenden Klammer weitergelesen wird.

Will man eine geschweifte Klammer im Ausgabetekst unterbringen, so muss man diese wie im folgenden Beispiel gezeigt verdoppeln.

```
>>> print("{}**{} = {}".format(x, power, x**power))
2**3 = 8. Und das ist eine geschweifte Klammer: {
```

Bis jetzt haben wir nur die Ausgabe von Variablen in einen Text eingebettet, ohne die Ausgabe jeder Variable selbst beeinflussen zu können. Dies ist aber beispielsweise bei Gleitkommazahlen wichtig.

```
>>> from math import sqrt
>>> print(sqrt(2))
1.41421356237
```

Vielleicht wollen wir jedoch gar nicht so viele Nachkommastellen ausgeben. Dies können wir mit Hilfe einer Formatspezifikation festlegen.

```
>>> print ("|{0:5.2f}|" .format(sqrt(2)))  
| 1.41|
```

Nach der Argumentnummer, die man hier auch weglassen könnte, folgt durch einen Doppelpunkt abgetrennt die Formatierungsangabe. Die Zahl vor dem Punkt gibt die minimale Feldbreite an, während die Zahl nach dem Punkt die Anzahl der Nachkommastellen angibt. Das abschließende `f` verlangt die Ausgabe in einem Format mit fester Kommastelle. Die beiden senkrechten Striche sollen nur dazu dienen, den Leerplatz vor der Zahl sichtbar zu machen, der dadurch entsteht, dass die gesamte Feldbreite gleich 5 sein soll.

```
>>> print ("|{0:.2f}|" .format(sqrt(2)))  
|1.41|
```

Lässt man die Spezifikation der Feldbreite weg, so wird die minimal benötigte Breite belegt. Bei der mehrzeiligen Ausgabe von Zahlen ist dann jedoch keine Ausrichtung nach dem Dezimalpunkt möglich. Bei der Ausrichtung ist auch das Vorzeichen relevant. Hierbei kann man angeben, wie bei einer positiven Zahl verfahren wird. Durch Eingabe eines Pluszeichens, eines Leerzeichens oder eines Minuszeichens in der Formatierungsangabe wird ein Plus, ein Leerzeichen bzw. gar nichts ausgegeben wie die folgenden Beispiele zeigen:

```
>>> print ("|{:+4.2f}|" .format(sqrt(2)))  
|+1.41|  
>>> print ("|{:+4.2f}|" .format(-sqrt(2)))  
|-1.41|  
>>> print ("|{: 4.2f}|" .format(sqrt(2)))  
| 1.41|  
>>> print ("|{: 4.2f}|" .format(-sqrt(2)))  
|-1.41|  
>>> print ("|{: -4.2f}|" .format(sqrt(2)))  
|1.41|  
>>> print ("|{: -4.2f}|" .format(-sqrt(2)))  
|-1.41|
```

Hier haben wir bewusst die Feldbreite nur auf 4 gesetzt, um den Unterschied zwischen der dritten und fünften Eingabe zu verdeutlichen.

Bei der Ausgabe von Gleitkommazahlen gibt es nun aber das Problem, dass bei sehr kleinen oder sehr großen Zahlen eine feste Anzahl von Nachkommastellen nicht unbedingt geeignet ist.

```
>>> print ("{:10.8f}" .format(sqrt(2)))  
1.41421356  
>>> print ("{:10.8f}" .format(sqrt(2)/10000000))  
0.00000014
```

In der zweiten Eingabe sieht man, dass die Zahl der ausgegebenen signifikanten Stellen dramatisch reduziert ist. In solchen Fällen bietet es sich an, eine Ausgabe in Exponentialdarstellung zu verlangen, die man mit Hilfe des Buchstabens `e` statt `f` erhält.

```
>>> print ("|{:10.8e}|" .format(sqrt(2)))  
|1.41421356e+00|  
>>> print ("|{:10.4e}|" .format(sqrt(2)))  
|1.4142e+00|  
>>> print ("|{:14.8e}|" .format(sqrt(2)/10000000))  
|1.41421356e-07|  
>>> print ("|{:20.8e}|" .format(sqrt(2)))  
|          1.41421356e+00|
```

Die erste Eingabe zeigt, wie man eine Exponentialdarstellung mit 8 Nachkommastellen erhält. Der Exponent wird mit ausgegeben, obwohl er nur für  $10^0 = 1$  steht. Die Zahl der Nachkommastellen lässt sich, wie erwartet und wie in der zweiten Eingabe zu sehen ist, bei Bedarf anpassen. In diesem Beispiel wird die Feldbreite von 10 gerade ausgenutzt. Das dritte Beispiel zeigt, dass wir nun auch bei der Ausgabe von kleinen Zahlen keine signifikanten Stellen verlieren. Entsprechendes wäre bei großen Zahlen der Fall. Macht man wie in Eingabe 3 die Feldlänge größer, so wird entsprechend viel Leerplatz auf der linken Seite ausgegeben.

**+** Um etwas über die Möglichkeiten der Positionierung der Ausgabe zu erfahren, können Sie im letzten Beispiel folgende Formatierungsspezifikationen ausprobieren: `{:<20.8e}`, `{:=+20.8e}` und `{:^20.8e}`.

Häufig möchte man die Exponentialschreibweise nur verwenden, wenn die auszugebende Zahl hinreichend groß oder klein ist. Ein solches Verhalten erreicht man durch Angabe des Buchstabens `g`.

```
>>> print ("|{:15.8g}|".format(sqrt(2)/100000))
| 1.4142136e-05|
>>> print ("|{:15.8g}|".format(sqrt(2)/10000))
| 0.00014142136|
>>> print ("|{:15.8g}|".format(sqrt(2)*10000000))
| 14142136|
>>> print ("|{:15.8g}|".format(sqrt(2)*100000000))
| 1.4142136e+08|
```

Hier wird durch den Wechsel der Darstellung insbesondere sichergestellt, dass immer die gleiche Anzahl von signifikanten Stellen ausgegeben wird.<sup>13</sup>

Betrachten wir nun noch die Formatierung von Integers.

```
>>> n = 42
>>> print ("|{0}|{0:5}|{0:05}|".format(n))
|42| 42|00042|
```

**?** Warum kann man die 0 zur Kennzeichnung der einzusetzenden Variable nicht weglassen?

Integers in Dezimaldarstellung benötigen keinen Buchstaben zur Formatspezifikation.<sup>14</sup> Man kann hier aber ähnlich wie bei den Gleitkommazahlen die Feldbreite festlegen. Gibt man vor der Feldbreite eine Null an, so wird das Feld vor der auszugebenden Zahl mit Nullen aufgefüllt. Dies kann zum Beispiel bei der Ausgabe in anderen Zahlensystemen interessant sein. Python unterstützt insbesondere die Ausgabe im Binärformat (`b`), Oktalformat (`o`) und im Hexadezimalformat (`x`).

```
>>> print ("|{0:}|{0:8b}|{0:8o}|{0:8x}|{0:08b}|".format(n))
|42| 101010| 52| 2a|00101010|
```

**?** Was ändert sich, wenn man `b`, `o` und `x` durch die entsprechenden Großbuchstaben ersetzt? Welche Auswirkungen hat ein `#`, das vor der Feldbreite inklusive einer eventuell vorhandenen Null steht?

Die hier besprochenen Formatierungsanweisungen decken bereits viele Anwendungsfälle ab. Dennoch sind die von Python 3 zur Verfügung gestellten Formatierungsmöglichkeiten noch vielfältiger. Für eine systematische und vollständige Darstellung der möglichen Formatierungen verweisen wir auf den [Abschnitt 6.1.3.1. Format Specification Mini-Language](#) in der Dokumentation der Python-Standardbibliothek.

Abschließend sei noch angemerkt, dass die `print`-Funktion standardmäßig einen Zeilenumbruch an das Ende des auszugebenden Textes anhängt. Dies ist jedoch nicht immer gewünscht und lässt sich mit Hilfe der Option `end` beeinflussen. Folgendes Beispiel zeigt die Funktionsweise. Die Befehle

```
print("x", end="..")
print("x", end="")
print("x")
```

erzeugen die Ausgabe `x . .xx` sowie einen anschließenden Zeilenumbruch.

<sup>13</sup> Die genaue Regel für die Umstellung der Darstellungsart kann man in der Dokumentation der Python-Standardbibliothek unter [6.1.3.1. Format Specification Mini-Language](#) nachlesen.

<sup>14</sup> Wenn man unbedingt möchte, kann man `d` für Dezimaldarstellung angeben.

---

## Kontrollstrukturen

---

Im Kapitel *Eine Vorschau* hatten wir bereits kurz die Möglichkeit angesprochen, den Ablauf eines Programms zu beeinflussen, sei es dadurch, dass ein Programmteil in einer Schleife mehrfach ausgeführt wird oder dass ein Programmteil nur dann ausgeführt wird, wenn eine gewisse Bedingung erfüllt ist. Wir werden nun solche Kontrollstrukturen genauer betrachten und neben den bereits angesprochenen `for`-Schleifen und Konstrukten der Form `if ... else` auch `while`-Schleifen und komplexere Verzweigungen kennenlernen.

### 4.1 For-Schleife

Sollen bestimmte Anweisungen mehrfach ausgeführt werden, wobei die Anzahl der Wiederholungen zuvor bestimmt werden kann, bietet sich die Verwendung einer `for`-Schleife an. Gegenüber der expliziten Wiederholung von Befehlen ergeben sich eine Reihe von Vorteilen. Zunächst einmal spart man sich Tipparbeit und verbessert erheblich die Lesbarkeit des Programms. Zudem ist eine explizite Wiederholung nur möglich, wenn die Zahl der Wiederholungen bereits beim Schreiben des Programms feststeht und nicht erst beim Ausführen des Programms berechnet wird. Im Kapitel *Eine Vorschau* hatten wir bereits die wesentliche Struktur einer `for`-Schleife kennengelernt, die wir hier nochmals an einem einfachen Beispiel illustrieren wollen.

```
for n in range(5):
    print("{:4} {:4}".format(n, n**2))

0      0
1      1
2      4
3      9
4     16
```

Das Schlüsselwort `for` kennzeichnet den Beginn einer Schleife. Dann folgt der Name der Variable, die bei der Abarbeitung der Schleife vorgegebene Werte annimmt, und im Rahmen der Schleife verwendet werden kann. Im Allgemeinen können hier auch mehrere Variablennamen vorkommen, wie wir später im Kapitel *Zusammengesetzte Datentypen* sehen werden. Die Werte, die die Variable `n` in unserem Beispiel annehmen kann, werden durch die `range`-Anweisung bestimmt. Zwar werden die Werte erst bei Bedarf generiert, aber wir können sie uns wie folgt ansehen:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

Es wird also eine Liste von aufeinanderfolgenden ganzen Zahlen erzeugt, die hier fünf Elemente enthält. Zu beachten ist, dass die Liste mit Null beginnt und nicht mit Eins. Wir werden uns im Abschnitt *Listen* diesen Datentyp noch genauer ansehen. Für den Moment genügt jedoch die intuitive Vorstellung von einer Liste. In der ersten Zeile der `for`-Schleife, die mit einem Doppelpunkt enden muss, wird also festgelegt, welchen Wert die Schleifenvariable bei den aufeinanderfolgenden Schleifendurchläufen jeweils annimmt.

Der Codeteil, der im Rahmen der Schleife im Allgemeinen mehrfach ausgeführt wird und im obigen Beispiel nur aus einer Zeile besteht, ist daran zu erkennen, dass er eingerückt ist. Vergleichen wir zur Verdeutlichung die beiden Codestücke

```
for n in range(2):  
    print("Schleifendurchlauf {}".format(n+1))  
    print("Das war's.")
```

und

```
for n in range(2):  
    print("Schleifendurchlauf {}".format(n+1))  
print("Das war's.")
```

so erhalten wir im ersten Fall die Ausgabe

```
Schleifendurchlauf 1  
Das war's.  
Schleifendurchlauf 2  
Das war's.
```

während die Ausgabe im zweiten Fall

```
Schleifendurchlauf 1  
Schleifendurchlauf 2  
Das war's.
```

lautet. Im ersten Codestück gehört die zweite `print`-Anweisung also noch zur Schleife, während dies im zweiten Codestück nicht der Fall ist. Im Prinzip ist die Zahl der Leerstellen bei Einrückungen unerheblich, sie muss allerdings innerhalb einer Schleife immer gleich sein. Das **PEP 8**<sup>1</sup>, in dem Konventionen für das Programmieren in Python festgelegt sind, empfiehlt, um vier Leerzeichen einzurücken. Dies ist ein guter Kompromiss zwischen kaum sichtbaren Einrückungen und zu großen Einrückungen, die bei geschachtelten Schleifen schnell zu Platzproblemen führen. Tabulatorzeichen sind zwar prinzipiell bei Einrückungen erlaubt. Es muss aber auf jeden Fall vermieden werden, Leerzeichen und Tabulatorzeichen zu mischen. Am besten verzichtet man ganz auf die Verwendung von Tabulatorzeichen.

Da die Verwendung der Einrückung als syntaktisches Merkmal ungewöhnlich ist, wollen wir kurz zwei Beispiele aus anderen Programmiersprachen besprechen. In FORTRAN 90 könnte eine Schleife folgendermaßen aussehen:

```
PROGRAM Quadrat  
DO n = 0, 4  
    PRINT '(2I4)', n, n**2  
END DO  
END PROGRAM Quadrat
```

Hier wurde nur aus Gründen der Lesbarkeit eine Einrückung vorgenommen. Relevant für das Ende der Schleife ist lediglich das abschließende `END DO`. Während man sich hier selbst dazu zwingen muss, gut lesbaren Code zu schreiben, zwingt Python den Programmierer durch seine Syntax dazu, übersichtlichen Code zu produzieren.

Auch im folgenden C-Code sind die Einrückungen nur der Lesbarkeit wegen vorgenommen worden. Der Inhalt der Schleife wird durch die öffnende geschweifte Klammer in Zeile 6 und die schließende geschweifte Klammer in Zeile 9 definiert. Würde man auf die Klammern verzichten, so wäre nur die der `for`-Anweisung folgende Zeile, also Zeile 7, Bestandteil der Schleife.

```
1  #include <stdio.h>  
2  
3  main() {  
4      int i;  
5      int quadrat;  
6      for(i = 0; i < 5; i++){  
7          quadrat = i*i;  
8          printf("%4i %4i\n", i, quadrat);  
9      }  
10 }
```

---

<sup>1</sup> PEP steht für »Python Enhancement Proposal«.



Kehren wir jetzt aber zu Python zurück und nehmen wir an, dass wir uns nur für gerade Zahlen interessieren und für das kleine Einmaleins nicht den Computer bemühen müssen. Dann können wir der `range()`-Funktion auch einen Startwert und eine Schrittweite vorgeben:

```
for n in range(20, 26, 2):  
    print(n, n**2)
```

Die zugehörige Ausgabe lautet:

```
20 400  
22 484  
24 576
```

Die Schleife wird also nur so lange ausgeführt, wie die Iterationsvariable kleiner als das zweite Argument ist. Bereits in dem eingangs betrachteten Beispiel war das Argument der `range()`-Funktion nicht in der Liste enthalten.

Statt mit der `range()`-Funktion eine Zahlenfolge zu erzeugen, können wir eine Liste mit den gewünschten Objekten vorgeben, wie dieses Beispiel

```
zahlen = [12, 17, 23]  
for n in zahlen:  
    print(n, n**2)
```

mit der Ausgabe

```
12 144  
17 289  
23 529
```

zeigt. Listen werden grundsätzlich durch eckige Klammern begrenzt und können auch Objekte mit verschiedenen Datentypen enthalten, wie wir im Kapitel *Zusammengesetzte Datentypen* sehen werden.

In den obigen Beispielen haben wir die Schleifenvariable mit `n` bezeichnet. Im Allgemeinen ist es aber besser, nach Möglichkeit einen aussagekräftigeren Namen zu verwenden, der sich aus der konkreten Anwendung ergibt. In dem speziellen Fall, in dem die Schleifenvariable bei der Abarbeitung der Schleife nicht verwendet wird, kann zur Verdeutlichung dieses Umstands der Unterstrich `_` als Variablenname verwendet werden. Hierbei handelt es sich, wie wir aus dem Kapitel *Variablen und Zuweisungen* wissen, um einen erlaubten Bezeichner.

```
>>> for _ in range(3):  
...     print("Python ist toll!")  
...  
Python ist toll!  
Python ist toll!  
Python ist toll!
```

Es sei jedoch ausdrücklich davon abgeraten, einen Unterstrich für Variablen zu verwenden, die später noch benötigt werden.

Gerade bei der Entwicklung eines Programms kann es vorkommen, dass man eine Schleife vorbereitet, den Schleifeninhalt noch nicht geschrieben hat, aber dennoch ein syntaktisch korrektes Programm haben möchte. Da der Schleifeninhalt nicht leer sein darf, verwendet man in einem solchen Fall die Anweisung `pass`, die sonst keine weiteren Auswirkungen hat. In dem Beispiel

```
for n in range(10):  
    pass
```

gibt es also keinen wirklichen Schleifeninhalt. Allerdings muss man sich darüber im Klaren sein, dass dennoch die Schleife abgearbeitet wird. Dabei werden in diesem Beispiel der Variablen `n` nacheinander die Werte von 0 bis 9 zugeordnet. Nach Abarbeitung der Schleife hat `n` also den Wert 9.

Eine typische Form der Anwendung einer `for`-Schleife ist im folgenden Beispiel gezeigt.

```
1 from math import sqrt  
2
```

```

3  summe = 0
4  for n in range(100000):
5      summe = summe+1/(n+1)**2
6  print(sqrt(6*summe))

```

Konzentrieren wir uns auf die Zeilen 3-5, deren Ziel es ist, eine unendliche Summe, die  $\pi^2/6$  ergibt, durch Beschränkung auf die ersten 100000 Terme näherungsweise auszuwerten. Nachdem uns die Form der Zeilen 4 und 5 inzwischen schon gut vertraut ist, betrachten wir insbesondere die Zeile 3. Hier wird, wie dies häufig bei Schleifen erforderlich ist, zunächst eine Variable, hier die Summationsvariable `summe`, initialisiert. Vergisst man dies, so ist diese Variable beim ersten Schleifendurchlauf auf der rechten Seite der Zeile 5 undefiniert, was zu einem `NameError` führt. Es ist also keineswegs so, dass Variablen auf magische Weise mit einer Null vorbelegt werden. Bei der Verwendung von Schleifen muss man also immer auch an die eventuell erforderliche Initialisierung von Variablen denken, die in der Schleife verwendet werden.

`for`-Schleifen können auch ineinander geschachtelt werden, wie folgendes Beispiel zeigt, das die Wahrheitstabelle der logischen UND-Verknüpfung mit Hilfe des `&`-Operators erzeugt.

```

print(" arg1 arg2      arg1 & arg2 ")
print("-----")
for arg1 in [0, 1]:
    for arg2 in [0, 1]:
        print("      {}      {}      {}".format(arg1, arg2, arg1 & arg2))

```

Die zugehörige Ausgabe lautet:

```

arg1 arg2      arg1 & arg2
-----
0      0          0
0      1          0
1      0          0
1      1          1

```

Wie man sieht, wird zunächst durch die äußere Schleife die Variable `arg1` auf den Wert 0 gesetzt. Dann wird die innere Schleife abgearbeitet und `arg2` durchläuft die Werte 0 und 1. Anschließend schreitet die äußere Schleife einen Schritt voran, wobei `arg1` auf den Wert 1 gesetzt wird. Dann läuft wiederum die innere Schleife über die Werte 0 und 1, so dass sich insgesamt vier Ausgabezeilen ergeben. Hinzu kommen die zu Beginn des Codes definierten zwei Ausgabezeilen, die den Kopf der Ausgabe bilden. Wichtig ist, dass die Einrückungen entsprechend den Zugehörigkeiten zu den jeweiligen Schleifen korrekt vorgenommen werden. So ist die innere Schleife Bestandteil des Codeblocks der äußeren Schleife und daher eingerückt. Die `print`-Anweisung gehört zum Codeblock der inneren Schleife und wurde somit entsprechend weiter eingerückt. Die Einrückung in der letzten Zeile, einer Fortsetzungszeile, ist dagegen willkürlich und vor allem so gewählt, dass die Lesbarkeit möglichst gut ist.

## 4.2 While-Schleife

Nicht immer kennt man vorab die Zahl der benötigten Schleifendurchläufe, sondern möchte die Beendigung einer Schleife von der Nichterfüllung einer Bedingung abhängig machen. Dann ist eine `while`-Schleife das Mittel der Wahl.

Als Beispiel betrachten wir eine physikalische Problemstellung. Beim schiefen Wurf beginne eine Punktmasse ihre Bahn am Ort  $x=0, y=0$  mit der Anfangsgeschwindigkeit  $(v_x0, v_y0)$ . Die Bahn soll für Zeiten in Schritten von  $dt$  so lange bestimmt werden wie die  $y$ -Koordinate nicht negativ ist. In der Praxis wäre die Problemstellung wahrscheinlich so kompliziert, dass man die newtonsche Bewegungsgleichung numerisch lösen müsste. In dem vorliegenden Fall kennen wir die Lösung analytisch und können sie im Programm verwenden.

Die Berechnung der Bahnkurve könnte durch folgendes Programm geschehen:

```

1  t = 0          # Startzeit
2  dt = 0.01      # Zeitschritt
3  g = 9.81       # Erdbeschleunigung in m/s²
4  x0 = 0         # horizontale Ausgangsposition in m

```

```

5  y0 = 0      # Anfangshöhe in m
6  vx0 = 2     # Anfangsgeschwindigkeit in x-Richtung in m/s
7  vy0 = 1     # Anfangsgeschwindigkeit in y-Richtung in m/s
8
9  x = x0
10 y = y0
11 print(" t      x      y")
12 print("-----")
13 while y >= 0:
14     print("{:4.2f}    {:4.2f}    {:8.6f}".format(t, x, y))
15     t = t + dt
16     x = x0 + vx0*t
17     y = y0 + vy0*t - 0.5*g*t**2

```

In den Zeilen 1-7 wird zunächst eine Reihe von Parametern festgelegt. Hierzu gehören die Größen, die auch in einer physikalischen Problemstellung benötigt werden, nämlich Ort ( $x_0$ ,  $y_0$ ) und Geschwindigkeit ( $v_{x0}$ ,  $v_{y0}$ ) zum Anfangszeitpunkt. Außerdem benötigen wir den Wert der Erdbeschleunigung  $g$ . Der Anfangszeitpunkt bestimmt den anfänglichen Wert der Zeitvariable  $t$ .

Da uns das Programm die Bahn nicht für eine kontinuierlich verlaufende Zeit liefern kann, müssen wir eine diskrete Zeit einführen. Hierfür legen wir den Abstand  $dt$  zwischen aufeinanderfolgenden Zeitpunkten fest.

In den Zeilen 9 und 10 wird der Ort zur Anfangszeit mit den Anfangsbedingungen belegt. Damit ist auch beim ersten Test der `while`-Bedingung die Variable `y` definiert. Zudem benötigen wir diese Werte für die Ausgabe in Zeile 14. Die `print`-Anweisungen in den Zeilen 11 und 12 dienen dazu, die Ausgabe selbsterklärend zu machen.

Der uns hier eigentlich interessierende Programmteil beginnt in Zeile 13 mit der `while`-Anweisung. Da wir uns mit der Berechnung der Wurfparabel für nicht negative Höhen begnügen wollen, lautet die zu erfüllende Bedingung  $y \geq 0$ . In Zeile 14 wird der aktuelle Bahnpunkt, für den ja überprüft wurde, dass der Wert der Variable `y` nicht negativ ist, ausgegeben. In Zeile 15 wird die Zeit um den vorgegebenen Zeitschritt erhöht und in den Zeilen 16 und 17 der zugehörige Bahnpunkt bestimmt.

**?** Wie würde sich die Ausgabe verändern, wenn man die Zeilen 14 und 15 hinter die Zeile 17 verschiebt, also zunächst den Bahnpunkt berechnet und ausgibt und anschließend die Zeit inkrementiert?

In den Zeilen 1-7 fällt noch auf, dass nach den Anweisungen ein „Gartenzaun“ (`#`) und eine Erläuterung folgt. In Python wird das Zeichen `#` als Kommentarzeichen interpretiert, so dass der gesamte Text nach diesem Zeichen ignoriert wird. Welches Zeichen als Kommentarzeichen dient, hängt von der Programmiersprache ab. In C++ beispielsweise wird diese Funktion von zwei Zeichen, nämlich `//`, übernommen. Kommentare sind eine Möglichkeit um sicherzustellen, dass die Funktionsweise eines Programms auch nach längerer Zeit leicht verstanden werden kann. **PEP 8** gibt Hinweise zur Verwendung und Formatierung von Kommentaren. Es lohnt sich, einen Blick hineinzuworfen, auch wenn man nicht alle Regeln immer zwingend befolgen muss.<sup>2</sup> Zum Beispiel sind unsere Kommentare auf deutsch geschrieben, was natürlich nur sinnvoll ist, wenn alle potentiellen Leser des Programms die deutsche Sprache beherrschen. Auch die Verwendung von Umlauten ist nicht immer empfehlenswert. In jedem Fall sollte aber der folgende Hinweis aus dem **PEP 8** beherzigt werden:

“Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!”

Andernfalls besteht die Gefahr, den Leser des Kommentars, also unter Umständen sich selbst, auf eine falsche Fährte zu setzen.

Als Anmerkung sei noch erwähnt, dass andere Sprachen eventuell ein `do...while`-Konstrukt (z.B. C) oder ein `repeat...until`-Konstrukt (z.B. Pascal) zur Verfügung stellen. Dort erfolgt der Abbruchtest nicht zu Beginn, sondern am Ende. Daher wird der Code auf jeden Fall einmal durchlaufen.

Ein Beispiel hierfür könnte in C folgendermaßen aussehen:

```

1  #include <stdio.h>
2
3  main() {

```

<sup>2</sup> Siehe hierzu den Abschnitt *A Foolish Consistency is the Hobgoblin of Little Minds* des **PEP 8**. Nebenbemerkung: Dieser Titel spielt auf ein Zitat von Ralph Waldo Emerson an.

```

4     int i=-1;
5     do {printf("%4i %4i\n", i, i*i);
6         i = i-1;
7     } while (i>0);
8 }

```

In Zeile 4 wird die Variable `i` mit `-1` initialisiert. Da der Test auf Positivität von `i` erst am Ende der Schleife erfolgt, wird die Schleife einmal durchlaufen und somit eine Zeile mit dem Inhalt `-1 1` ausgegeben. In Pascal würde man das Gleiche mit diesem Code erhalten:

```

program Quadrat;
var
    i: integer;
begin
    i := -1;

    repeat
        writeln(i, ' ', i*i);
        i := i-1;
    until i <= 0;

end.

```

Abschließend sei betont, dass der Programmierer bei der Verwendung von `while`-Schleifen und ähnlichen Konstrukten selbst dafür verantwortlich ist sicherzustellen, dass die Schleife irgendwann beendet wird. Andernfalls liegt eine Endlosschleife vor und das Programm muss von außen abgebrochen werden. Dieses Szenario kann allerdings gezielt bei Programmen eingesetzt werden, die durch äußere Ereignisse wie Tastendrucke oder Mausbewegungen gesteuert werden. In diesem Fall durchläuft das Programm eine Endlosschleife, um bei Bedarf auf äußere Ereignisse adäquat zu reagieren. Aber auch in diesem Fall ist darauf zu achten, dass es eine Möglichkeit gibt, das Programm kontrolliert, beispielsweise durch Drücken der Taste `q`, zu beenden. Im Python-Code verwendet man dann den `break`-Befehl, um die Ausführung des Programmcodes außerhalb der Schleife fortzusetzen.

## 4.3 Verzweigungen

Eine andere Art von Kontrollstrukturen, bei denen es nicht um die mehrfache Ausführung von Code geht, sind Verzweigungen. Dabei wird ein Code nur dann ausgeführt, wenn eine vorgegebene Bedingung erfüllt ist. Im einfachsten Fall geschieht dies mit der `if`-Anweisung wie folgendes Beispiel zeigt:

```

>>> x = 5
>>> if x != 0:
...     print("Der Kehrwert von {} ist {}".format(x, 1./x))
...
Der Kehrwert von 5 ist 0.2.
>>> x = 0
>>> if x != 0:
...     print("Der Kehrwert von {} ist {}".format(x, 1./x))
...

```

Im ersten Fall wird der Code des `if`-Blocks ausgeführt, im zweiten Fall dagegen nicht. Wie schon bei den `for`- und `while`-Konstrukten ist der Umfang des Codeblocks, der nur bedingt ausgeführt wird, durch die Einrückung bestimmt, kann also auch mehr als eine Zeile umfassen. Ist der Code nur eine Zeile lang, darf er auch direkt hinter der `if`-Anweisung stehen.

```

>>> x = -5
>>> if x < 0: x = -x
...
>>> print("Diese Zahl ist bestimmt nicht negativ:", x)
Diese Zahl ist bestimmt nicht negativ: 5

```

Die `print`-Anweisung wird in diesem Fall immer ausgeführt.

In diesen Beispielen ist die Bedeutung der verwendeten Bedingung ziemlich offensichtlich. Gelegentlich können aber auch kompliziertere Bedingungen auftreten, deren Bedeutung sich nicht ohne Weiteres erschließt. Dann kann es sinnvoll sein, eine Variable für den entsprechenden Wahrheitswert nach folgendem Muster einzuführen:

```
1 x = -5
2 zahl_ist_negativ = x<0
3 if zahl_ist_negativ:
4     x = -x
```

In Zeile 2 dokumentiert die Bezeichnung der Wahrheitswertvariable die Bedeutung des logischen Ausdrucks auf der rechten Seite.

Häufig möchte man abhängig von einer Bedingung eine von zwei Alternativen ausführen lassen. Hierfür verwendet man das `if ... else ...`-Konstrukt:

```
>>> x = 0
>>> if x != 0:
...     print("Der Kehrwert von {} ist {}".format(x, 1/x))
... else:
...     print("Ich weiß nicht wie man durch Null dividiert.")
...
Ich weiß nicht wie man durch Null dividiert.
```

Verzweigungen lassen sich auch schachteln.

```
1 if x < 0:
2     print("x ist negativ.")
3 else:
4     if x == 0:
5         print("x ist gleich Null.")
6     else:
7         print("x ist positiv.")
```

Wie die Einrückung zeigt, stellen die Zeilen 4-7 den Codeblock der ersten `else`-Anweisung dar. Da solche Konstruktionen häufig vorkommen, stellt Python hier eine `elif`-Anweisung zur Verfügung, mit der sich der Code folgendermaßen schreiben lässt:

```
1 if x < 0:
2     print("x ist negativ.")
3 elif x == 0:
4     print("x ist gleich Null.")
5 else:
6     print("x ist positiv.")
```

Um die Programmlaufzeit zu optimieren, sollten die Abfragen so formuliert werden, dass in der Mehrheit der Fälle bereits die erste Bedingung erfüllt ist. Muss man davon ausgehen, dass die Variable `x` in den meisten Fällen positiv ist, so wäre es sinnvoll, den obigen Code umzuformulieren.

Man kann sich durchaus mehrere Schachtelungsebenen vorstellen, die mit entsprechend vielen `elif`-Anweisungen realisiert werden. Einige Programmiersprachen stellen hierfür eine so genannte `case`-Anweisung zur Verfügung. In Python gibt es eine solche Anweisung allerdings nicht. Mit Hilfe des Datentyps `dictionary`, den wir im Abschnitt [Dictionaries](#) noch kennenlernen werden, lässt sich aber ein effizienter Ersatz schaffen, der eine lange Hierarchie von Abfragen vermeidet.

In Verzweigungen wie im obigen Beispiel, vor allem aber, wenn man mehrere `elif`-Blöcke vorliegen hat, sollte man beachten, dass nach der Abarbeitung eines Blockes an das Ende der gesamten `if`-Konstruktion gesprungen wird. Dies gilt auch dann, wenn eine später folgende Bedingung erfüllt ist, wie das folgende Beispiel illustriert.

```
x = 2
if x < 0:
    print("x ist negativ.")
elif x == 2:
    print("x ist gleich 2.")
elif x > 0:
    print("x ist größer als Null.")
```

Hier ist die zweite Bedingung erfüllt, so dass der Text »x ist gleich 2.« ausgegeben wird. Im weiteren Programmablauf wird die dritte Bedingung in diesem Fall überhaupt nicht ausgewertet.

## 4.4 Abfangen von Ausnahmen

In einem der Beispiele des vorigen Abschnitts haben wir sichergestellt, dass die Variable ungleich Null ist, bevor wir den Kehrwert gebildet haben <sup>3</sup>. In Python verwendet man gerne eine Alternative, die davon ausgeht, dass es leichter ist, um Verzeihung zu bitten also um Erlaubnis <sup>4</sup>. Im Abschnitt *Gleitkommazahlen* hatten wir bereits gesehen, dass bei einer Division durch Null eine `ZeroDivisionError`-Ausnahme (engl. *Exception*) geworfen wird. Diese Ausnahme kann folgendermaßen abgefangen werden:

```
1 from math import sin
2 x = 0
3 try:
4     y = sin(x)/x
5 except ZeroDivisionError:
6     y = 1.
7 print(y)
```

Als Ausgabe ergibt sich 1.0. Zunächst wird versucht, den Codeblock nach der `try`-Anweisung, also hier den Code in Zeile 4, auszuführen. Wird dabei eine `ZeroDivisionError`-Ausnahme geworfen, so wird der Programmablauf im Codeblock der `except`-Anweisung, also in Zeile 6, fortgesetzt. Dies gilt jedoch nur für die in der `except`-Anweisung angegebene(n) Ausnahme(n). Wäre z.B. die Variable `x` vom Typ `String`, so würde das Programm mit einer `TypeError`-Ausnahme abbrechen. Gibt man nach der `except`-Anweisung keine Ausnahme(n) an, so wird der Programmablauf beim Auftreten einer Ausnahme unabhängig von deren Typ in Zeile 6 fortgesetzt. Auch wenn ein solches Vorgehen zunächst als sehr bequem erscheint, ist es aus mehreren Gründen nicht ratsam. Zum einen würde die Behandlung beispielsweise einer `TypeError`-Ausnahme durch den Code in Zeile 6 nicht korrekt sein und somit sehr wahrscheinlich zu einem fehlerhaften Ergebnis führen. Zum anderen tritt der fehlerhafte Typ der Variable `x`, der seinen Ursprung wohl in einem Programmierfehler hat, nicht mehr in Erscheinung, so dass der Programmierfehler möglicherweise unentdeckt bliebe.

**+** Das `try...except...finally...`-Konstrukt erlaubt auch noch einen `finally...`-Block, der unabhängig vom Auftreten einer Ausnahme ausgeführt wird. Dies ist zum Beispiel wichtig, wenn Dateizugriffe erfolgen und die Datei am Ende auf jeden Fall geschlossen werden muss.

<sup>3</sup> Diese Strategie wird gelegentlich mit dem Akronym LBYL = "look before you leap" belegt.

<sup>4</sup> Das Gegenteil zu LBYL ist EAFP = "easier to ask forgiveness than permission".

## Funktionen

Funktionen im mathematischen Sinne sind uns in Physik und Materialwissenschaften wohlbekannt. Dass solche Funktionen auch von Programmiersprachen zur Verfügung gestellt werden, haben wir z.B. im Abschnitt *Funktionen für reelle Zahlen* gesehen. Allerdings ist der Begriff der Funktionen in Programmiersprachen wesentlich weiter gefasst. So muss eine Funktion nicht unbedingt ein Argument besitzen, und sie muss auch nicht unbedingt einen Wert zurückgeben. In manchen Sprachen wird nach diesem letzten Kriterium unterschieden. So kennt FORTRAN *functions*, die ein Resultat zurückgeben, und *subroutines*, die dies nicht tun.

Wozu sind Funktionen gut, wenn man einmal davon absieht, dass Funktionen dem Naturwissenschaftler vertraut sind? Funktionen eignen sich vor allem dazu, Programmcode, der sonst im Programm mehrfach auftreten würde, an einer einzigen Stelle unterzubringen. Wählt man geeignete Funktionsnamen, so kann dies zum einen die Lesbarkeit des Codes deutlich steigern. Zum andern verbessert sich auch die Wartbarkeit erheblich, da Korrekturen nur an einer Stelle vorgenommen werden müssen. Andernfalls muss man auf eine konsistente Korrektur des Programms an verschiedenen Stellen achten. Stellt man also beim Programmieren fest, dass sich Code wiederholt, so sollte man sich überlegen, ob es nicht sinnvoll wäre, für die betreffende Aufgabe eine Funktion zu definieren. Auch ähnliche Aufgaben kann man in einer Funktion unterbringen, wenn man geeignete Funktionsargumente einführt. Es hilft, wenn man sich beim Programmieren immer daran erinnert, sich nicht unnötig zu wiederholen.<sup>1</sup> Schließlich ist die Verwendung von Funktionen auch dann angebracht, wenn man die gleiche Funktionalität in verschiedenen Programmen benötigt. Man kann solche Funktionen in einem Modul sammeln und dann bei Bedarf importieren, wie wir es schon mit dem `math`-Modul und den darin enthaltenen Funktionen getan haben.

### 5.1 Funktionsdefinitionen

Betrachten wir zunächst eine Funktion, die weder ein Argument besitzt noch Daten zurückgibt.<sup>2</sup>

```

1 def f():
2     print("in der Funktion f")
3
4 print("** Anfang")
5 f()
6 print("** Ende")

```

Hier wird in den Zeilen 1 und 2 eine Funktion definiert. In der ersten Zeile der Definition steht dabei zunächst das Schlüsselwort `def`. Dann folgt der Funktionsname, der im Rahmen der auch für Variablen geltenden Regeln gewählt werden kann. Das anschließende Klammerpaar ist hier leer, weil keine Argumente übergeben werden. Andernfalls sind hier die Funktionsargumente aufzuführen. Auf das Klammerpaar kann in Python in keinem Fall verzichtet werden. Die Zeile muss mit einem Doppelpunkt enden. Der Inhalt der Funktionsdefinition ergibt sich wie immer in Python aus der Einrückung. Die nicht mehr eingerückte Zeile 4 gehört somit nicht mehr zur Funktionsdefinition. Die leere Zeile 3 spielt bei der Interpretation des Codes keine Rolle, sondern dient nur der Übersichtlichkeit.

<sup>1</sup> Dies ist das OAAO-Prinzip: once and only once.

<sup>2</sup> Genau genommen hat auch eine solche Funktion einen Rückgabewert, nämlich `None`. Dies lässt sich überprüfen, indem man den Rückgabewert mit `x = f()` einer Variablen zuweist und diese anschließend ausdrucken lässt.

Wenn der Pythoninterpreter dieses Beispielprogramm ausführt, wird zunächst die Funktion `f` definiert. Dabei wird jedoch noch nicht der darin enthaltene Code ausgeführt. Ab Zeile 4 werden dann die Anweisung ausgeführt, die Anweisung in Zeile 5 weist den Interpreter an, den Funktionscode, hier die Zeile 2, auszuführen. Anschließend wird mit der Abarbeitung des Programms in Zeile 6 fortgefahren. Entsprechend lautet die Ausgabe dieses Programms

```
** Anfang
in der Funktion f
** Ende
```

Einer Funktion können auch ein oder mehrere Argumente übergeben werden. Sehen wir uns wieder ein einfaches Beispiel an.

```
1 def f(k):
2     print("Das Quadrat von {} ist {}".format(k, k**2))
3
4 for n in range(3):
5     f(n)
```

In den Zeilen 1 und 2 wird die Funktion `f` definiert, deren Aufgabe darin besteht, das Quadrat des Arguments `k` zu bestimmen und in geeigneter Weise auszugeben. In den Zeilen 4 und 5 wird diese Funktion dann in einer Schleife dreimal mit folgendem Ergebnis ausgeführt:

```
Das Quadrat von 0 ist 0.
Das Quadrat von 1 ist 1.
Das Quadrat von 2 ist 4.
```

Alternativ könnte man die Funktion den Funktionswert berechnen und an den aufrufenden Programmteil zurückgeben lassen. Die gleiche Ausgabe lässt sich also auch folgendermaßen erhalten:

```
def f(k):
    ergebnis = k**2
    return ergebnis

for n in range(3):
    print("Das Quadrat von {} ist {}".format(n, f(n)))
```

Im Gegensatz zu einigen anderen Programmiersprachen ist die `return`-Anweisung nicht erforderlich, um das Ende der Funktionsanweisung zu kennzeichnen. Dies geschieht in Python ausschließlich mit Hilfe der Einrückung. Die `return`-Anweisung wird jedoch benötigt, um Ergebnisse an den aufrufenden Code zurückzugeben. Die gerade vorgestellte Funktionsdefinition kann noch kompakter gestaltet werden, da die `return`-Anweisung nicht nur eine Variable, sondern einen ganzen Ausdruck enthalten kann:

```
def f(k):
    return k**2
```

Bei Bedarf kann eine Funktion auch mehrere `return`-Anweisungen enthalten, wie in dem folgenden Beispiel. Die Funktion `is_prime()` soll feststellen, ob es sich bei der Integer-Variable `n` um eine Primzahl handelt.

```
def is_prime(n):
    for divisor in range(2, n):
        if n % divisor == 0:
            return False
    return True

for n in range(2, 20):
    if is_prime(n):
        print(n)
```

**?** Der hier vorgestellte Primzahltest ist nicht sonderlich effizient. Wie könnte man ihn verbessern?

Funktionen können auch mehr als ein Argument besitzen und mehr als einen Wert zurückgeben wie folgendes Beispiel zeigt:



```
def vektorfeld(x, y):
    ax = -y
    ay = x
    return ax, ay

for x in range(2):
    for y in range(2):
        vx, vy = vektorfeld(x, y)
        print("{:2},{:2} -> {:2},{:2}".format(x, y, vx, vy))
```

Wichtig ist, dass die Zuordnung durch die Position der jeweiligen Variablen erfolgt, nicht durch deren Namen. Es gibt allerdings auch die Möglichkeit einer namensbasierten Übergabe der Argumente, die wir im Abschnitt *Schlüsselworte und Defaultwerte* kennenlernen werden.

**+** Eine Funktionsdefinition kommt nicht ohne einen Codeblock aus, der auf die mit `def` beginnende Deklarationszeile folgt. Gelegentlich möchte man beim Entwickeln eines Programms bereits die zu erstellenden Funktionen notieren, ohne die entsprechende Funktionalität gleich zu implementieren. Um dennoch schon zu diesem Zeitpunkt ein syntaktisch korrektes Programm zu haben, kann man den Befehl `pass` verwenden, der lediglich dazu dient, den benötigten Codeblock bereitzustellen. `pass` hat ansonsten keinerlei Auswirkungen auf den Programmablauf.

**+** Auch wenn eine Funktion keine `return`-Anweisung enthält, wird ein Wert zurückgegeben, nämlich `None` wie folgendes einfache Beispiel zeigt.

```
>>> def f():
...     pass
...
>>> print(f())
None
```

Gelegentlich kann es vorkommen, dass man sich über diese unerwartete Rückgabe wundert. Grund hierfür ist dann häufig eine vergessene `return`-Anweisung.

## 5.2 Dokumentation von Funktionen

Wie generell beim Programmieren ist es auch in Funktionen sinnvoll, an eine ausreichende Dokumentation zu denken. Dies könnte mit Hilfe von Kommentaren erfolgen, die mit einem `#` eingeleitet werden. In Python gibt es für Funktionen jedoch eine geeignetere Art der Dokumentation, nämlich einen Dokumentationsstring, der direkt auf die erste Zeile der Funktionsdefinition folgt. Das könnte beispielsweise folgendermaßen aussehen:

```
from math import sqrt

def mitternacht(a, b, c):
    """Berechne die beiden Lösungen einer quadratischen Gleichung
    ax^2+bx+c=0.

    Es wird die Mitternachtsformel verwendet.
    Achtung: Es wird stillschweigend vorausgesetzt, dass b^2-4ac>0.

    """
    diskriminante = sqrt(b**2-4*a*c)
    root1 = (-b+diskriminante)/(2*a)
    root2 = (-b-diskriminante)/(2*a)
    return root1, root2
```

Der Dokumentationsstring wird hier nicht nur mit einem, sondern mit drei Anführungszeichen begrenzt, da er dann über mehrere Zeilen gehen kann. Der Vorteil dieser Dokumentationsweise besteht darin, dass dieser Dokumentationstext mit dem Befehl `help(mitternacht)` ausgegeben werden kann, wie wir im Abschnitt *Gleitkommazahlen* schon gesehen hatten, wo `help(math)` die Dokumentation für das Modul `math` ausgab.

```
>>> help(mitternacht)
```

```
Help on function mitternacht in module __main__:
```

```
mitternacht(a, b, c)
    Berechne die beiden Lösungen einer quadratischen Gleichung  $ax^2+bx+c=0$ .

    Es wird die Mitternachtsformel verwendet.
    Achtung: Es wird stillschweigend vorausgesetzt, dass  $b^2-4ac>0$ .
```

Weitere Hinweise zu Dokumentationsstrings sind in [PEP 257](#) zu finden.

## 5.3 Lokale und globale Variable

Dem letzten Beispiel im Abschnitt *Funktionsdefinitionen* kann man entnehmen, dass die Benennung der zurückzugebenden Variablen in der Funktionsdefinition ( $ax$ ,  $ay$ ) und in der Anweisung, die den Funktionsaufruf enthält, ( $vx$ ,  $vy$ ) nicht identisch sein muss. Gleiches gilt auch für die Argumente, die der Funktion übergeben werden. Daraus folgt unter anderem, dass die Reihenfolge der Argumente in der Funktionsdefinition und im Funktionsaufruf übereinstimmen müssen. Hier hilft es auch nicht, die gleichen Variablennamen zu verwenden, wie aus den folgenden Überlegungen deutlich wird. Will man die Argumente in einer willkürlichen Reihenfolge verwenden, so muss man Schlüsselworte verwenden, wie im Abschnitt *Schlüsselworte und Defaultwerte* genauer erklärt wird.

Wie verhält es sich nun mit Variablennamen, die sowohl in der Funktionsdefinition als auch im Hauptprogramm vorkommen? Betrachten wir dazu ein Beispiel

```
1 def f(x):
2     x = x+1
3     print("lokale Variable: ", locals())
4     print("globale Variable:", globals())
5     return x*y
6
7 x = 5
8 y = 2
9 print("f(3) =", f(3))
10 print("x =", x)
```

das die folgende Ausgabe produziert:

```
lokale Variable: {'x': 4}
globale Variable: {'f': <function f at 0xb7e42ed4>,
                  '__builtins__': <module '__builtin__' (built-in)>,
                  '__file__': 'test.py',
                  '__package__': None,
                  'x': 5,
                  'y': 2,
                  '__name__': '__main__',
                  '__doc__': None}

f(3) = 8
x = 5
```

In Python kann man sich mit Hilfe der eingebauten Funktionen `globals()` und `locals()` die globalen bzw. lokalen Variablen ausgeben lassen, die die Funktion sieht. Die Variable `x` kommt dabei sowohl im lokalen als auch im globalen Kontext vor. Die lokale Variable `x` wird zu Beginn der Abarbeitung der Funktion generiert und gemäß Zeile 9 mit dem Wert 3 belegt. Nach dem Inkrementieren in Zeile 2 ergibt sich der in `locals()` angegebene Wert von 4. Gleichzeitig hat `x` im globalen Kontext den Wert 5.

Wird auf eine Variable zugegriffen, so sucht Python zunächst in den lokalen Variablen, dann in den globalen Variablen und zuletzt in den eingebauten Pythonfunktionen. Für die Variable `x` hat dies in der Funktion zur Folge, dass die lokale Variable gemeint ist. So verändert das Inkrementieren in Zeile 2 auch nicht den Wert der globalen Variable `x`. Nachdem die Variable `y` nicht in den lokalen Variablen vorkommt, greift Python in Zeile 5 auf die globale Variable mit dem Wert 2 aus Zeile 8 zurück. Innerhalb der Funktion ist es nicht möglich, globale Variable

zu verändern, was in den meisten Fällen auch nicht erwünscht sein dürfte. Eine wichtige Ausnahme hiervon sind Listen. Den Grund hierfür werden wir in Kapitel *Listen* besser verstehen.

Greift man beim Programmieren in einer Funktion auf eine globale Variable zurück, so sollte man sich immer überlegen, ob es nicht günstiger ist, der Funktion diese Variable als Argument zu übergeben. Dies führt zwar zu aufwendigeren Funktionsaufrufen, hilft aber, bei Änderungen im aufrufenden Programmcode Fehler in der Funktion zu vermeiden. Solche Fehler können leicht dadurch entstehen, dass man die Verwendung der globalen Variable in der Funktion übersieht. Will man überlange Argumentlisten vermeiden, so können auch Techniken von Nutzen sein, die wir im Kapitel *Objektorientiertes Programmieren* besprechen werden.

## 5.4 Rekursive Funktionen

Um die Verwendung von rekursiven Funktionen zu illustrieren, betrachten wir die Berechnung der Fakultät  $n!$ , die man durch Auswertung des Produkts  $1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n$  erhält. Der folgende Code stellt eine direkte Umsetzung dieser Vorschrift dar:

```
1 def fakultaet(n):
2     produkt = 1
3     for m in range(n):
4         produkt = produkt*(m+1)
5     return produkt
6
7 for n in range(1, 10):
8     print(n, fakultaet(n))
```

Alternativ lässt sich die Fakultät auch rekursiv definieren. Mit Hilfe der Beziehung  $n! = (n-1)! \cdot n$  lässt sich die Fakultät von  $n$  auf die Fakultät von  $n-1$  zurückführen. Wenn man nun noch verwendet, dass  $0! = 1$  ist, kann man die Fakultät rekursiv bestimmen, wie der folgende Code zeigt:

```
1 def fakultaet_rekursiv(n):
2     if n>0:
3         return n*fakultaet_rekursiv(n-1)
4     elif n==0:
5         return 1
6     else:
7         raise ValueError("Argument darf nicht negativ sein")
8
9 for n in range(1, 10):
10    print(n, fakultaet_rekursiv(n))
```

Hier wird nun in der Funktion wiederum die Funktion selbst aufgerufen, was nicht in jeder Programmiersprache erlaubt ist. Der Vorteil einer rekursiven Funktion ist häufig die unmittelbare Umsetzung einer Berechnungsvorschrift. Allerdings muss darauf geachtet werden, dass die Aufrufserie irgendwann beendet wird. Ist dies nicht der Fall, könnte das Programm nicht zu einem Ende kommen und, da Funktionsaufrufe auch Speicherplatz erfordern, immer mehr Speicher belegen. Über kurz oder lang wird das Betriebssystem dann das Programm beenden, wenn nicht gar der Computer abstürzt und neu gestartet werden muss. Daher ist in der Praxis die Zahl der rekursiven Aufrufe begrenzt. In Python lässt sich diese Zahl in folgender Weise bestimmen:

```
>>> import sys
>>> sys.getrecursionlimit()
1000
```

Mit `setrecursionlimit(n)` lässt sich mit einer natürlichen Zahl  $n$  als Argument notfalls die Rekursionstiefe verändern, was jedoch sicher kein Ausweg ist, wenn die rekursive Programmierung fehlerhaft durchgeführt wurde.

## 5.5 Funktionen als Argumente von Funktionen

In Python können, im Gegensatz zu vielen anderen Programmiersprachen, Funktionen Variablen zugewiesen oder als Argumente von anderen Funktionen verwendet werden. Folgendes Beispiel illustriert dies:

```

1 >>> from math import sin, cos
2 >>> funktion = sin
3 >>> print(funktion(1), sin(1))
4 0.841470984808 0.841470984808
5 >>> for f in [sin, cos]:
6     ...     print("{}: {:.6f}".format(f.__name__, f(1)))
7     ...
8 sin: 0.841471
9 cos: 0.540302

```

In Zeile 2 wird die Funktion `sin` der Variable `funktion`, die natürlich im Rahmen der Regeln für die Benennung von Variablen auch anders heißen könnte, zugewiesen. Statt der Funktion `sin` könnte hier auch eine selbst definierte Funktion stehen. Zu beachten ist, dass der Funktion auf der rechten Seite kein Argument mitgegeben wird. Aus den Zeilen 3 und 4 wird deutlich, dass nach der Zuweisung in Zeile 2 auch die Variable `funktion` verwendet werden kann, um den Sinus zu berechnen. Eine mögliche Anwendung ist in den Zeilen 5 und 6 angedeutet, wo eine Schleife über zwei Funktionen ausgeführt wird. Zeile 6 zeigt zudem, dass das Attribut `__name__` des Funktionsobjekts den entsprechenden Funktionsnamen angibt.

Interessant ist auch die Möglichkeit, Funktionen als Argumente zu übergeben. Wenn zum Beispiel die Ableitung einer mathematischen Funktion numerisch bestimmt werden soll, kann man für jede spezielle mathematische Funktion, die abgeleitet werden soll, eine entsprechende Ableitungsfunktion definieren. Viel besser ist es natürlich, dies nur einmal für eine beliebige mathematische Funktion zu tun. Eine mögliche Lösung könnte folgendermaßen aussehen:

```

1 from math import sin
2
3 def ableitung(f, x):
4     h = 1e-7
5     df = (f(x+h)-f(x-h))/(2*h)
6     return df
7
8 print(ableitung(sin, 0))

```

Natürlich ist dieser Code verbesserungsbedürftig. Er sollte unter anderem ordentlich dokumentiert werden, und man müsste sich Gedanken über die Überprüfung der Konvergenz des Grenzwertes  $h \rightarrow 0$  machen. Unabhängig davon kann der Ableitungscode in den Zeilen 3-6 aber für beliebige abzuleitende mathematische Funktionen aufgerufen werden. Jede Verbesserung dieses Codes würde damit nicht nur der Ableitung einer speziellen mathematischen Funktion zugute kommen, sondern potentiell jeder numerischen Ableitung einer mathematischen Funktion mit Hilfe dieses Codes.

## 5.6 Lambda-Funktionen

In Abschnitt *Funktionsdefinitionen* hatten wir ein Beispiel gesehen, in dem die Berechnung der Funktion vollständig in der `return`-Anweisung untergebracht war. Solche Funktionen kommen relativ häufig vor und können mit Hilfe so genannter Lambda-Funktionen in einer einzigen Zeile deklariert werden:

```

1 >>> quadrat = lambda x: x**2
2 >>> quadrat(3)
3 9

```

Der Variablenname auf der linken Seite in Zeile 1 fungiert im Weiteren als Funktionsname so wie wir es schon im Abschnitt *Funktionen als Argumente von Funktionen* kennengelernt hatten. `lambda` ist ein Schlüsselwort<sup>3</sup>, das die Definition einer Lambda-Funktion andeutet. Darauf folgt ein Funktionsargument, hier `x`, oder auch mehrere, durch Komma getrennte Argumente und schließlich nach dem Doppelpunkt die Funktionsdefinition.

Solche Funktionsdefinitionen sind sehr praktisch, wenn man eine Funktion als Argument in einem Funktionsaufruf übergeben möchte, aber auf die explizite Definition der Funktion verzichten will. Die im Abschnitt *Funktionen als Argumente von Funktionen* definierte Ableitungsfunktion könnte beispielsweise folgendermaßen aufgerufen werden:

<sup>3</sup> Siehe hierzu Abschnitt *Variablen und Zuweisungen*.

```

1 def ableitung(f, x):
2     h = 1e-7
3     df = (f(x+h)-f(x-h))/(2*h)
4     return df
5
6 print(ableitung(lambda x: x**3, 1))

```

Von Rundungsfehlern abgesehen liefert dies das korrekte Ergebnis 3. In diesem Fall musste man sich nicht einmal Gedanken darüber machen, wie man die Funktion benennen will.

## 5.7 Schlüsselworte und Defaultwerte

Bis jetzt sind wir davon ausgegangen, dass die Zahl der Argumente in der Funktionsdefinition und im Funktionsaufruf übereinstimmen und die Argumente auch die gleiche Reihenfolge haben. Dies ist nicht immer praktisch. Man kann sich vorstellen, dass die Zahl der Argumente nicht im Vorhinein feststeht. Es sind auch optionale Argumente denkbar, die, sofern sie nicht explizit im Aufruf angegeben werden, auf einen bestimmten Wert, den so genannten Defaultwert, gesetzt werden. Schließlich ist es vor allem bei längeren Argumentlisten praktisch, wenn die Reihenfolge der Argumente nicht zwingend vorgeschrieben ist.

Wir wollen uns zunächst die Verwendung von Schlüsselworten und Defaultwerten ansehen und ziehen hierzu wiederum die bereits bekannte Funktion zur numerischen Auswertung von Ableitungen mathematischer Funktionen heran:

```

def ableitung(f, x):
    h = 1e-7
    df = (f(x+h)-f(x-h))/(2*h)
    return df

```

Zu Beginn des Abschnitts *Lokale und globale Variable* hatten wir erklärt, dass die Variablen im Funktionsaufruf nicht genauso benannt werden müssen wie in der Funktionsdefinition, und es somit auf die Reihenfolge der Argumente ankommt. Kennt man die in der Funktionsdefinition verwendeten Variablennamen, so kann man diese auch als Schlüsselworte verwenden. In diesem Fall kommt es dann nicht mehr auf die Reihenfolge an. Statt

```
print(ableitung(lambda x: x**3, 1))
```

könnte man auch den Aufruf

```
print(ableitung(f=lambda x: x**3, x=1))
```

oder

```
print(ableitung(x=1, f=lambda x: x**3))
```

verwenden, wobei das Schlüsselwort `x` nichts mit dem in der Definition der Lambda-Funktion auftretenden `x` zu tun hat. Dass es wichtig ist, verschiedene Bezeichner zu unterscheiden, macht auch die folgende Erweiterung dieses Aufrufs deutlich.

```

x = 1
print(ableitung(x=x, f=lambda x: x**3))

```

Hier ist im ersten Argument zwischen dem Schlüsselwort `x`, also dem ersten `x` im ersten Argument, und der Variable `x`, dem zweiten `x` im ersten Argument, zu unterscheiden. Nachdem der Variable `x` zuvor der Wert 1 zugewiesen wurde, wird also im ersten Argument der Funktion `ableitung()` der Wert 1 für die lokale Variable `x` übergeben. Die Variable `x` in der Lambdafunktion wiederum hat weder mit dem einen noch mit dem anderen `x` im ersten Argument etwas zu tun.

Hat man eine längere Argumentliste und übergibt man manche Argumente ohne und andere Argumente mit Schlüsselwort, so müssen die Argumente ohne Schlüsselwort zwingend vor den Argumenten mit Schlüsselwort stehen. Die Argumente ohne Schlüsselwort werden dann nach ihrer Reihenfolge beginnend mit dem ersten Argument zugeordnet, die anderen Argumente werden gemäß dem verwendeten Schlüsselwort übergeben.

Wenn wir dem Benutzer in unserer Ableitungsfunktion die Möglichkeit geben wollen, die Schrittweite bei Bedarf anzupassen, dies aber nicht unbedingt verlangen wollen, können wir einen Defaultwert vorgeben. Dies könnte folgendermaßen aussehen:

```
def ableitung(f, x, h=1e-7):  
    df = (f(x+h)-f(x-h))/(2*h)  
    return df
```

Der Aufruf

```
print(ableitung(lambda x: x**3, 1, 1e-3))
```

würde die Ableitung der dritten Potenz an der Stelle 1 auswerten, wobei die Schrittweite gleich 0.001 gewählt ist. Mit dem Aufruf

```
print(ableitung(lambda x: x**3, 1))
```

wird dagegen der Defaultwert für  $h$  aus der Funktionsdefinition, also  $10^{-7}$ , verwendet. Nach dem was wir weiter oben gesagt haben, würden die Argumente des Aufrufs

```
print(ableitung(lambda x: x**3, h=1e-3, x=1))
```

folgendermaßen interpretiert werden: Das erste Argument trägt keinen Variablennamen und wird demnach der ersten Variablen, also  $f$ , in der Funktionsdefinition zugeordnet. Anschließend kommen mit Schlüsselworten versehene Argumente, bei denen es jetzt nicht mehr auf die Reihenfolge ankommt. Tatsächlich ist die Reihenfolge im angegebenen Aufruf gegenüber der Funktionsdefinition vertauscht. Nach dem ersten Argument mit Schlüsselwort müssen alle folgenden Argumente mit einem Schlüsselwort versehen sein, was hier in der Tat der Fall ist. Andernfalls liegt, wie im folgenden Beispiel, ein Syntaxfehler vor.

```
print(ableitung(lambda x: x**3, h=1e-3, 1))
```

```
File "<stdin>", line 1  
SyntaxError: non-keyword arg after keyword arg
```

Neben den besprochenen Möglichkeiten kann man in Python auch noch eine nicht durch die Funktionsdefinition festgelegte Zahl von Variablen übergeben, wobei auch Schlüsselworte vorkommen können, die nicht in der Variablenliste der Funktionsdefinition zu finden sind. Auf diese Art der Argumentübergabe werden wir im Kapitel *Dictionaries* zurückkommen, da wir erst dort den hierzu benötigten Datentyp kennenlernen werden.

---

## Zusammengesetzte Datentypen

---

### 6.1 Listen

Bei numerischen Anwendungen in Physik und Materialwissenschaften will man neben den einfachen Datentypen des Kapitels *Einfache Datentypen, Variablen und Zuweisungen* unter anderem auch Vektoren und Matrizen verwenden. Echte Vektoren und Matrizen mit zugehörigen Funktionen wie beispielsweise dem Skalarprodukt werden von Python nicht direkt zur Verfügung gestellt. Hierzu greift man auf das Numpy-Modul zurück, das im Abschnitt *Arrays und Anwendungen* besprochen wird. Python stellt jedoch sehr wohl Datentypen zur Verfügung, die als Ansammlung von Zahlen verwendet werden können. Der erste Typ dieser Art, den wir hier besprechen wollen, ist die Liste. Wir haben sie bereits im Abschnitt *For-Schleife* kennengelernt. Dort hatten wir mit Hilfe von `range()` und `list()` Listen erzeugt, auch wenn wir uns dessen dort nicht wirklich bewusst waren. Dementsprechend lässt sich eine, zugegebenermaßen sehr spezielle Liste folgendermaßen erzeugen:

```
>>> liste = list(range(20))
>>> print(liste)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> type(liste)
<class 'list'>
```

Wie wir aus dem Abschnitt *For-Schleife* wissen, erzeugt die `range()`-Funktion bei Angabe eines einzigen Arguments eine Sequenz von Zahlen, die mit Null beginnt und mit dem um Eins verminderten Argument endet. Aufeinanderfolgende Listenelemente unterscheiden sich in diesem Fall jeweils um Eins.

Häufig ist es notwendig, die Anzahl der Elemente einer Liste zu kennen. Diese Information erhält man mit Hilfe der Funktion `len()`:

```
>>> liste = list(range(1, 17, 3))
>>> print(liste)
[1, 4, 7, 10, 13, 16]
>>> len(liste)
6
```

Es ist möglich, auf einzelne Elemente der Liste zuzugreifen und diese auch zu verändern:

```
>>> liste = [1, 17, 3]
>>> liste[1]
17
>>> liste[1] = 2
>>> liste
[1, 2, 3]
```

Dabei ist immer zu bedenken, dass der Index, mit dem die Elemente durchnummeriert werden, mit Null beginnt. Das zweite Element ist also in diesem Fall `liste[1]`. Diese Zählweise wird auch in einigen anderen Programmiersprachen verwendet, nicht aber zum Beispiel in FORTRAN.

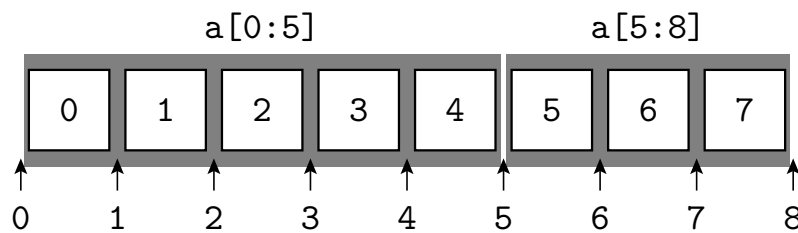
Neben einzelnen Elementen kann man auch Unterlisten, so genannte »slices«, erzeugen.

```
>>> a = [2, 3, 5, 7, 11, 13, 17, 19, 23]
>>> a[1:5]
[3, 5, 7, 11]
```

Vor dem Doppelpunkt steht der Index des ersten Elements des Ausschnitts der Liste. Dies ist hier also `a[1]`. Der Index des letzten Elements, hier `a[4]`, ist durch die um Eins verminderte Zahl nach dem Doppelpunkt gegeben. Dieses Verhalten entspricht genau dem, was wir von der `range()`-Funktion her kennen. Eine Unterliste von `a`, die direkt an `a[1:5]` anschließt, hat als ersten Index die 5, also

```
>>> a[5:8]
[13, 17, 19]
```

Die Funktionsweise der Indizes lässt sich anschaulich verstehen, wenn man sie nicht als Index eines Listeneintrags ansieht, sondern als Markierung »zwischen« den Listeneinträgen, wie es die folgende Abbildung zeigt:



? Was ergibt `a[2:2]`?

Wird einer der beiden Indizes nicht angegeben, so wird er durch den Index ersetzt, der auf den Beginn bzw. das Ende der Liste zeigt.

```
>>> a[:5]
[2, 3, 5, 7, 11]
>>> a[5:]
[13, 17, 19, 23]
>>> a[:]
[2, 3, 5, 7, 11, 13, 17, 19, 23]
```

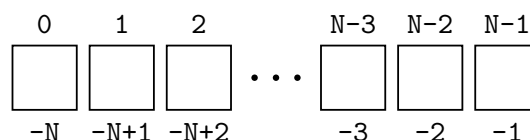
In der letzten Anweisung wurden beide Indizes weggelassen, so dass man die gesamte Liste erhält, was zum Beispiel auch folgendermaßen möglich wäre:

```
>>> a
[2, 3, 5, 7, 11, 13, 17, 19, 23]
>>> a[0:len(a)]
[2, 3, 5, 7, 11, 13, 17, 19, 23]
```

Was passiert nun, wenn man den ersten Index außerhalb des Bereichs zwischen 0 und  $N-1$  oder den zweiten Index außerhalb des Bereichs 1 und  $N$  wählt, wobei  $N$  die Listenlänge sei? In manchen Programmiersprachen kann es passieren, dass man auf ein zufällig im Speicher benachbart liegendes Objekt zugreift. In den meisten Fällen wird dies zu einem unerwünschten Ergebnis führen. Python dagegen weist auf den illegalen Zugriff hin:

```
>>> a[13]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Allerdings sind negative Indizes bis zu  $-N$  erlaubt. Dabei beginnt die Zählung mit  $-1$  vom letzten Listenelement an rückwärts, so dass das erste Element auch mit dem Index  $-N$  angesprochen werden kann. Das folgende Bild stellt die Zuordnung der Indizes dar.





Die letzten beiden Elemente einer Liste erhält man demnach mit Hilfe von

```
>>> a = [2, 3, 5, 7, 11, 13, 17, 19, 23]
>>> a[-2:]
[19, 23]
```

Schließlich gibt es auch noch die Möglichkeit, die Schrittweite bei der Erzeugung der Unterliste festzulegen, wie die folgenden Beispiele zeigen:

```
>>> a[0:6:2]
[2, 5, 11]
>>> a[:3]
[2, 7, 17]
>>> a[::-1]
[23, 19, 17, 13, 11, 7, 5, 3, 2]
```

In der ersten Anweisung wird jedes zweite Listenelement aus dem Indexbereich 0 bis 5 ausgewählt, also die Listenelemente `a[0]`, `a[2]` und `a[4]`. In der zweiten Anweisung wird jedes dritte Listenelement aus der gesamten Liste ausgewählt. Letzteres ergibt sich daraus, dass die ersten beiden Indizes nicht explizit angegeben sind. Eine Schrittweite von `-1` führt schließlich zu einer Umkehrung der Reihenfolge der Listenelemente. Eine Alternative hierzu bietet die `reverse()`-Methode an, die wir am Ende dieses Unterkapitels besprechen werden.

In Python ist zu beachten, dass sich das Kopieren von Listen nicht so verhält, wie man es vielleicht erwarten würde.

```
1 >>> a = [2, 17, 9]
2 >>> id(a)
3 3070057740
4 >>> b = a
5 >>> id(b)
6 3070057740
7 >>> a[0] = 111
8 >>> a
9 [111, 17, 9]
10 >>> b
11 [111, 17, 9]
```

Die Zuweisung in Zeile 4 führt nicht zu einem neuen Listenobjekt, das unabhängig verändert werden kann. Vielmehr zeigen die Zeilen 2 und 3 bzw. 5 und 6, dass es sich bei `a` und `b` um dasselbe Objekt handelt. Ändert man ein Element der Liste `a`, so wird, wie die Zeilen 8-11 zeigen, auch die Liste `b` entsprechend geändert. Letztlich hat man durch die Anweisung in Zeile 4 lediglich einen alternativen Namen definiert, über den die Liste angesprochen werden kann. Eine unabhängige Kopie einer Liste erhält man unter anderem auf folgende Weise:

```
1 >>> a = [2, 17, 9]
2 >>> b = a[:]
3 >>> id(a)
4 3070057740
5 >>> id(b)
6 3070058220
7 >>> a[0] = 111
8 >>> a
9 [111, 17, 9]
10 >>> b
11 [2, 17, 9]
```

In Zeile 1 werden die Listeneinträge von `a` in eine neue Liste `b` kopiert. Wie die Zeilen 2-5 zeigen, wird dabei tatsächlich ein neues Listenobjekt erzeugt. Die beiden Listen `a` und `b` lassen sich damit unabhängig voneinander verändern, wie die Zeilen 6-10 zeigen.

In den bisherigen Beispielen waren alle Elemente der Liste vom gleichen Typ, hier speziell vom Typ Integer. In manchen Programmiersprachen ist dies nicht anders möglich oder nur unter Verwendung anderer Sprachelemente. In Python können Listenelemente beliebige Objekte sein, die auch nicht unbedingt vom gleichen Typ sein müssen. Im Abschnitt *Funktionen als Argumente von Funktionen* hatten wir beispielsweise schon eine Liste kennengelernt,

deren Elemente Funktionsnamen waren. Dadurch, dass nicht alle Elemente von gleichem Typ sein müssen, ist der folgende Code im Prinzip möglich:

```
1 >>> from math import exp
2 >>> aufgabe = [exp, 5.2]
3 >>> aufgabe[0](aufgabe[1])
4 181.27224187515122
5 >>> exp(5.2)
6 181.27224187515122
```


In Zeile 2 wird eine Liste definiert, die einen Funktionsnamen und einen Float-Wert enthält. Anschließend werden die Listenelemente in Zeile 3 verwendet, um einen Funktionswert zu berechnen, der in den Zeilen 5 und 6 überprüft wird. Natürlich ist es auch möglich, Listen als Listenelemente zu verwenden.

```
1 >>> a = [[1, 3], [2, 4]]
2 >>> a[0]
3 [1, 3]
4 >>> a[1]
5 [2, 4]
6 >>> a[0][1]
7 3
```

In den Zeilen 2 und 4 werden Elemente der Liste `a` ausgewählt, wie wir es schon aus den obigen Beispielen kennen. Das Ergebnis ist hier jeweils eine Liste, aus der wiederum ein Element ausgewählt werden kann. In Zeile 6 bedeutet `a[0][1]` also, dass das Listenelement `a[0]` betrachtet werden soll, eine Liste von der wiederum das Element mit dem Index 1 ausgewählt wird. Auch wenn die in dem Beispiel definierte Liste `a` sehr an eine Matrix erinnert, ist zu bedenken, dass Python für Listen keine spezifischen Matrixoperationen zur Verfügung stellt. Entweder definiert man sich solche Operationen selbst oder man greift auf die Möglichkeiten zurück, die das [Numpy-Modul](#) bietet, das im Kapitel *Numerische Programmibibliotheken am Beispiel von NumPy/SciPy* genauer besprochen wird.

Bis jetzt haben wir bei der Definition einer Liste immer gleich alle Elemente festgelegt. Wenn wir nachträglich Elemente hinzufügen wollen, ergibt sich das Problem, dass wir nicht auf nicht existierende Elemente zugreifen dürfen, da sonst ein `IndexError` geworfen wird. In manchen Programmiersprachen ist bei der Definition einer Liste die Größe fest vorzugeben, wobei jedoch nicht unbedingt jedes Element spezifiziert werden muss. In Python dagegen lässt sich die Länge der Liste durch Hinzufügen von Elementen verändern. In dem folgenden Beispiel definieren wir zunächst in Zeile 1 eine leere Liste, an die wir anschließend Elemente anhängen:

```
1 >>> a = []
2 >>> for n in range(5):
3 ...     a.append(n**2)
4 ...
5 >>> a
6 [0, 1, 4, 9, 16]
```

 `for`-Schleifen werden in Python relativ langsam abgearbeitet. Die Funktionalität des angegebenen Codes lässt sich wesentlich effizienter als so genannte »list comprehension« realisieren <sup>1</sup>, die für unser Beispiel die folgende Form hat:

```
>>> a = [n**2 for n in range(5)]
>>> a
[0, 1, 4, 9, 16]
```

Mit Hilfe der `append()`-Methode lassen sich also Elemente zu einer Liste hinzufügen. Sollen zunächst alle Listenelemente gleich sein, so kann man auch eine Multiplikation verwenden.

```
>>> a = [0]*10
>>> a
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Mit der `append()`-Methode verwandt ist die `extend()`-Methode, mit der man eine Liste an eine andere Liste anhängen kann.

<sup>1</sup> Eine ausführlichere Beschreibung von »list comprehensions« findet sich in der Python-Dokumentation im [Kapitel über Datenstrukturen](#).

```
1 >>> a = [1, 2, 3]
2 >>> b = ["eins", "zwei", "drei"]
3 >>> b.extend(a)
4 >>> b
5 ['eins', 'zwei', 'drei', 1, 2, 3]
```

Hier wird in Zeile 3 die Liste `b` um die Elemente der Liste `a` erweitert. Alternativ kann man zwei Listen mit Hilfe des `+`-Operators aneinanderhängen wie folgendes Beispiel zeigt.

```
1 >>> a = [1, 2, 3]
2 >>> b = ["eins", "zwei", "drei"]
3 >>> b+a
4 ['eins', 'zwei', 'drei', 1, 2, 3]
5 >>> b+3
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8   TypeError: can only concatenate list (not "int") to list
9 >>> b+[3]
10 ['eins', 'zwei', 'drei', 3]
11 Traceback (most recent call last):
12   File "<stdin>", line 1, in <module>
13   TypeError: can only concatenate list (not "int") to list
```

Dabei ist allerdings darauf zu achten, dass beide Objekte Listen sein müssen wie aus den Zeilen 5-10 zu sehen ist.

Wir wollen im Folgenden nicht auf alle Möglichkeiten eingehen, mit Listen zu arbeiten<sup>2</sup>, sondern nur noch einige ausgewählte Punkte ansprechen. Gelegentlich möchte man in einer Liste nach Elementen suchen. Die `index()`-Methode gibt den Index des ersten Auftretens des gesuchten Objekts an.

```
>>> a = [1, 3, 2, -2, 3]
>>> a.index(3)
1
```

Will man nach einem weiteren Auftreten des Objekts suchen, so betrachtet man die auf das erste Auftreten folgende Unterliste. Das Ergebnis bezeichnet dann jedoch den betreffenden Index in der Unterliste, nicht in der ursprünglichen Liste.

```
>>> a[2:].index(3)
2
```

Ist das gesuchte Objekt nicht in der Liste vorhanden, so wird ein `ValueError` geworfen.

```
>>> a.index(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.index(x): x not in list
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.index(x): x not in list
```

Dieser Fall sollte mit einer `try ... except`-Konstruktion abgefangen werden. Der entsprechende Code könnte folgendermaßen aussehen:

```
try:
    print(a.index(4))
except ValueError:
    print("Objekt nicht gefunden")
```

Manchmal genügt es auch, danach zu fragen, ob das Objekt in der Liste vorhanden ist. Hier bekommt man als Antwort entweder `True` oder `False`. Den Index des gesuchten Listenelements erfährt man auf diese Weise allerdings nicht.

---

<sup>2</sup> Eine ausführliche Dokumentation der auf Listen anwendbaren Methoden findet sich in der Python-Dokumentation im [Kapitel über Datenstrukturen](#).

```
>>> 3 in a
True
>>> 4 in a
False
```

Zwei Methoden geben keine neue Liste zurück, sondern verändern die aktuelle Liste. So kann man die Reihenfolge einer Liste umkehren

```
>>> a = [1, 2, 3]
>>> a.reverse()
>>> a
[3, 2, 1]
```

oder die Elemente einer Liste sortieren

```
a = [7, 2, -5, 3]
>>> a.sort()
>>> a
[-5, 2, 3, 7]
```

Es sei nur kurz erwähnt, dass im Prinzip auch eine beliebige Sortierfunktion vorgegeben werden kann, was insbesondere bei nichtnumerischen Listenelementen interessant sein kann.

Zum Abschluss der Diskussion von Listen sei noch einmal kurz an die Verwendung in `for`-Schleifen erinnert. Das folgende Beispiel zeigt eine Anwendung, bei der jedes Listenelement eine zwei Zahlen umfassende Liste ist.

```
1 >>> a = [[5, 3], [2, 4]]
2 >>> for x, y in a:
3     ...     print("{} - {} = {}".format(x, y, x-y))
4     ...
5 5 - 3 = 2
6 2 - 4 = -2
```

Entsprechend sind in Zeile 2 der `for`-Schleife zwei Variable anzugeben, die entsprechend ihrer Reihenfolge den beiden Zahlen in den Listen zugeordnet werden.

## 6.2 Tupel

Tupel sind ähnlich wie Listen Sequenzen von Objekten beliebigen Typs. Der wesentliche Unterschied zu Listen besteht darin, dass Tupel unveränderlich sind. Man kann also wie bei Listen auf einzelne Elemente oder Untersequenzen zugreifen. Es ist jedoch nicht möglich, die Sequenz durch eine Zuweisung zu verändern.

```
1 >>> a = (1, 7, 19)
2 >>> a[-1]
3 19
4 >>> a[0:1]
5 (1,)
```

Im Gegensatz zu Listen, die durch eckige Klammern gekennzeichnet werden, sind bei Tupeln runde Klammern, wie in Zeile 1 zu sehen ist, zu verwenden. Zugriffe auf einzelne Elemente, wie in Zeile 2 auf das letzte Element der Sequenz, sowie auf Untersequenzen sind problemlos möglich. Enthält die Sequenz wie in Zeile 5 nur ein Element, so ist ein Komma vor der schließenden Klammer notwendig. `(1)` würde nicht als Tupel, sondern als geklammerte Eins interpretiert werden.

```
>>> x = (1)
>>> type(x)
<class 'int'>
>>> x = (1,)
>>> type(x)
<class 'tuple'>
```

Die folgenden Zeilen zeigen schließlich, dass der Versuch, ein Element des Tupels zu verändern, mit einem `TypeError` beendet wird. Tupel sind also, wie oben behauptet, tatsächlich unveränderlich.

```
>>> a[1] = 23
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

## 6.3 Zeichenketten

Eine wichtige Art von ebenfalls unveränderlichen Sequenzen sind Zeichenketten oder »Strings«. Ihnen sind wir in verschiedenen Codebeispielen schon gelegentlich begegnet. Solange die Zeichenkette in eine Zeile passt, wird sie durch Hochkommas (') oder Anführungszeichen (") begrenzt.

```
>>> s = "Hallo"
>>> t = 'Hallo'
>>> s==t
True
```

Diese beiden Strings sind also identisch. Die Möglichkeit Strings auf zwei Arten zu begrenzen, erleichtert es, Hochkommas oder Anführungszeichen in einem String unterzubringen. Wenn der Begrenzer jedoch auch innerhalb des Strings auftreten soll, muss diesem mit einem vorgestellten Backslash (\) seine Sonderbedeutung genommen werden.

```
s = "God said, \'Let Newton be!\' and all was light" (Alexander Pope)
>>> print(s)
"God said, 'Let Newton be!' and all was light" (Alexander Pope)
```

Der Backslash kann auch dazu benutzt werden, dem nachfolgenden Zeichen eine besondere Bedeutung als Steuerzeichen zu geben. Besonders wichtig ist `\n`, das einen Zeilenumbruch zur Folge hat.

```
>>> s = "Eine Zeile\nund noch eine Zeile"
>>> print(s)
Eine Zeile
und noch eine Zeile
```

Wie wir im Kapitel *Dokumentation von Funktionen* bereits gesehen haben, lassen sich über mehrere Zeilen gehende Strings auch durch Begrenzung mit jeweils drei Hochkommas oder drei Anführungszeichen angeben. Dabei sind die Fortsetzungszeilen nicht speziell zu markieren.

Gelegentlich von Nutzen sind `\t` für einen horizontalen Tabulator und `\f` für einen Seitenvorschub. Wenn man einen Backslash als normales Zeichen in einem String benötigt, so kann er entweder mit einem zusätzlichen Backslash versehen werden (`\\`) oder der ganze String durch Voranstellen eines `r` oder eines `R` als »raw string« gekennzeichnet werden.

```
>>> s = r"Eine Zeile\nund noch eine Zeile"
>>> s
'Eine Zeile\\nund noch eine Zeile'
```

Aus der letzten Zeile ersieht man, dass »raw strings« nur für die Eingabe relevant sind. Für die interne Darstellung verwendet Python dann den doppelten Backslash.

In Python 3 werden grundsätzlich alle Zeichen durch ihren Unicode-Codepoint identifiziert. Damit ist eine größtmögliche Anzahl von Zeichen für eine konsistente Verarbeitung in Strings verfügbar. Im Anhang *Unicode* werden einige Informationen zum Unicode-Standard gegeben. Sollen Strings nicht nur innerhalb eines bestimmten Python-Programms verarbeitet werden, sondern zum Beispiel von Dateien gelesen oder in solche geschrieben werden, so müssen die Zeichen kodiert werden. Ein wichtiges Beispiel hierfür ist die im Unicode-Standard definierte UTF-8-Kodierung. Solchermaßen kodierte Zeichen werden in Python 3 in Objekten vom `bytes`-Typ

abgespeichert. Eine Variante hiervon ist der `bytearray`-Typ, der es, ähnlich wie bei dem im Abschnitt *Listen* besprochenen `list`-Typ, erlaubt, einzelne Bytes zu modifizieren.<sup>3</sup> Der folgende Code soll diese Aspekte verdeutlichen.

```

1  >>> s = "Prüfung"
2  >>> len(s)
3  7
4  >>> s[0], s[2]
5  ('P', 'ü')
6  >>> x = bytes(s, "utf-8")
7  >>> x
8  b'Pr\xc3\xbcfung'
9  >>> len(x)
10 8
11 >>> x[0]
12 80
13 >>> x[2]
14 195

```

Die Zahl der Zeichen der in Zeile 1 definierten Zeichenkette wird in den Zeilen 2 und 3 trotz des Umlauts richtig bestimmt. Beim Zugriff auf einzelne Zeichen der Zeichenkette bekommt man, wie in den Zeilen 4 und 5 zu sehen ist, die entsprechenden Zeichen. Man kann nun die Zeichenkette in einer Binärdarstellung kodieren. Zeile 6 zeigt dies für die Umwandlung in eine Bytefolge mit Hilfe der UTF-8-Kodierung. Sofern das entsprechende Byte im Rahmen der auf sieben Bit beruhenden ASCII-Kodierung interpretiert werden kann, wird diese Darstellung in Zeile 8 verwendet. Der Umlaut ü wird im Rahmen der UTF-8-Kodierung jedoch mit Hilfe von zwei Bytes dargestellt, wobei in beiden Fällen das erste Bit auf Eins gesetzt ist. Daraus ergibt sich die hexadezimale Darstellung der Bytes, beispielsweise als `\xc3`. Wie man Zeile 10 entnehmen kann, gibt die Länge der Bytedarstellung nun nicht mehr notwendigerweise die Anzahl der ursprünglichen Zeichen an. Greift man auf einzelne Elemente des `bytes`-Objekts zu, so erhält man den dem jeweiligen Byte entsprechenden Integer. Dies ist in den Zeilen 11-14 zu sehen.

Die folgenden Beispiele zeigen, dass man mit Hilfe des Backslash Unicode-Strings auch über ihren Codepoint (`\u`) oder mit dem Unicodenamen des Zeichens (`\N`) definieren kann.

```

>>> s = "\u03a8"
>>> s
'Ψ'
>>> s = "\N{GREEK SMALL LETTER PI}"
>>> s
'π'

```

Wie wir im vorigen Abschnitt gesehen haben, sind Tupel unveränderlich. Das gilt auch für Strings:

```

>>> s = "abc"
>>> id(s)
3085106464L
>>> s = s+"def"
>>> id(s)
3085106560L

```

Die Identität `id(s)` der hier definierten Strings `s` ist tatsächlich verschieden. Dieses Codebeispiel zeigt auch die Anwendung des Additionsoperators. Zudem ist ein Multiplikationsoperator wie folgt definiert:

```

>>> s = "abc"
>>> s*5
'abccabccabccabc'

```

**+** Die Verwendung des Additionsoperators ist wegen der Notwendigkeit, ein neues Objekt zu erzeugen, nicht sehr effizient. Will man viele Teilstrings aneinanderfügen, so verwendet man besser die `join()`-Methode, bei der sich noch eine Zeichenkette angeben lässt, die zwischen den Elementen der Liste von Strings im Argument der `join()`-Methode eingesetzt wird. So ergibt `"--".join(["1", "2", "3"])` das Resultat `"1--2--3"`.

<sup>3</sup> Die Datentypen `bytes` und `bytearray` wurden in Python 3 neu eingeführt. In Python 2 musste dagegen zwischen Byte-Strings und Unicode-Strings unterschieden werden, was gelegentlich zu Verwirrung führte. Die für Python 3 getroffene Wahl erlaubt eine klare Unterscheidung zwischen den beiden Datentypen.

Will man auf die Einfügung verzichten, so würde das Beispiel stattdessen `"".join(["1", "2", "3"])` lauten.

Bei Strings lassen sich wie in Abschnitt [Listen](#) beschriebenen Teilstrings generieren, und es lässt sich über die darin enthaltenen Zeichen iterieren wie im Folgenden kurz demonstriert wird:

```
>>> s = "Hallo"
>>> s[::-1]
'ollaH'
>>> for ch in s:
...     print(ch)
...
H
a
l
l
o
>>> list(s)
['H', 'a', 'l', 'l', 'o']
```

Wie die letzten beiden Zeilen zeigen, kann ein String in eine Liste umgewandelt werden, wobei die Zeichenkette in die einzelnen Zeichen aufgelöst wird.

Es sei hier nur kurz erwähnt, dass für String-Objekte eine ganze Reihe von Methoden existieren, von denen hier einige beispielhaft erwähnt werden sollen.<sup>4</sup>

```
1 >>> s = "   Hallo   "
2 >>> s.lstrip()
3 'Hallo   '
4 >>> s = "---Hallo-----"
5 >>> s.rstrip("-")
6 '---Hallo'
7 >>> s.lower()
8 '---hallo---'
9 >>> s = "eins, zwei, drei"
10 >>> s.split(",")
11 ['eins', ' zwei', ' drei']
12 >>> "H".isdigit()
13 False
14 >>> "H".isupper()
15 True
```

Zeile 2 zeigt, dass Leerzeichen mit `lstrip()` links und mit `rstrip()` auch rechts entfernt werden können, wobei sich alternativ wie in Zeile 5 andere Zeichen entfernen lassen. Zeile 7 zeigt die Umwandlung in Kleinbuchstaben und in Zeile 10 wird ein String an einem vorgegebenen Zeichen, hier einem Komma, aufgespalten. Es lassen sich auch eine Reihe von Tests an Strings durchführen. Zwei Beispiele sind in den Zeilen 12-15 gezeigt.

## 6.4 Dictionaries

Dictionaries sind assoziative Felder, die Schlüssel Werten zuordnen. Man kann sich Dictionaries wie Wörterbücher oder Telefonbücher vorstellen. Betrachten wir ein Beispiel.

```
1 >>> geburtsjahre = {"Galilei": 1564, "Maxwell": 1831, "Einstein": 1879}
2 >>> geburtsjahre["Einstein"]
3 1879
4 >>> geburtsjahre["Heisenberg"] = 1901
5 >>> geburtsjahre
6 {'Heisenberg': 1901, 'Maxwell': 1831, 'Galilei': 1564, 'Einstein': 1879}
7 >>> geburtsjahre["Newton"]
```

<sup>4</sup> Für eine ausführlichere Beschreibung siehe den Abschnitt über Stringmethoden im Kapitel über eingebaute Datentypen der Python-Dokumentation.

```

8  Traceback (most recent call last):
9    File "<stdin>", line 1, in <module>
10  KeyError: 'Newton'
11  Traceback (most recent call last):
12    File "<stdin>", line 1, in <module>
13  KeyError: 'Newton'

```

Jedes Element des Dictionaries besteht aus einem Schlüssel (»key«), der vor dem Doppelpunkt steht und hier vom Typ String ist, und einem Wert (»value«), der nach dem Doppelpunkt steht und hier vom Typ Integer ist. Als Schlüssel können beliebige unveränderliche Objekte verwendet werden, wozu insbesondere die numerischen Objekte und Strings gehören, aber auch Tupel. Als Werte kommen dagegen auch veränderliche Objekte, insbesondere Listen, in Frage. Im Gegensatz zu Listen, die durch eckige Klammern begrenzt werden, und Tupeln, die durch runde Klammern begrenzt werden, werden die Elemente eines Dictionaries, wie in Zeile 1 zu sehen ist, von geschweiften Klammern umschlossen.

Der zu einem Schlüssel gehörende Wert kann, wie in Zeile 2 gezeigt, abgefragt werden. Umgekehrt kann einem existierenden oder noch nicht existierenden Schlüssel ein Wert zugeordnet werden. Der Schlüssel in Zeile 4 ist noch nicht im Dictionary vorhanden, so dass die Anzahl der Elemente des Dictionaries mit der Ausführung der Anweisung um Eins zunimmt. Wird umgekehrt der Wert zu einem nicht vorhandenen Schlüssel abgefragt, kommt es zu einem `KeyError`. Wie der Vergleich der Zeilen 1 und 6 zeigt, kann sich die Reihenfolge der Elemente in einem Dictionary ändern.

**+** Sollte man tatsächlich ein Dictionary mit fester Ordnung benötigen, so steht in Python 3 auch der Objekttyp `OrderedDict` zur Verfügung, der sich die Reihenfolge merkt, in der dem Dictionary Schlüssel zugeordnet wurden.

Wie oben schon angedeutet, können als Schlüssel auch Integer verwendet werden. Worin bestehen die Vorteile gegenüber einer Liste, die ja auch mit Integer adressiert wird? Stellen wir uns vor, dass wir eine Liste mit den Quadraten der ersten fünf Primzahlen erstellen wollen. Eine mögliche Lösung wäre die folgende Liste.

```
prim2 = [0, 0, 4, 9, 0, 25, 0, 49, 0, 0, 0, 121]
```

Ist der Index eine Primzahl, so ist die entsprechende Quadratzahl eingetragen. Alle anderen Zahlen werden gemäß der Aufgabenstellung nicht benötigt. Es ist jedoch klar, dass mit diesem Zugang unter Umständen sehr viel Speicherplatz verschwendet wird. Eine alternative Lösung wäre die Liste:

```
prim2 = [4, 9, 25, 49, 121]
```

Allerdings hat man jetzt keine einfache Zuordnung zwischen Primzahl und Listenindex. Wo steht beispielsweise das Quadrat der Primzahl 7? Man könnte nun für jeden Listeneintrag ein Tupel aus der Primzahl und ihrem Quadrat vorsehen, was aber die Suche aufwendiger macht. Alternativ könnte man eine zweite Liste anlegen, die zwischen Primzahl und Listenindex vermittelt. Am effizientesten ist aber ein Dictionary.

```

>>> prim2 = {2: 4, 3: 9, 5: 25, 7: 49, 11: 121}
>>> prim2[7]
49

```

Über die Einträge eines Dictionaries kann iteriert werden. Zur Illustration verwenden wir wieder unser Geburtsjahr-Dictionary

```

>>> for key in geburtsjahre:
...     print("{} wurde im Jahr {} geboren.".format(key, geburtsjahre[key]))
...
Einstein wurde im Jahr 1879 geboren.
Heisenberg wurde im Jahr 1901 geboren.
Maxwell wurde im Jahr 1831 geboren.
Galilei wurde im Jahr 1564 geboren.

```

Diese Funktionalität könnte man natürlich auch mit Hilfe einer Liste erhalten. Allerdings wäre es viel aufwendiger, gezielt auf bestimmte Einträge zuzugreifen.

Es kann auch auf die Existenz eines Schlüssels abgefragt werden:



```
>>> "Einstein" in geburtsjahre
True
>>> "Newton" in geburtsjahre
False
```

Anstatt zunächst die Existenz eines Schlüssel zu überprüfen, würde man in Python allerdings die Behandlung einer eventuell geworfenen `KeyError`-Ausnahme vorziehen.

```
>>> physiker = "Newton"
>>> try:
...     print("{} wurde im Jahr {} geboren.".format(physiker, geburtsjahre[physiker]))
... except KeyError:
...     print("Das Geburtsjahr von {} ist nicht gespeichert.".format(physiker))
...
Das Geburtsjahr von Newton ist nicht gespeichert.
```

**+** Ein zusammengesetzter Datentyp, der hier nur kurz wegen seiner Verwandtschaft mit Dictionaries erwähnt werden soll, ist das Set, das man sich wie ein Dictionary vorstellen kann, das nur Schlüssel, aber keine zugehörigen Werte enthält. Auf diese Weise lassen sich Mengen definieren, die kein Element mehrfach enthalten, und die Mengenoperationen wie Vereinigungs- und Schnittmenge zulassen. Sets haben beispielsweise dann einen großen Vorteil gegenüber Listen, wenn man überprüfen möchte, ob ein Element in einer gegebenen Menge vorhanden ist.

Im Abschnitt *Formatierung von Ausgaben* hatten wir gesehen, dass man beim Zusammenbauen eines Strings einzelne Objekte mit Hilfe ihres Namens übergeben kann. Manchmal ist es praktisch, hierzu ein Dictionary zu verwenden. Stellt man dem Namen des Dictionaries zwei Sternchen voran, so wird aus dem Dictionary eine Liste von Parametern, die über ihre Namen identifiziert werden, wie man im folgenden Beispiel sieht.

```
1 zahlworte = [{"dt": "eins", "en": "one", "fr": "un"},
2             {"dt": "zwei", "en": "two", "fr": "deux"},
3             {"dt": "drei", "en": "three", "fr": "trois"}]
4
5 for zahlwort in zahlworte:
6     print("{dt:7s} | {en:7s} | {fr:7s}".format(**zahlwort))
```

Damit ergibt sich die folgende Ausgabe:

```
eins    | one    | un
zwei    | two    | deux
drei    | three  | trois
```

Diese Verwendung von Dictionaries führt einerseits zu einer impliziten Dokumentation des Codes und zum anderen zu einer größeren Stabilität des Codes bei Erweiterungen.

Ähnlich kann man bei Funktionen vorgehen, um Argumente mit Hilfe von Schlüsselworten zu übergeben. In Abschnitt *Schlüsselworte und Defaultwerte* hatten wir diese Möglichkeit für festgelegte Variablennamen kennengelernt. Im folgenden Beispiel wird gezeigt, wie man Objekte, die mit beliebigen Schlüsselworten übergeben werden, als Dictionary in der Funktion verfügbar macht.

```
1 def test(x, **kwargs):
2     print(x)
3     for k in kwargs:
4         print(k, kwargs[k])
5
6 test(0, foo=1, spam=2)
```

In Zeile 1 wird in der Argumentliste wieder ein durch zwei Sternchen gekennzeichneteter Name eines Dictionaries angegeben, in dem die durch nicht vorgegebene Schlüsselworte übergebenen Argumente gesammelt werden. Wir nennen dieses Dictionary hier `kwargs` für »keyword arguments«. Ein anderer, im Rahmen der Vorgaben für Variablennamen erlaubter Name wäre jedoch genauso möglich. Entsprechend dieser Erläuterungen sieht die Ausgabe der Funktion folgendermaßen aus:

```
0
foo 1
spam 2
```

Neben zusätzlichen, mit Schlüsselworten versehenen Argumenten kann man auch noch zusätzliche Argumente ohne Schlüsselwort übergeben. Diese werden in einem Tupel gesammelt und stehen somit für die weitere Verarbeitung zur Verfügung. Dies wird im folgenden Beispiel an einer Funktion zur Berechnung eines Mittelwerts illustriert.

```
def mittelwert1(x, *args):
    sum = x
    for x in args:
        sum = sum+x
    return sum/(len(args)+1)

print(mittelwert1(1, 2.5, 0.7, 3.8, 2.9))
```

Die Funktion `mittelwert1()` hat ein Pflichtargument, das sicherstellt, dass sich der Mittelwert sinnvoll berechnen lässt. Darüber hinaus kann eine beliebige Anzahl von Argumenten übergeben werden, die in einem Tupel, hier `args` genannt, gesammelt werden. Im Gegensatz zu dem zuvor besprochenen Dictionary-Argument, das durch zwei vorangestellte Sternchen gekennzeichnet ist, wird das Tupelargument, `args` in unserem Beispiel, nur durch ein einziges Sternchen gekennzeichnet. Will man sowohl Tupelargumente als auch Schlüsselwortargumente ermöglichen, so müssen diese genau in dieser Reihenfolge vorkommen. Im Argument einer Funktionsdefinition könnte also `f(*args, **kwargs)` stehen.

Alternativ zum vorigen Beispiel könnte man natürlich auch wie in folgendem Beispiel ein einziges Tupelargument vorsehen. Dies impliziert im Aufruf zusätzliche Klammern. Außerdem sollte in diesem Beispiel überprüft werden, ob das Tupel mindestens ein Element enthält.

```
def mittelwert2(argumente):
    sum = 0
    for x in argumente:
        sum = sum+x
    return sum/len(argumente)

print(mittelwert2((1, 2.5, 0.7, 3.8, 2.9)))
```

Eine weitere Anwendung von Dictionaries betrifft Mehrfachverzweigungen, die wir bereits im Abschnitt *Verzweigungen* diskutiert hatten. Häufig will man auf eine Liste von vorgegebenen Werten abprüfen und je nach Wert bestimmte Anweisungen ausführen. Programmiersprachen stellen hierfür häufig eine so genannte *case-* oder *switch-*Anweisung zur Verfügung. In folgendem Pascal-Beispiel wird abhängig vom Wert der Variable `i` eine bestimmte mathematische Funktion auf die Variable `x` angewandt:<sup>5</sup>

```
case i of
  0: x := 0;
  1: x := sin(x);
  2: x := cos(x);
  3: x := exp(x);
  4: x := ln(x);
end
```

In Python gibt es die Mehrfachverzweigung in dieser Form nicht. Stattdessen kann man in solchen Fällen häufig ein Dictionary einsetzen.

```
1 def func1():
2     print("führe Funktion 1 aus")
3
4 def func2():
5     print("führe Funktion 2 aus")
6
7 def func3():
8     print("führe Funktion 3 aus")
9
10 def default():
```

<sup>5</sup> nach: K. Jensen, N. Wirth, *PASCAL User Manual and Report*, S. 31 (Springer, 1975)

```
11     print("führe Defaultfunktion aus")
12
13 def verzweigung(n):
14     alternativen = {1: func1, 2: func2, 3: func3}
15     try:
16         alternativen[n]()
17     except KeyError:
18         default()
19
20 for n in [1, 2.7, "???"]:
21     verzweigung(n)
```

Der wesentliche Teil ist hier in den Zeilen 13-18 in eine Funktion verpackt, was man jedoch nicht zwingend machen muss. Entscheidend ist, dass man in der Zeile 14 ein Dictionary definiert, das hier abhängig von dem vorgegebenen Integer eine Funktion zurückgibt, die dann in Zeile 16 aufgerufen wird. Außerdem ist in den Zeilen 17 und 18 die Möglichkeit vorgesehen, bei einer anderen als den im Dictionary vorgesehenen Eingaben eine Defaultfunktion auszuführen. Iteriert man über die Liste in Zeile 20, so erhält man die folgende Ausgabe:

```
führe Funktion 1 aus
führe Defaultfunktion aus
führe Defaultfunktion aus
```



---

## Ein- und Ausgabe

---

Das Ergebnis der Abarbeitung eines Programms wird in den meisten Fällen die Ausgabe des Ergebnisses zur Folge haben. In den bisherigen Kapiteln haben wir die Ausgabe auf dem Bildschirm mit Hilfe der `print`-Funktion kennengelernt. Insbesondere bei umfangreicheren Ausgaben wird man das Ergebnis in eine Datei schreiben wollen, um es zu speichern oder später weiter zu verarbeiten, beispielsweise mittels eines Grafikprogramms. Häufig ist es auch notwendig, dem Programm Parameter zu übergeben, entweder beim Programmaufruf über Argumente auf der Kommandozeile, auf Anfrage des Programms über die Tastatur oder durch Einlesen aus einer Datei. Im Folgenden werden wir uns zunächst die Eingabe über die Kommandozeile und die Tastatur ansehen und uns anschließend mit der Ein- und Ausgabe von Daten mit Hilfe von Dateien beschäftigen.

### 7.1 Eingabe über die Kommandozeile und die Tastatur

Ein Python-Programm, nennen wir es `foo.py`<sup>1</sup>, lässt sich von der Kommandozeile mit Hilfe des Aufrufs

```
python foo.py
```

starten. An diesen Aufruf kann man jedoch auch weitere Argumente anhängen, die innerhalb des Programms verfügbar sind. Ein solcher Aufruf könnte beispielsweise lauten

```
python foo.py Hallo 3
```

Es könnte sich dabei um den Aufruf eines Programms handeln, das den angegebenen String so oft ausgibt wie es durch das letzte Argument des Aufrufs vorgegeben ist. Der Zugriff auf die Argumente erfolgt dabei über die `argv`-Variable des `sys`-Moduls, wobei `argv` für »argument vector« steht. Eine Realisierung des Programms könnte also folgendermaßen aussehen:

```
1 import sys
2
3 print(sys.argv)
4 for n in range(int(sys.argv[2])):
5     print(sys.argv[1])
```

Dieser Code ergibt die folgende Ausgabe

```
['foo.py', 'Hallo', '3']
Hallo
Hallo
Hallo
```

Aus der ersten Zeile der Ausgabe wird deutlich, dass die Variable `sys.argv` die Kommandozeilenargumente in Form einer Liste enthält. Dabei gibt das erste Element den Namen des aufgerufenen Programms an. Dies ist unter anderem dann von Interesse, wenn man das Programm unter verschiedenen Namen aufrufen kann und dabei ein unterschiedliches Verhalten erreichen möchte. Des Weiteren zeigt die erste Zeile der Ausgabe, dass alle Argumente in der Liste `sys.argv` als Strings auftreten. Daher musste das letzte Argument des Programmaufrufs in Zeile 4 des Programmcodes erst mit Hilfe der `int()`-Funktion in einen Integer umgewandelt werden.

---

<sup>1</sup> Zum Ursprung dieses Namens siehe [RFC 3092](#).

Will man für ein Programm eine flexible oder umfangreiche Übergabe von Optionen auf der Kommandozeile vorsehen, so lohnt sich ein Blick auf das `argparse`-Modul<sup>2</sup>. Dieses stellt einige nützliche Funktionalitäten zur Verfügung, unter anderem auch die automatisierte Erstellung einer Hilfeausgabe.

Eingaben können auch während des Programmablaufs erfolgen. Dies geschieht mit Hilfe der `input()`-Funktion, die in dem folgenden Beispiel illustriert wird.

```
1 while 1:
2     try:
3         x = input("Aufgabe: ")
4         print(eval(x))
5     except SyntaxError:
6         break
```

Uns kommt es zunächst auf die Zeile 3 an. Die `input()`-Funktion gibt das Stringargument aus und gibt die Eingabe als String, hier an die Variable `x`, zurück. Diese Funktionalität wird in dem Beispiel verwendet, um vom Benutzer eingegebene Rechenaufgaben zu lösen. Hierzu wird mit der Zeile 1 eine Dauerschleife eingeleitet, in der in Zeile 3 versucht wird, eine Aufgabe einzulesen. In Zeile 4 wird die `eval()`-Funktion verwendet, um den String als Anweisung zu interpretieren und auszuführen. Tritt dabei ein `SyntaxError` auf, so wird die Dauerschleife in den Zeilen 5 und 6 beendet.

Die Eingabe kann aber auch einfach in einem Tupel bestehen, dessen Bestandteile an mehrere Variablen übergeben werden:

```
x = y = 1
while x*y:
    x, y = eval(input("Multiplikanden: "))
    print("Produkt = {}".format(x*y))
```

Dabei muss die Eingabe das erforderliche Komma enthalten:

```
Multiplikanden: 4, 5
Produkt = 20
Multiplikanden: 3, 0
Produkt = 0
```

In diesem Beispiel wird die Schleife beendet, sobald das Produkt gleich Null ist, was dem Wahrheitswert `False` entspricht. In einem anderen Beispiel wird eine Liste eingegeben, die in einer Schleife abgearbeitet wird.

```
zahlen = eval(input("Geben Sie eine Liste ein: "))
for n in zahlen:
    print("{:6}\t{:10}".format(n, n**2))
```

Die Eingabe einer Liste gibt die Listenelemente und die zugehörigen Quadrate aus:

```
Geben Sie eine Liste ein: [-17, 5, 247]
-17          289
 5           25
247         61009
```

## 7.2 Lesen und Schreiben von Dateien

Häufig wird man statt der manuellen Eingabe von Daten und der Ausgabe von Ergebnissen auf dem Bildschirm das Einlesen aus einer Datei und das Schreiben in eine Datei vorziehen. Wir betrachten zunächst den Fall, dass eine Datei vorliegt, auf deren Inhalt wir in einem Programm zugreifen wollen. Für die folgenden Beispiele nehmen wir an, dass eine Datei namens `foo_utf8.dat` mit dem Inhalt

Einführung in das  
Programmieren für

---

<sup>2</sup> Dieses Modul ist unter dem Titel `argparse – Parser for command-line options, arguments and sub-commands` in der Python-Dokumentation beschrieben.

Physiker und  
Materialwissenschaftler

existiert. Dabei liege diese Datei in der UTF-8-Kodierung vor. Zur Illustration sei noch eine Datei `foo_latin1.dat` vorhanden, die die ISO-8859-1-Kodierung, auch als Latin-1-Kodierung bekannt, verwendet. Während in der ersten Datei der Umlaut »ü« hexadezimal durch `C3BC` kodiert ist, ist er in der zweiten Datei hexadezimal als `FC` dargestellt.

Bevor Daten aus dieser Datei gelesen werden können, muss die Datei geöffnet werden. Dies könnte wie in der ersten Zeile gezeigt geschehen:

```
1 >>> datei = open("foo_utf8.dat")
2 >>> datei
3 <_io.TextIOWrapper name='foo.dat' encoding='UTF-8'>
```

Damit haben wir ein Dateiojekt erhalten, das den Zugriff auf die Datei mit dem in der ersten Zeile als Argument angegebenen Namen ermöglicht. Falls nichts anderes beim Öffnen der Datei angegeben wird, ist die Datei lediglich zum Lesen geöffnet. Die Datei kann also nicht überschrieben werden, und es kann auch nichts angefügt werden.

Standardmäßig erwartet wird die auf dem jeweiligen System bevorzugte Kodierung. In unserem Fall ist dies UTF-8.

```
>>> import locale
>>> locale.getpreferredencoding()
'UTF-8'
```

Der Versuch, auf eine nicht existierende Datei lesend zuzugreifen, wird mit einem `IOError` beantwortet:

```
>>> datei = open("foo.txt")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'foo.txt'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'foo.txt'
```

Nachdem die Datei geöffnet wurde, gibt es verschiedene Möglichkeiten, auf ihren Inhalt zuzugreifen. Mit der `read()`-Funktion wird, sofern kein Argument eingegeben wurde, die gesamte Datei in einen String eingelesen:

```
>>> datei.read()
'Einführung in das\nProgrammieren für\nPhysiker und\nMaterialwissenschaftler\n'
```

Die in dem String auftretenden `\n` geben Zeilenumbrüche an<sup>3</sup>. Die Datei besteht also aus vier Zeilen. Versucht man auf die gleiche Weise, die Datei `foo_latin1.dat` einzulesen, erhält man einen `UnicodeDecodeError` weil die den Umlauten entsprechenden Bytes in dieser Datei nicht im Rahmen der UTF-8-Kodierung interpretiert werden können.

```
>>> datei = open("foo_latin1.dat")
>>> datei.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.5/codecs.py", line 321, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf8' codec can't decode byte 0xfc in position 4: invalid start byte
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.5/codecs.py", line 321, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf8' codec can't decode byte 0xfc in position 4: invalid start byte
```

Dagegen funktioniert das Einlesen problemlos, wenn man die richtige Kodierung angibt.

<sup>3</sup> Hierzu und zu den folgenden Überlegungen zur Zeichenkodierung sei auf den Anhang *Unicode* hingewiesen.

```
>>> datei = open("foo_latin1.dat", encoding="latin1")
>>> datei.read()
'Einführung in das\nProgrammieren für\nPhysiker und\nMaterialwissenschaftler\n'
```

Nach dem Lesen einer gesamten Datei mit der `read()`-Funktion steht der Zeiger, der die aktuelle Position in der Datei angibt, am Dateiende. Ein weiteres Lesen ergibt daher nur einen leeren String wie Zeile 7 in dem folgenden Beispiel zeigt.

```
1 >>> datei = open("foo_utf8.dat")
2 >>> datei.read()
3 'Einführung in das\nProgrammieren für\nPhysiker und\nMaterialwissenschaftler\n'
4 >>> datei.tell()
5 75
6 >>> datei.read()
7 ''
```

In den Zeilen 4 und 5 haben wir mit Hilfe der `tell()`-Methode die aktuelle Position des Dateizeigers abgefragt. Dabei zählen die Zeilenumbrüche mit. Man muss allerdings beachten, dass das Resultat der `tell()`-Methode auf der Bytedarstellung beruht und das UTF8-kodierte `ü` als zwei Bytes zählt.

Mit Hilfe der `seek()`-Funktion kann man gezielt an bestimmte Stellen der Datei springen, wobei allerdings wieder die Bytedarstellung relevant ist. Es besteht also potentiell die Gefahr, mitten in einem Mehrbyte-Code zu landen. Daher ist es sinnvoll, `seek()` auf der Basis von Positionen zu verwenden, die mit `tell()` bestimmt wurden.

Eindeutig ist jedoch der Dateianfang, der der Zeigerposition 0 entspricht. Nach einem `seek(0)` liest der zweite Aufruf der `read()`-Funktion im folgenden Beispiel nochmals die gesamte Datei ein:

```
1 >>> datei = open("foo_utf8.dat")
2 >>> datei.read()
3 'Einführung in das\nProgrammieren für\nPhysiker und\nMaterialwissenschaftler\n'
4 >>> datei.seek(0)
5 >>> datei.read(10)
6 'Einführung'
7 >>> datei.read(10)
8 ' in das\nPr'
```

Man kann sich die Funktionsweise wie bei einem Magnetband vorstellen, bei dem die Position des Lesekopfes durch die `tell()`-Funktion angegeben wird, während die `seek()`-Funktion den Lesekopf neu positioniert. Im gerade gezeigten Beispiel wird der Lesekopf an den Anfang, d.h. auf die absolute Position 0 zurückgesetzt. Anschließend werden zweimal je zehn Zeichen eingelesen.

Nicht immer möchte man die ganze Datei auf einmal einlesen, sei es weil die Datei sehr groß ist oder weil man den Inhalt zum Beispiel zeilenweise verarbeiten möchte. Hierzu stellt Python verschiedene Möglichkeiten zur Verfügung. Mit Hilfe der `readlines()`-Funktion lassen sich die einzelnen Zeile für die weitere Verarbeitung in eine Liste aufnehmen:

```
>>> datei = open("foo_utf8.dat")
>>> inhalt = datei.readlines()
>>> print(inhalt)
['Einführung in das\n', 'Programmieren für\n', 'Physiker und\n',
'Materialwissenschaftler\n']
```

Bei der Verarbeitung der einzelnen Zeilen ist zu beachten, dass die Zeichenketten am Ende noch die Zeilenumbruchkennzeichnung `\n` enthalten.

Zeilen lassen sich auch einzeln einlesen.

```
>>> datei = open("foo_utf8.dat")
>>> datei.readline()
'Einführung in das\n'
>>> datei.readline()
'Programmieren für\n'
>>> datei.readline()
```



```
'Physiker und\n'
>>> datei.readline()
'Materialwissenschaftler\n'
>>> datei.readline()
''
```

Nachdem alle Zeilen eingelesen wurden, steht der Dateizeiger am Dateiende, so dass bei weiteren Aufrufen der `readline()`-Funktion nur ein leerer String zurückgegeben wird.

Eine elegante Methode, die Zeilen einer Datei in einer Schleife abzuarbeiten, zeigt das folgende Beispiel.

```
>>> datei = open("foo_utf8.dat")
>>> for zeile in datei:
...     print(zeile.upper())
...
EINFÜHRUNG IN DAS
PROGRAMMIEREN FÜR
PHYSIKER UND
MATERIALWISSENSCHAFTLER
```

Will man die zusätzlichen Leerzeilen vermeiden, so muss man das `\n` am Ende der Zeilen entfernen, entweder unter Verwendung der `rstrip()`-Methode oder durch Verwendung eines Slices.

```
>>> datei = open("foo_utf8.dat")
>>> for zeile in datei:
...     print(zeile.upper().rstrip("\n"))
...
EINFÜHRUNG IN DAS
PROGRAMMIEREN FÜR
PHYSIKER UND
MATERIALWISSENSCHAFTLER
```

Die gleiche Ausgabe erhält man mit

```
>>> datei = open("foo_utf8.dat")
>>> for zeile in datei:
...     print(zeile[:-1].upper())
...

```

In allen bisherigen Beispielen haben wir eine Anweisung unterschlagen, die man am Ende der Arbeit mit einer Datei immer ausführen lassen sollte. Nachdem man eine Datei zunächst geöffnet hat, sollte man sie am Ende auch wieder schließen.

```
>>> datei = open("foo_utf8.dat")
>>> inhalt = datei.read()
>>> datei.closed
False
>>> datei.close()
>>> datei.closed
True
```

Das Schließen einer Datei gibt die im Zusammenhang mit der geöffneten Datei benötigten Ressourcen wieder frei und bewahrt einen unter Umständen auch vor einem teilweisen oder vollständigen Verlust der geschriebenen Daten.

Bevor wir uns mit dem Schreiben von Dateien beschäftigen, müssen wir uns zunächst noch ansehen, wie man Zahlen aus einer Datei liest, eine bei numerischen Arbeiten sehr häufige Situation. Als Eingabedatei sei eine Datei namens `spam.dat`<sup>4</sup> mit dem Inhalt

---

<sup>4</sup> Die Verwendung von `spam` in Python-Beispielen als Name ohne spezifische Bedeutung ist ein Verweis auf einen Sketch der Komikergruppe Monty Python (siehe [Wikipedia: Spam-Sketch](#)).

```

1.37  2.59
10.3  -1.3
5.8   2.0

```

gegeben. Das folgende Programm berechnet zeilenweise das Produkt des jeweiligen Zahlenpaares.

```

1 daten = open("spam.dat")
2 for zeile in daten:
3     x, y = zeile.split()
4     print(float(x)*float(y))
5 daten.close()

```

In Zeile 3 wird jede eingelesene Zeile an Leerräumen wie zum Beispiel Leerstellen oder Tabulatorzeichen, aufgeteilt. Damit ergeben sich je zwei Strings, die die Information über die jeweilige Zahl enthalten. Allerdings kann man Strings nicht miteinander multiplizieren. Daher muss in Zeile 4 vor der Multiplikation mit Hilfe der `float()`-Funktion eine Umwandlung in Gleitkommazahlen erfolgen. Das Schließen der Datei erfolgt in Zeile 5 außerhalb der `for`-Schleife, da sonst die Datei bereits nach dem Einlesen der ersten Zeile geschlossen würde.

Als Alternative zu der im vorigen Beispiel gezeigten expliziten Umwandlung kann es sinnvoll sein, die von Python zur Verfügung gestellte `map()`-Funktion zu verwenden. Dies ist insbesondere bei mehreren Zahlen oder wenn deren Anzahl nicht bekannt ist, nützlich. Das Beispiel lautet dann

```

1 daten = open("spam.dat")
2 for zeile in daten:
3     x, y = map(float, zeile.split())
4     print(x*y)
5 daten.close()

```

Dabei wird in Zeile 3 die `float()`-Funktion zur Umwandlung aller Elemente der Liste `zeile.split()` in Gleitkommazahlen angewandt.

In einem Programm möchte man nicht nur Daten aus einer Datei einlesen, sondern vor allem auch die Ergebnisse in einer Datei speichern. Wie beim Lesen aus Dateien muss man beim Schreiben in Dateien zunächst eine Datei öffnen. Dies kann auf verschiedene Weise geschehen. Betrachten wir zunächst das folgende Beispiel:

```

1 datei = open("foo.dat", "w")
2 for n in range(5):
3     datei.write("{:4}{:4}\n".format(n, n*n))
4 datei.close()

```

Die Anweisung in Zeile 1 kennen wir im Prinzip schon, nur dass jetzt das zweite Argument explizit auf `w`, also »write« gesetzt ist. Damit wird die Datei `foo.dat` zum Schreiben geöffnet. Ob die Datei schon existiert, ist dabei unerheblich. Existiert sie nicht, so wird eine neue Datei angelegt. Existiert die Datei dagegen schon, so wird ihre Länge vor dem Schreiben auf Null gesetzt. Damit wird die zuvor existierende Datei effektiv überschrieben. In der dritten Zeile erfolgt das Schreiben in die Datei mit Hilfe der `write()`-Methode. Wie bei dem uns bereits bekannten `print`-Befehl muss als Argument ein String angegeben werden. Dabei können natürlich die im Abschnitt *Formatierung von Ausgaben* besprochenen Formatspezifikationen verwendet werden. Zu beachten ist, dass im Gegensatz zur `print`-Anweisung bei Bedarf ein Zeilenumbruch explizit mit `\n` zu verlangen ist. Die `read()`- und die `write()`-Methode sind also insofern symmetrisch als in beiden Fällen der Zeilenumbruch in den jeweiligen Zeichenketten explizit auftritt. Nicht vergessen werden sollte das Schließen der Datei in Zeile 4, da ansonsten die Gefahr bestehen könnte, dass Daten verloren gehen.

Öffnet man eine existierende Datei im Modus `r+`, so kann man von ihr lesen und in sie schreiben. Ähnliches geschieht bei `w+`, wobei bei Bedarf jedoch eine neue Datei angelegt wird. Gelegentlich möchte man Daten an eine Datei anhängen. In diesem Falle verwendet man den Modus `a` für »append« oder `a+` falls man aus der Datei auch lesen möchte.

**+** Ab Python 3.3 gibt es auch noch die Option `"x"`, die nur dann eine Datei erfolgreich öffnet, falls diese Datei noch nicht existiert.

Bei numerischen Rechnungen ist es oft sinnvoll, die verwendeten Parameter im Dateinamen aufzuführen wie es im folgenden Beispiel gezeigt ist. Dazu wird in der Zeile 2 beim Öffnen der Datei ein geeigneter Konvertierungsspezifikator verwendet.

```
1 parameter = 12
2 datei = open("resultate_{:05}.dat".format(parameter), "w")
3 for n in range(parameter):
4     datei.write("{:10}\n".format(n*n))
5 datei.close()
```

Entsprechend dem Wert der Variable `parameter` erfolgt die Ausgabe in die Datei `resultate_00012.dat`. Die Formatierung, den Integer bis zur geforderten Feldbreite von links mit Nullen aufzufüllen, ist hier nützlich, um bei einer großen Anzahl von Parameterwerten eine ordentlich sortierte Übersicht über die vorhandenen Dateien bekommen zu können.

Da das Überschreiben von Dateien unangenehme Folgen haben kann, ist es nützlich zu wissen, wie man die Existenz einer Datei überprüfen kann. Mit einer Methode aus dem `os.path`-Modul geht das wie im Folgenden gezeigt,

```
import os

datei = "foo.dat"
if os.path.exists(datei):
    print("Achtung! {} existiert bereits.".format(datei))
else:
    print("Die Datei {} existiert noch nicht.".format(datei))
```

Existiert die Datei bereits, so würde man in einer echten Anwendung dem Benutzer wohl die Möglichkeit geben, das Programm an dieser Stelle geordnet zu beenden oder einen alternativen Dateinamen anzugeben.

Abschließend sei noch erwähnt, dass Python für bestimmte Dateiformate spezielle Module zum Lesen und Schreiben zur Verfügung stellt. Hierzu gehört zum Beispiel das `csv`-Modul, das den Zugriff auf Dateien im `csv`-Format<sup>5</sup> erlaubt. Dieses Format wird häufig von Tabellenkalkulationsprogrammen wie zum Beispiel `Microsoft Excel` oder `Calc` aus `OpenOffice` bzw. `LibreOffice` benutzt. Hat man solche Programme bei der Erfassung der Daten verwendet, so ist es sinnvoll, sich das `csv`-Modul<sup>6</sup> anzusehen.

Bei einer aufwendigen Übergabe von Parametern an ein Programm kann auch das `ConfigParser`-Modul<sup>7</sup> von Interesse sein, das mit Dateien im `INI`-Format umgehen kann. Dabei werden Parameter in Name-Wert-Paaren beschrieben, wobei eine Unterteilung in Abschnitte möglich ist.

---

<sup>5</sup> `csv` steht für *comma separated values*, wobei allerdings kein verbindlicher Standard existiert. Beispielsweise können Felder genauso gut durch Kommas wie durch Strichpunkte getrennt sein.

<sup>6</sup> Dieses Modul ist unter dem Titel `csv — CSV File Reading and Writing` in der Python-Dokumentation beschrieben.

<sup>7</sup> Dieses Modul ist unter dem Titel `ConfigParser — Configuration file parser` in der Python-Dokumentation beschrieben.



---

## Numerische Programmbibliotheken am Beispiel von NumPy/SciPy

---

Bei der numerischen Lösung von Problemen aus der Physik oder den Materialwissenschaften benötigt man häufig Funktionalität, die von Python und der zugehörigen Standardbibliothek nicht unmittelbar zur Verfügung gestellt wird. Man denke an spezielle Funktionen, die nicht im `math`-Modul enthalten sind, beispielsweise Besselfunktionen, an die numerische Auswertung von Integralen oder die Lösung von Differentialgleichungen. Auch die Verwendung von Matrizen ist nicht unmittelbar möglich. Zwar erlauben es Listen, matrizenartige Objekte darzustellen, aber es ist nicht direkt möglich, Matrizen miteinander zu multiplizieren. Hier bleibt einem die Möglichkeit, die benötigte Funktionalität selbst zu implementieren oder auf eines der vielen für die verschiedensten Problemstellungen zur Verfügung stehenden Module zurückzugreifen. Letzteres ist sicherlich bequemer. Zudem handelt es sich häufig um effiziente und sorgfältige Implementierungen. In diesem Kapitel wollen wir uns beispielhaft das Numerikmodul NumPy/SciPy ansehen.

### 8.1 Installation

Nachdem NumPy/SciPy nicht Bestandteil von Pythons Standardbibliothek ist, steht es standardmäßig noch nicht zur Verfügung. Ob es installiert ist, lässt sich leicht mit Hilfe eines `import`-Versuchs feststellen:

```
>>> import numpy
>>> numpy.__version__
'1.11.0'
>>> import scipy
>>> scipy.__version__
'0.17.1'
```

Haben Sie beispielsweise die Anaconda-Distribution installiert, so sollte das Importieren problemlos funktionieren, auch wenn vielleicht eine andere Versionsnummer angezeigt wird <sup>1</sup>. Sind die Bibliotheken nicht vorhanden, so würde eine `ImportError`-Ausnahme geworfen. Dann müssen NumPy und SciPy installiert werden. Informationen hierzu finden Sie für verschiedene Betriebssysteme unter [www.scipy.org/install.html](http://www.scipy.org/install.html).

### 8.2 Arrays und Anwendungen

Im Folgenden soll anhand von wenigen Beispielen ein kurzer Einblick in die Möglichkeiten gegeben werden, die NumPy bietet. Das Programm

```
1 import numpy as np
2
3 matrixA = np.array([[1.3, 2.5], [-1.7, 3.9]])
4 matrixB = np.array([[2.1, -4.5], [0.9, -2.1]])
5 print(matrixA, end="\n\n")
6 print(matrixB, end="\n\n")
7 print(np.dot(matrixA, matrixB), end="\n\n")
```

---

<sup>1</sup> Man beachte, dass vor und nach `version` jeweils *zwei* Unterstriche einzugeben sind.

```

8 anweisung = "{:g}*{:g}+{:g}*{:g}".format(matrixA[0, 0], matrixB[0, 0],
9                                           matrixA[0, 1], matrixB[1, 0])
10 print("{} = {:g}".format(anweisung, eval(anweisung)))

```

erzeugt die Ausgabe

```

[[ 1.3  2.5]
 [-1.7  3.9]]

[[ 2.1 -4.5]
 [ 0.9 -2.1]]

[[ 4.98 -11.1 ]
 [-0.06 -0.54]]

1.3*2.1+2.5*0.9 = 4.98

```

In der ersten Zeile des Programmcodes wird das NumPy-Modul geladen, wobei die Verwendung des kurzen Bezeichners `np` eine häufig verwendete Konvention ist, um im Programmtext auf die Herkunft der verwendeten Methoden hinzuweisen. In den Zeilen 3 und 4 ist eine Möglichkeit gezeigt, Matrizen mit Hilfe von NumPy aus einer Liste zu erzeugen. In Zeile 9 wird das Produkt der beiden Matrizen berechnet und ausgegeben. Die in den Zeilen 11 und 12 erzeugte Ausgabe weist die Korrektheit anhand des obersten Diagonalelements des Produkts durch Multiplikation der entsprechenden Zeile von `matrixA` und Spalte von `matrixB` nach. Zugleich erkennt man aus Zeile 11, dass die Nummerierung der Matrixelemente, wie die von Python-Listen, mit 0 beginnt. Im Gegensatz zu Listen werden die Matrixelemente jedoch durch Kommas getrennt in einem einzigen eckigen Klammerpaar angegeben. Daraus wird deutlich, dass im Gegensatz zu der hierarchischen Konstruktion der Listen von Listen hier die verschiedenen Dimensionen eines Arrays gleichberechtigt sind.

Multipliziert man zwei Arrays mit Hilfe des Multiplikationsoperators `*`, so wird keine Matrixmultiplikation durchgeführt. Vielmehr werden die Elemente an den jeweils gleichen Positionen der beiden Matrizen miteinander multipliziert.

```

>>> matrixA*matrixB
array([[ 2.73 -11.25],
       [-1.53  -8.19]])

```

**+** Ab Python 3.5 und NumPy 1.10 kann die Matrixmultiplikation von Arrays mit Hilfe des Operators `@` ausgeführt werden:

```

>>> matrixA @ matrixB
array([[ 4.98, -11.1 ],
       [-0.06, -0.54]])

```

Das folgende Beispiel zeigt die Berechnung von Skalarprodukt, dyadischem Produkt sowie Kreuzprodukt für zwei Vektoren.

```

import numpy as np

vecA = np.array([2, -3, 0])
vecB = np.array([5, 4, 0])
print np.dot(vecA, vecB)
print
print np.outer(vecA, vecB)
print
print np.cross(vecA, vecB)

```

Als Ausgabe findet man erwartungsgemäß

```

-2

[[ 10  8  0]
 [-15 -12 0]
 [ 0  0  0]]

```

```
[ 0  0 23]
```

Interessant ist die Möglichkeit, Arrays als Argumente von mathematischen Funktionen zu verwenden:

```
1 >>> import numpy as np
2 >>> import math
3 >>> x = np.linspace(0, 1, 11)
4 >>> x
5 array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ])
6 >>> np.exp(x)
7 array([ 1.          ,  1.10517092,  1.22140276,  1.34985881,  1.4918247 ,
8         1.64872127,  1.8221188 ,  2.01375271,  2.22554093,  2.45960311,
9         2.71828183])
10 >>> math.exp(x)
11 Traceback (most recent call last):
12   File "<stdin>", line 1, in <module>
13   TypeError: only length-1 arrays can be converted to Python scalars
14 Traceback (most recent call last):
15   File "<stdin>", line 1, in <module>
16   TypeError: only length-1 arrays can be converted to Python scalars
```

Eine praktische Methode, um ein Array mit äquidistanten Werten zwischen zwei Grenzen zu erzeugen, ist die `linspace()`-Funktion, die in Zeile 3 zur Erzeugung der Argumente benutzt wird. Der Aufruf der Exponentialfunktion aus NumPy gibt in den Zeilen 6-9 ein ganzes Array der entsprechenden Ergebnisse zurück. Zeilen 10-13 zeigen, dass dies mit der Exponentialfunktion aus dem `math`-Modul nicht möglich wäre. Bei umfangreichen Arrays spart die Verwendung der NumPy-Funktion Rechenzeit gegenüber einer Schleife, die die Funktion nacheinander auf jedes Element einzeln anwendet. Als Faustregel gilt bei der Verwendung von NumPy, dass im Hinblick auf die Effizienz eines Programms `for`-Schleifen nach Möglichkeit durch geeignete `array`-Operationen ersetzt werden sollten.

Zum Abschluss kehren wir noch einmal zu den Matrizen zurück und sehen uns einige Funktionen aus dem Bereich der Linearen Algebra an.

```
1 >>> import numpy as np
2 >>> from numpy import linalg as LA
3 >>> a = np.array([[1, 3], [2, 5]])
4 >>> LA.det(a)
5 -1.0
6 >>> LA.inv(a)
7 array([[ -5.,  3.],
8        [ 2., -1.]])
9 >>> np.dot(a, LA.inv(a))
10 array([[ 1.,  0.],
11        [ 0.,  1.]])
12 >>> LA.eig(a)
13 (array([-0.16227766,  6.16227766]), array([[ -0.93246475, -0.50245469],
14        [ 0.36126098, -0.86460354]]))
15 >>> eigenwerte, eigenvektoren = LA.eig(a)
16 >>> for i in range(len(eigenwerte)):
17 ...     print(np.dot(a, eigenvektoren[:, i]), eigenwerte[i]*eigenvektoren[:, i])
18 ...
19 [ 0.1513182 -0.05862459] [ 0.1513182 -0.05862459]
20 [-3.09626531 -5.32792709] [-3.09626531 -5.32792709]
```

In den Zeilen 1 und 2 werden zunächst die weiter unten benötigten Funktionen importiert. Dabei bezieht sich Zeile 2 auf die Funktionen, die im Modul zur Linearen Algebra von NumPy enthalten sind. Nachdem in Zeile 3 eine Matrix definiert wurde, wird in Zeile 4 die zugehörige Determinante bestimmt. In Zeile 6 wird die inverse Matrix berechnet und die Korrektheit des Ergebnisses durch Multiplikation mit der ursprünglichen Matrix nachgewiesen. In Zeile 9 werden die Eigenwerte und Eigenvektoren der Matrix `a` berechnet. Um auf das Tupel nicht über die entsprechenden Indizes zugreifen zu müssen, kann man das Ergebnis wie in Zeile 12 gezeigt gleich in die Eigenwerte und die Eigenvektoren aufteilen. In den Zeilen 13-17 wird schließlich nachgewiesen, dass die Eigenwerte und Eigenvektoren korrekt sind. Dabei wird verwendet, dass die Spalten der Eigenvektormatrix den Eigenvektoren

ren entsprechen. Der erste Eigenvektor wird mit `eigenvektor[:, 0]` angegeben. Wie bei Listen bedeutet der einzelne Doppelpunkt, dass der erste Index von seinem Minimalwert 0 bis zu seinem Maximalwert, hier 1, läuft.

## 8.3 Numerische Integration

Als Anwendung von SciPy betrachten wir die numerische Auswertung des Integrals

$$J_0(1) = \frac{1}{\pi} \int_0^\pi \cos(\cos(x)) dx.$$

Hierbei ist  $J_0(z)$  die Besselfunktion erster Gattung und nullter Ordnung, deren Wert wir probierhalber ebenfalls mit Hilfe von SciPy berechnen lassen können. Das folgende Programm führt die notwendigen Berechnungen durch:

```
1 from math import cos, pi
2 from scipy import integrate, special
3
4 resultat, fehler = integrate.quad(lambda x: cos(cos(x)), 0, pi)
5 print(resultat/pi, fehler/pi)
6 print(special.j0(1))
```

Die zugehörige Ausgabe lautet

```
0.7651976865579664 7.610964456309953e-11
0.765197686558
```

In den ersten beiden Programmzeilen werden zunächst die benötigten Unterpakete von SciPy, `integrate` für die Integration und `special` für spezielle Funktionen, sowie der Kosinus und die Kreiszahl aus dem `math`-Modul importiert. In Zeile 4 wird zur Integration die Funktion `quad()` (von »Quadratur« oder Englisch »quadrature«) aus dem `integrate`-Modul verwendet. `quad()` verlangt zwingend eine Funktion, die den Integranden beschreibt und hier als Lambdafunktion angegeben ist, sowie die Integrationsgrenzen. Ausgegeben werden das Resultat der numerischen Integration und eine Abschätzung des absoluten Fehlers. Zur Beurteilung der Qualität des Resultats verwenden wir in Zeile 6 die Besselfunktion `j0()` aus dem `special`-Modul von SciPy. Der Vergleich des Ergebnisses der numerischen Integration mit  $J_0(1)$ , für das SciPy einen ebenfalls im Prinzip mit Fehlern behafteten numerischen Wert bestimmt, ergibt perfekte Übereinstimmung.

Wenn man die Konstante `inf` aus NumPy importiert, kann man auch uneigentliche Integrale berechnen:

```
from scipy import integrate
import numpy as np

resultat, fehler = integrate.quad(lambda x: 1/(x*x+1), -np.inf, np.inf)
print(resultat/np.pi, fehler)
```

Der erste Wert der Ausgabe

```
1.0 5.155583905474508e-10
```

zeigt, dass der Wert des Integrals

$$\int_{-\infty}^{\infty} \frac{1}{x^2 + 1} dx = \pi$$

korrekt bestimmt wird. Man sollte sich von der Qualität dieses Ergebnisses jedoch nicht täuschen lassen. Nicht immer kann ein numerisches Resultat mit einer solchen Genauigkeit erhalten werden. Manchmal muss das Integrationsproblem auch zunächst geeignet formuliert werden, zum Beispiel in der Nähe von Singularitäten oder wenn der Integrand schnell oszilliert.

## 8.4 Integration gewöhnlicher Differentialgleichungen

Häufig steht man in der Physik und den Materialwissenschaften vor der Aufgabe, Differentialgleichungen zu lösen. Wir beschränken uns hier auf gewöhnliche Differentialgleichungen, die als Anfangswertproblem gelöst werden



sollen. Wir beginnen mit einer Differentialgleichung erster Ordnung

$$\dot{x} = -x^2,$$

die sich durch Trennung der Variablen lösen lässt. Will man nicht selbst ein Lösungsverfahren, zum Beispiel das Euler- oder Runge-Kutta-Verfahren implementieren, so kann man wiederum auf das SciPy-Paket zurückgreifen. Dort wird unter anderem die Funktion `odeint` zur Verfügung gestellt, die wir im Folgenden benutzen wollen. Der Name der Funktion enthält das englische »ordinary differential equation« in abgekürzter Form.

Da der Funktionsaufruf in einem solchen Fall durchaus komplexer sein kann, muss man sich zunächst über die von der Funktion erwarteten Argumente informieren. In Python kann man das leicht mit der `help()`-Funktion tun, aber auch im Internet unter <http://docs.scipy.org/doc/> finden sich Dokumentationen.

```
>>> from scipy import integrate
>>> help(integrate.odeint)
```

Help on function odeint in module scipy.integrate.odepack:

```
odeint(func, y0, t, args=(), Dfun=None, col_deriv=0, full_output=0, ml=None,
mu=None, rtol=None, atol=None, tcrit=None, h0=0.0, hmax=0.0, hmin=0.0, ixpr=0,
mxstep=0, mxhnil=0, mxordn=12, mxords=5, printmessg=0)
```

Integrate a system of ordinary differential equations.

Solve a system of ordinary differential equations using lsoda from the FORTRAN library odepack.

Solves the initial value problem for stiff or non-stiff systems of first order ode-s::

```
dy/dt = func(y,t0,...)
```

where y can be a vector.

Parameters

-----

func : callable(y, t0, ...)

Computes the derivative of y at t0.

y0 : array

Initial condition on y (can be a vector).

t : array

A sequence of time points for which to solve for y. The initial value point should be the first element of this sequence.

[...]

Returns

-----

y : array, shape (len(t), len(y0))

Array containing the value of y for each desired time in t, with the initial value y0 in the first row.

[...]

Wir haben an den mit [...] markierten Stellen einigen Text ausgelassen. Bereits der gleich zu Beginn angegebene Funktionsaufruf zeigt, dass eine Vielzahl an Parametern übergeben werden können. Die meisten sind jedoch mit Defaultwerten belegt, so dass wir nicht gezwungen sind, sie zu spezifizieren. Sollte es jedoch zum Beispiel nötig sein, den relativen oder absoluten Fehler besser zu kontrollieren, so kann man dies tun. Beim Aufruf der Funktion `odeint()` müssen wir aber auf jeden Fall eine aufrufbare Funktion übergeben, die es erlaubt, die Ableitung zu berechnen. Diese Funktion muss zumindest zwei Argumente besitzen, nämlich die aktuellen Werte der abhängigen und der unabhängigen Variablen. Ferner benötigen wir einen Anfangswert und einen Vektor, der die Werte der unabhängigen Variablen enthält, zu der die gesuchte Lösung der Differentialgleichung bestimmt werden soll. Das folgende Programm berechnet eine numerische Lösung für die oben genannte Differentialgleichung.

```
1 import numpy as np
2 from scipy import integrate
```

```

3
4 pts = np.linspace(0, 100, 101)
5 ergebnis = integrate.odeint(lambda x, t: -x**2, 1, pts)[: , 0]
6 for n in range(len(pts)):
7     exakt = 1/(1+pts[n])
8     print("{:3.0f}  {:10.8f}  {:11.5g}".format(
9         pts[n],
10        ergebnis[n],
11        (ergebnis[n]-exakt)/exakt))

```

In Zeile 1 importieren wir zunächst das `numpy`-Modul, das wir benötigen, um in Zeile 4 ein Array mit äquidistanten Zeitpunkten zu erzeugen. Außerdem wird in Zeile 2 das `integrate`-Unterpaket aus `SciPy` importiert, aus dem wir die Funktion `odeint()` zum Lösen der Differentialgleichung verwenden wollen. Dies geschieht in Zeile 5. Dabei haben wir die Ableitung der Einfachheit halber als Lambda-Funktion in den Aufruf geschrieben. Der Anfangswert ist im zweiten Argument gleich 1 gesetzt und das dritte Argument enthält das NumPy-Array mit den Punkten, für die die Lösung bestimmt werden soll. `odeint()` gibt ein zweidimensionales Array zurück, das bei Differentialgleichungssystemen in jeder Spalte den Zeitverlauf für eine Komponente enthält. Da wir in unserem Beispiel nur eine einzige Differentialgleichung erster Ordnung vorliegen haben, wählen wir explizit die Spalte 0 aus. Ab Zeile 6 wird das exakte Ergebnis an den vorgegebenen Punkten ausgewertet und die Lösung samt den Werten der unabhängigen Variablen und des relativen Fehlers ausgegeben. Führt man das Programm aus, so erhält man etwa die folgende Ausgabe

```

0  1.000000000      0
1  0.500000000  1.1693e-09
2  0.333333332 -5.1641e-08
3  0.249999998 -8.066e-08
4  0.199999998 -1.0617e-07
[...]
95 0.01041665 -1.1279e-06
96 0.01030927 -1.126e-06
97 0.01020407 -1.1343e-06
98 0.01010100 -1.146e-06
99 0.00999999 -1.1439e-06
100 0.00990098 -1.135e-06

```

Wir verzichten darauf, die gesamte Ausgabe zu reproduzieren. Man sieht aber bereits an diesen Zeilen, dass das erhaltene Ergebnis nicht exakt ist und der relative Fehler mit zunehmendem Abstand vom Startwert zunimmt. Dennoch ist der relative Fehler gut kontrolliert, so dass hier eine brauchbare Lösung erzeugt wurde.

Wenn wir uns die Dokumentation der `odeint()`-Funktion noch einmal ansehen, stellen wir fest, dass die Funktion zunächst für Differentialgleichungen erster Ordnung gedacht ist. Allerdings kann es sich bei der Variablen `y` um einen Vektor handeln. Dies gibt uns die Möglichkeit, auch Differentialgleichungen höherer Ordnung numerisch zu behandeln. Wir müssen sie nur in ein System von Differentialgleichungen erster Ordnung umformulieren. Betrachten wir als Beispiel die Differentialgleichung eines gedämpften harmonischen Oszillators

$$\ddot{x} + \gamma \dot{x} + x = 0,$$

wobei  $\gamma$  die Dämpfungskonstante ist. Diese Differentialgleichung ist zu dem Satz zweier Differentialgleichungen erster Ordnung

$$\dot{p} = -x - \gamma p$$

$$\dot{x} = p$$

äquivalent, den wir nun mit den Anfangsbedingungen  $x(0) = 0$ ,  $p(0) = 1$  numerisch lösen wollen.

```

1 from scipy import integrate
2 import numpy as np
3 from math import exp, sin, sqrt
4
5 def ableitung(y, t, gamma):
6     x, p = y

```

```

7     return np.array([p, -x-gamma*p])
8
9     pts = np.linspace(0, 10, 101)
10    anfangsbedingungen = np.array([0, 1])
11    gamma = 0.3
12    omega = sqrt(1-0.25*gamma**2)
13
14    ergebnis = integrate.odeint(ableitung, anfangsbedingungen, pts, (gamma,))
15    ort = ergebnis[:, 0]
16
17    for n in range(len(pts)):
18        exakt = exp(-0.5*gamma*pts[n])*sin(omega*pts[n])/omega
19        print("{:4.1f}   {:8.5f}   {:11.5g}".format(
20            pts[n], ort[n], ort[n]-exakt))

```

Nachdem wir bereits ein Beispiel besprochen haben, können wir uns hier auf die neuen Aspekte beschränken. In den Zeilen 5-7 wurde diesmal eine Funktion definiert, die einen Vektor mit den benötigten ersten Ableitungen zurückgibt. Die in Zeile 10 definierten Anfangsbedingungen müssen jetzt ebenfalls aus einem Vektor bestehen. Außerdem enthält die Funktion `ableitung()` ein zusätzliches Argument, nämlich `gamma`, das übergeben werden muss. Dazu sieht `odeint()` ein viertes Argument vor, das ein Tupel sein muss und dessen Elemente dem dritten und eventuell weiteren Argumenten der Ableitungsfunktion zugeordnet werden. Es ist zu beachten, dass ein einzelner eingeklammerter Variablenname nur dann als Tupel interpretiert wird, wenn dieser von einem Komma gefolgt wird.

? Warum wird hier im Gegensatz zum ersten Beispiel nicht der relative sondern der absolute Fehler ausgegeben?

Auch wenn der hier vorgestellte Programmcode nur die Position des Oszillators als Funktion der Zeit ausgibt, könnte man genauso seine Geschwindigkeit ausgeben. Nachdem wir zwei Differentialgleichungen erster Ordnung gelöst haben, ist die Geschwindigkeit bei der von uns gewählten Reihenfolge als zweite Spalte `ergebnis[:, 1]` in der Ergebnismatrix zugänglich.

? Wie ändert sich die Trajektorie, wenn in der Bewegungsgleichung  $\dot{x}$  durch  $\dot{x}^2$  ersetzt wird, so dass die Bewegungsgleichung nichtlinear wird? Mit den in Kapitel *Erstellung von Grafiken* dargestellten Techniken können Sie die berechneten Trajektorien leicht vergleichen.



---

## Objektorientiertes Programmieren

---

In diesem Kapitel soll ein, wenn auch knapper, Einblick in das objektorientierte Programmieren gegeben werden. Dabei spielt das Konzept von Objekten eine zentrale Rolle, die gewisse Eigenschaften, sogenannte Attribute, haben, sowie Methoden bereitstellen, um mit dem Objekt zu kommunizieren. Damit wird eine Kapselung von Daten erreicht, und es werden definierte Schnittstellen festgelegt. Attribute und Methoden sind in einer Art Bauplan, der Klassendefinition, festgelegt. Im Programm wird dann mit Instanzen, tatsächlichen Realisierungen der abstrakten Definition, gearbeitet. Ein weiteres wichtiges Konzept ist die Vererbung, die es erlaubt, verwandte Klassen voneinander abzuleiten. Damit wird es möglich, dass eine Unterklasse Attribute und Methoden von einer Basisklasse erbt.

### 9.1 Klassen, Attribute und Methoden

Diese abstrakten Bemerkungen werden klarer, wenn wir gleich ein konkretes Beispiel betrachten. Wir haben eine ganze Reihe an Datentypen kennengelernt, die von Python zur Verfügung gestellt werden. Echte Brüche waren jedoch nicht darunter<sup>1</sup>. Im Folgenden soll nun gezeigt werden, wie ein solcher Datentyp zur Verfügung gestellt werden kann. Offenbar besitzt ein Bruch Attribute, nämlich den Wert des Zählers sowie den des Nenners. Darüber hinaus gibt es Methoden. Man kann beispielsweise Brüche addieren oder multiplizieren oder sie in verschiedenen Weisen ausgeben.

Wir definieren uns nun zunächst eine Klasse `Bruch` zusammen mit der Konstruktormethode `__init__()`

```
1 class Bruch:
2
3     def __init__(self, zaehler, nenner=1):
4         self.zaehler = zaehler
5         self.nenner = nenner
```

und wenden diese Klassendefinition sofort an

```
1 >>> x = Bruch(2, 5)
2 >>> print(x.zaehler, x.nenner)
3 2 5
```

Während die Klassendefinition die Attribute `zaehler` und `nenner` in abstrakter Weise definiert, wird bei der Zuweisung an die Variable `x` eine Instanz, also eine konkrete Realisierung eines Bruchs erzeugt. Dabei wird die Konstruktormethode `__init__()`, die in der Klassendefinition in den Zeilen 3-5 definiert wird, mit Hilfe des Klassennamens aufgerufen. In unserem Beispiel handelt es sich um die Anweisung `Bruch(2, 5)`. Die Variable `self`, die in der Konstruktormethode als erstes Argument auftritt, kann man sich als Platzhalter für die tatsächliche Instanz vorstellen. Man beachte, dass dieses Argument im Aufruf der Konstruktormethode immer fehlt. Bei der Ausführung der Konstruktormethode werden in den Zeilen 4 und 5 die Argumente der Konstruktormethode den Attributen des Objekts zugewiesen. Auf diese Attribute kann im Hauptprogramm zugegriffen werden, wie die Zeile 2 im zweiten Codeblock zeigt. Dabei wird der Attributname mit einem Punkt an den Variablenname des Objekts angehängt. Schließlich sei noch angemerkt, dass der Defaultwert für `nenner` hier dafür sorgt, dass

---

<sup>1</sup> Python stellt Brüche über das Modul `fractions` zur Verfügung. Weitere Informationen sind in der Python-Dokumentation unter <https://docs.python.org/3/library/fractions.html> verfügbar.

auch bei nur einem Argument im Aufruf der Konstruktormethode eine sinnvolle Instanz des `Bruch`-Objekts erzeugt wird. Im Folgenden wird angenommen, dass die Konstruktormethode mit strikt positiven ganzen Zahlen aufgerufen wird.

Nun müssen wir unsere `Bruch`-Klasse mit etwas Funktionalität ausstatten. Zunächst einmal wollen wir den Fall vorsehen, dass der Bruch gekürzt werden kann. Außerdem wollen wir den Bruch ausgeben können, ohne explizit auf seine Attribute zugreifen zu müssen. Dies wird von dem folgenden Code erledigt

```

1 class Bruch:
2
3     def __init__(self, zaehler, nenner=1):
4         self.zaehler = zaehler
5         self.nenner = nenner
6         self.__reduce()
7
8     def __reduce(self):
9         a = self.zaehler
10        b = self.nenner
11        while a!=b and a!=1 and b!=1:
12            a, b = min(a, b), abs(a-b)
13        if a==b:
14            self.zaehler = self.zaehler//a
15            self.nenner = self.nenner//a
16
17    def __str__(self):
18        if self.nenner!=1:
19            return "{}/{ {}".format(self.zaehler, self.nenner)
20        else:
21            return str(self.zaehler)

```

wie hier zu sehen ist

```

>>> x = Bruch(21, 15)
>>> print(x)
7/5

```

Zum Kürzen des Bruchs rufen wir in Zeile 6 die Methode `__reduce()` auf. Da es sich um eine in der Klasse `Bruch` definierte Methode handelt, muss ein von einem Punkt gefolgt `self` vorangestellt werden. Sonst würde nach einer Funktion außerhalb der Klassendefinition gesucht werden. Die zwei Unterstriche zu Beginn des Methodennamens machen die Methode zu einer privaten Methode, die nur zum Aufruf innerhalb der Klasse gedacht ist. Auf entsprechende Weise kann man auch private Attribute einführen. Wie schon in der Konstruktormethode `__init__()` muss das erste Argument `self` sein. Damit stehen die Attribute `zaehler` und `nenner` in der Methode zur Verfügung. In den Zeilen 9-15 ist der euklidische Algorithmus zur Bestimmung des größten gemeinsamen Teilers implementiert, mit dessen Hilfe der Bruch gekürzt werden kann.

In den Zeilen 17-21 ist eine Methode mit dem speziellen Namen `__str__()` implementiert, die automatisch immer dann aufgerufen wird, wenn das `Bruch`-Objekt in einen String umgewandelt werden soll. Dies ist beispielsweise bei der Ausgabe mit `print` der Fall. Solche speziellen Methodennamen existieren zum Beispiel auch für die Addition, die Multiplikation und einige andere Operationen mehr. Wir wollen uns hier auf die Implementierung der ersten beiden Methoden, der Umwandlung in den `float`-Typ sowie zweier Vergleichsmethoden beschränken und erweitern unsere Klassendefinition in folgender Weise:

```

1 class Bruch:
2
3     def __init__(self, zaehler, nenner=1):
4         self.zaehler = zaehler
5         self.nenner = nenner
6         self.__reduce()
7
8     def __reduce(self):
9         a = self.zaehler
10        b = self.nenner
11        while a!=b and a!=1 and b!=1:

```

```

12         a, b = min(a, b), abs(a-b)
13     if a==b:
14         self.zaehler = self.zaehler//a
15         self.nenner = self.nenner//a
16
17     def __str__(self):
18         if self.nenner!=1:
19             return "{}/{ {}".format(self.zaehler, self.nenner)
20         else:
21             return str(self.zaehler)
22
23     def __add__(self, other):
24         return Bruch(self.zaehler*other.nenner+self.nenner*other.zaehler,
25                      self.nenner*other.nenner)
26
27     def __mul__(self, other):
28         return Bruch(self.zaehler*other.zaehler,
29                      self.nenner*other.nenner)
30
31     def __float__(self):
32         return float(self.zaehler)/self.nenner
33
34     def __lt__(self, other):
35         return self.zaehler*other.nenner < other.zaehler*self.nenner
36
37     def __eq__(self, other):
38         return self.zaehler==other.zaehler and self.nenner==other.nenner

```

Jetzt sind wir in der Lage, echte Brüche zu addieren und zu multiplizieren, sie in Gleitkommazahlen umzuwandeln und zu vergleichen:

```

>>> x = Bruch(2, 5)
>>> y = Bruch(3, 7)
>>> print(x+y)
29/35
>>> print(x*y)
6/35
>>> float(y)
0.428571428571
>>> x>y
False
>>> x = Bruch(20, 15)
>>> y = Bruch(8, 6)
>>> x==y
True

```

Dieses Beispiel zeigt, dass man in einem Programm durchaus mehrere Instanzen einer Klasse verwenden kann. Das objektorientierte Programmieren erlaubt es, mit diesen Instanzen zu arbeiten, ohne sich um deren »Innenleben« kümmern zu müssen. Zähler und Nenner sind in unserem Beispiel zwar zugänglich, bei der Addition oder Multiplikation brauchen wir uns um diese jedoch nicht explizit zu kümmern. Dadurch gewinnt das Programm enorm an Übersichtlichkeit. Dies ist ein nicht zu unterschätzender Vorteil dieser Programmweise.

Bis jetzt haben wir spezielle Methoden definiert, die mit Operatoren wie zum Beispiel + und \* verknüpft sind. Damit wurden diese Operatoren, die zunächst für andere Datentypen definiert waren, auch für Brüche bereitgestellt. Man spricht hier vom Überladen von Operatoren. Wir können aber auch Methoden definieren, die mit ihrem Namen von außerhalb der Klasse aufgerufen werden. Als Beispiel definieren wir eine `prettyprint()`-Methode, die den Bruch in mehreren Zeilen mit einem horizontalen Bruchstrich ausgeben soll. Unsere Klassendefinition nimmt dann die folgende Form an:

```

class Bruch:

    def __init__(self, zaehler, nenner=1):
        self.zaehler = zaehler

```

```
self.nenner = nenner
self.__reduce()

def __reduce(self):
    a = self.zaehler
    b = self.nenner
    while a!=b and a!=1 and b!=1:
        a, b = min(a, b), abs(a-b)
    if a==b:
        self.zaehler = self.zaehler//a
        self.nenner = self.nenner//a

def __str__(self):
    if self.nenner!=1:
        return "{}/{ {}".format(self.zaehler, self.nenner)
    else:
        return str(self.zaehler)

def __add__(self, other):
    return Bruch(self.zaehler*other.nenner+self.nenner*other.zaehler,
                 self.nenner*other.nenner)

def __mul__(self, other):
    return Bruch(self.zaehler*other.zaehler,
                 self.nenner*other.nenner)

def __float__(self):
    return float(self.zaehler)/self.nenner

def __lt__(self, other):
    return self.zaehler*other.nenner < other.zaehler*self.nenner

def __eq__(self, other):
    return self.zaehler==other.zaehler and self.nenner==other.nenner

def prettyprint(self):
    zaehler_str = str(self.zaehler)
    nenner_str = str(self.nenner)
    feldbreite = max(len(zaehler_str), len(nenner_str))
    bruchstrich = "-"*feldbreite
    print "{}\n{}\n{}".format(zaehler_str.center(feldbreite),
                               bruchstrich,
                               nenner_str.center(feldbreite))
```

Nun können wir die `prettyprint()`-Methode anwenden:

```
>>> x = Bruch(213, 53)
>>> y = Bruch(7, 3091)
>>> z = x*y
>>> z.prettyprint()
1491
-----
163823
>>> (x+y).prettyprint()
658754
-----
163823
```

Wie beim Aufruf der `__reduce()`-Methode in der Konstruktormethode der `Bruch`-Klasse wird der Methodenname mit einem Punkt an das Objekt angehängt. Letzteres kann durch eine Variable, hier `z`, spezifiziert sein oder durch einen Ausdruck, hier `x+y`. Im Allgemeinen können beim Aufruf einer Methode natürlich auch Argumente übergeben werden.

Nun wird klar, dass wir auch in früheren Kapiteln immer wieder Methoden aufgerufen haben ohne uns dessen



wirklich bewusst gewesen zu sein. Wenn wir beispielsweise im Kapitel über *Ein- und Ausgabe* die Anweisung `datei.write(...)` verwendet haben, so hatten wir ein Dateiobjekt `datei` benutzt und dessen `write`-Methode aufgerufen.

**+** Zum Abschluss dieses Unterkapitels soll noch auf zwei Aspekte im Zusammenhang mit dem hier entwickelten Beispiel eingegangen werden. Vor allem wenn man eine Klasse programmiert, um sie anderen Nutzern zur Verfügung zu stellen, ist die Dokumentation der Methoden wichtig, mit deren Hilfe der Nutzer mit den Objekten arbeiten kann. Wie schon im Kapitel *Dokumentation von Funktionen* für Funktionen besprochen, kann auch bei Methoden in einer Klassendefinition ein Dokumentationstext direkt nach der mit `def` beginnenden Zeile eingebaut werden. Die Methode `prettyprint` könnte dann folgendermaßen aussehen:

```
def prettyprint(self):
    """Gibt den Bruch dreizeilig aus, wobei Zähler und Nenner
    zentriert gesetzt sind.

    """
    zaehler_str = str(self.zaehler)
    nenner_str = str(self.nenner)
    feldbreite = max(len(zaehler_str), len(nenner_str))
    bruchstrich = "-"*feldbreite
    print "{}\n{}\n{}".format(zaehler_str.center(feldbreite),
                              bruchstrich,
                              nenner_str.center(feldbreite))
```

In der Python-Shell kann man Information über alle Methoden in folgender Weise bekommen:

```
>>> help(Bruch)
```

```
Help on class Bruch in module __main__:
```

```
class Bruch
|   Methods defined here:
|
|   __add__(self, other)
|
|   __eq__(self, other)
|
|   __float__(self)
|
|   __init__(self, zaehler, nenner=1)
|
|   __lt__(self, other)
|
|   __mul__(self, other)
|
|   __str__(self)
|
|   prettyprint(self)
|       Gibt den Bruch dreizeilig aus, wobei Zähler und Nenner
|       zentriert gesetzt sind.
[...]
```

```
>>> help(Bruch.prettyprint)
```

```
Help on method prettyprint in module __main__:
```

```
prettyprint(self)
    Gibt den Bruch dreizeilig aus, wobei Zähler und Nenner
    zentriert gesetzt sind.
```

Man kann also Informationen über alle Methoden oder aber über eine spezifische Methode erhalten, wie wir das schon in den Kapiteln *Funktionen für reelle Zahlen* und *Integration gewöhnlicher Differentialgleichungen* kennengelernt haben. Hier sehen wir zudem, dass Python sich bemüht, hilfreiche Informationen über die Methoden

zur Verfügung zu stellen, selbst wenn keine expliziten Dokumentationstexte vorhanden sind. Dies ist jedoch keinesfalls eine Entschuldigung dafür, auf eine Dokumentation von Seiten des Programmierers zu verzichten!

**+** Ein Manko unserer bisherigen Version der `Bruch`-Klasse besteht darin, dass stillschweigend vorausgesetzt wird, dass Zähler und Nenner als `Integers` übergeben werden müssen. Dies ist unnötig restriktiv. Es würde ausreichen, wenn die entsprechenden Werte in `Integers` umwandelbar sind. Die `__init__()`-Methode könnte zum Beispiel wie folgt erweitert werden:

```
1 def __init__(self, zaehler, nenner=1):
2     try:
3         self.zaehler = int(zaehler)
4         self.nenner = int(nenner)
5     except ValueError:
6         raise ValueError("Die Bruchklasse erwartet ganzzahlige Zähler und Nenner.")
7     self.__reduce()
```

In den Zeilen 3 und 4 wird versucht, die Werte der Variablen `zaehler` und `nenner` in `Integers` umzuwandeln. Da dies durchaus misslingen kann, wie wir gleich noch sehen werden, wird hier die `ValueError`-Ausnahme abgefangen. Um dem aufrufenden Programm den Fehler mitzuteilen, wird in Zeile 6 diese Ausnahme gleich wieder geworfen. Dies bietet die Gelegenheit, eine aussagekräftige Fehlermeldung mitzuliefern, und ermöglicht es dem aufrufenden Programmteil, den Fehler adäquat zu behandeln.

```
1 >>> x = Bruch("33", "47")
2 >>> print(x)
3 33/47
4 >>> y = Bruch("a", "b")
5 Traceback (most recent call last):
6   File "<stdin>", line 1, in <module>
7   File "bruch.py", line 10, in __init__
8     raise ValueError, "Die Bruchklasse erwartet ganzzahlige Zähler und Nenner."
9 ValueError: Die Bruchklasse erwartet ganzzahlige Zähler und Nenner.
10 >>> z = Bruch([22, 33], 44)
11 Traceback (most recent call last):
12   File "<stdin>", line 1, in <module>
13   File "bruch.py", line 7, in __init__
14     self.zaehler = int(zaehler)
15 TypeError: int() argument must be a string or a number, not 'list'
16 Traceback (most recent call last):
17   File "<stdin>", line 1, in <module>
18   File "bruch.py", line 7, in __init__
19     self.zaehler = int(zaehler)
20 TypeError: int() argument must be a string or a number, not 'list'
```

In den Zeilen 1 bis 3 sieht man, dass die neue `__init__()`-Methode in der Lage ist, auch Strings korrekt zu verarbeiten, sofern sie sich in `Integers` umwandeln lassen. In Zeile 4 ist dies nicht der Fall, und es wird somit eine `ValueError`-Ausnahme ausgelöst, die mit einer entsprechenden Fehlermeldung in Zeile 9 versehen ist. Wie die Zeilen 10 bis 15 zeigen, kann man allerdings auch eine `TypeError`-Ausnahme hervorrufen, die somit in der `__init__()`-Methode noch adäquat abgefangen werden müsste.

## 9.2 Vererbung

Bei der Definition von Klassen kann man auf Attribute und Methoden anderer Klassen zurückgreifen, die dann nicht noch einmal implementiert werden müssen, sondern vererbt werden. Wir wollen die Vererbung anhand zweier Objekte illustrieren, dem Massepunkt und dem Rotationskörper. Der Massepunkt besitze seine Position als Attribut. Als Methoden wollen wir vorsehen, dass der Körper verschoben werden kann und dass sich seine aktuelle Position ausgeben lässt. Diese Attribute und Methoden lassen sich für den Schwerpunkt des Rotationskörpers direkt übernehmen. Hinzu kommt als neues Attribut der Vektor in dessen Richtung die Achse des Rotationskörpers zeigt. Neben der Methode, die die Richtung der Achse ausgibt, wollen wir eine Methode implementieren, die die Achse des Körpers dreht, wobei Drehwinkel und Drehachse als Argumente übergeben werden.

Wir definieren zunächst eine Klasse für den Massepunkt.

```

import numpy as np

class Massepunkt:

    def __init__(self):
        self.pos = np.array([0, 0, 0])

    def verschiebe(self, shift):
        self.pos = self.pos+shift

    def position(self):
        print("Die Masse befindet sich am Ort ({:g}, {:g}, {:g}).".format(*self.pos))

```

Die Konstruktormethode legt den Massepunkt in den Ursprung. Da wir für die Drehung des Rotationskörpers Skalar- und Kreuzprodukt benötigen, verwenden wir die NumPy-Bibliothek um Vektoren zu definieren. Die verschiebe()-Methode verschiebt den Massepunkt um den Vektor shift. Mit der position()-Methode lässt sich die aktuelle Position des Massepunkts ausgeben, auf die sich auch mit Hilfe des Attributs pos zugreifen lässt. Testen wir zunächst die Funktionalität dieser Klasse:

```

>>> m = Massepunkt()
>>> m.pos
array([0, 0, 0])
>>> m.position()
Die Masse befindet sich am Ort (0, 0, 0).
>>> m.verschiebe(np.array([2, 9, -3]))
>>> m.position()
Die Masse befindet sich am Ort (2, 9, -3).
>>> m.verschiebe(np.array([1, -5, 0]))
>>> m.position()
Die Masse befindet sich am Ort (3, 4, -3).

```

Von der Massepunkt-Klasse leiten wir nun die Rotationskoerper-Klasse ab und bekommen so insgesamt den folgenden Code:

```

1 import numpy as np
2 from math import sqrt, sin, cos, pi
3
4 class Massepunkt:
5
6     def __init__(self):
7         self.pos = np.array([0, 0, 0])
8
9     def verschiebe(self, shift):
10         self.pos = self.pos+shift
11
12     def position(self):
13         print("Die Masse befindet sich am Ort ({:g}, {:g}, {:g}).".format(*self.pos))
14
15
16 class Rotationskoerper(Massepunkt):
17
18     def __init__(self):
19         self.achse = np.array([0, 0, 1])
20         Massepunkt.__init__(self)
21
22     def drehe(self, drehachse, winkel):
23         drehachse = drehachse/np.linalg.norm(drehachse)
24         winkel = winkel*pi/180
25         self.achse = self.achse*cos(winkel) + \
26                     drehachse*np.dot(drehachse, self.achse)*(1-cos(winkel)) + \
27                     np.cross(drehachse, self.achse)*sin(winkel)
28
29     def orientierung(self):

```

```
30     print("Die Achse des starren Körpers liegt in Richtung "  
31           "des Vektors ({:g}, {:g}, {:g}).".format(*self.achse))
```

In Zeile 16 wird die Basisklasse `Massepunkt` als Argument der Klasse `Rotationskoerper` angegeben, so dass Attribute und Methoden der Basisklasse geerbt werden können. Im Prinzip ist es möglich, auch mehrere Basisklassen anzugeben. In der Konstruktormethode der `Rotationskoerper`-Klasse initialisieren wir zunächst die Achse, die die Orientierung des Körpers angibt. Um die Position des Körpers zu initialisieren, greifen wir auf die Konstruktormethode der `Massepunkt`-Klasse zurück. Dies ist sinnvoll, aber keineswegs verpflichtend.

**+** Um auf die Konstruktormethode der Elternklasse zuzugreifen, könnte man in Zeile 20 alternativ `super().__init__()` verwenden.

In der `Rotationskoerper`-Klasse definieren wir zwei neue Methoden. Es wäre durchaus auch möglich, Methoden der `Massepunkt`-Klasse zu überladen. Wir wollen dies hier jedoch nicht tun, da wir die Methoden dieser Klasse unverändert verwenden wollen. Sehen wir uns nun an, ob die `Rotationskoerper`-Klasse wie gewünscht funktioniert:

```
>>> rk = Rotationskoerper()  
>>> rk.position()  
Die Masse befindet sich am Ort (0, 0, 0).  
>>> rk.orientierung()  
Die Achse des starren Körpers liegt in Richtung des Vektors (0, 0, 1).  
>>> rk.verschiebe(np.array([2, 1, 3]))  
>>> rk.position()  
Die Masse befindet sich am Ort (2, 1, 3).  
>>> rk.orientierung()  
Die Achse des starren Körpers liegt in Richtung des Vektors (0, 0, 1).  
>>> rk.drehe(np.array([1, 1, 0]), 45)  
>>> rk.orientierung()  
Die Achse des starren Körpers liegt in Richtung des Vektors (0.5, -0.5, 0.707107).  
>>> rk.drehe(np.array([0, 0, 1]), 45)  
>>> rk.orientierung()  
Die Achse des starren Körpers liegt in Richtung des Vektors (0.707107, -5.55112e-17, 0.707107).
```

Das `Rotationskoerper`-Objekt `rk` besitzt tatsächlich sowohl die Objektattribute der `Massepunkt`-Klasse als auch die der `Rotationskoerper`-Klasse. Zudem können die Methoden beider Klassen verwendet werden.

---

## Erstellung von Grafiken

---

Häufig ist es notwendig, die numerischen Ergebnisse eines Programms in grafischer Weise darzustellen. Hierzu gibt es verschiedene Möglichkeiten. Man kann beispielsweise die Ergebnisse zunächst in einer Datei abspeichern und dann mit Hilfe eines speziellen Programms eine grafische Aufbereitung vornehmen. In diesem zweiten Schritt kann man eigene, in Python oder einer anderen Sprache geschriebene Programme einsetzen oder aber auf existierende Programme zurückgreifen. Es gibt eine Vielzahl solcher Programme, aus denen man je nach den jeweiligen Anforderungen ein geeignetes auswählen kann. Ein häufig verwendetes Programm zur Darstellung numerischer Daten ist zum Beispiel **gnuplot**<sup>1</sup>.

Möchte man die Erzeugung der Grafik mit einem eigenen Python-Programm bewerkstelligen, so greift man sinnvollerweise auf vorhandene Programmbibliotheken wie **matplotlib**<sup>2</sup> oder **PyX**<sup>3</sup> zurück. Da in einem solchen Fall sowohl die Daten als auch die Grafik in einem Python-Programm erzeugt werden, hat man die Möglichkeit, beide Schritte in dem gleichen Programm auszuführen. Lassen sich die Daten jedoch nur mit großem numerischem Aufwand erhalten, so sollte man daran denken, die Daten abzuspeichern, selbst wenn man die Grafik anschließend direkt erzeugt. Nichts ist ärgerlicher, als viel Rechenzeit in den Sand gesetzt zu haben, weil der Grafikteil einen Programmierfehler enthielt und so die errechneten Daten letztlich verloren gingen. Gegebenenfalls kann man den Grafikteil des Programms mit schnell erzeugbaren Daten testen.

Im Folgenden werden wir eine Einführung in die Benutzung von **matplotlib** und **PyX** geben, die allerdings beide wesentlich mächtiger sind als es in diesem Rahmen gezeigt werden kann. Die Beschreibung kann also nur einen allerersten Eindruck geben. Für einen guten Überblick über die Möglichkeiten dieser Grafikmodule empfiehlt sich ein Blick auf die Beispielseiten <http://matplotlib.org/gallery.html> und die Unterseiten von <http://pyx.sourceforge.net/examples/index.html> und <http://sourceforge.net/p/pyx/gallery/index>.

### 10.1 matplotlib

Nachdem SciPy/NumPy keine grafischen Fähigkeiten besitzt, wird in diesem Umfeld zur Erzeugung von Grafiken häufig **matplotlib** empfohlen, das unter anderem auf den Array-Datentyp von NumPy zurückgreift. **matplotlib** kann auf zwei verschiedene Arten verwendet werden. Zum einen kann man mit Hilfe des Python-Interpreters durch direkte Eingabe von Kommandos schnell Grafiken erzeugen. Zum anderen kann man Grafiken aber auch mit einem vollständigen Python-Programm erstellen. Letzteres ist vor allem dann sinnvoll, wenn man Eigenschaften der Grafik im Detail festlegen will. Wir beginnen zunächst mit der ersten Variante und betrachten ein Beispiel, in dem zwei Besselfunktionen unter Verwendung von SciPy dargestellt werden.

```
1 >>> import numpy as np
2 >>> import matplotlib.pyplot as plt
3 >>> from scipy.special import j0, j1
4 >>> x = np.arange(0, 50, 0.2)
5 >>> y1 = j0(x)
6 >>> y2 = j1(x)
7 >>> plt.plot(x, y1)
```

---

<sup>1</sup> Für weitere Informationen siehe die [Gnuplot-Webseite](#).

<sup>2</sup> Die Programmbibliothek zum Herunterladen und weitere Informationen findet man auf der [matplotlib-Webseite](#).

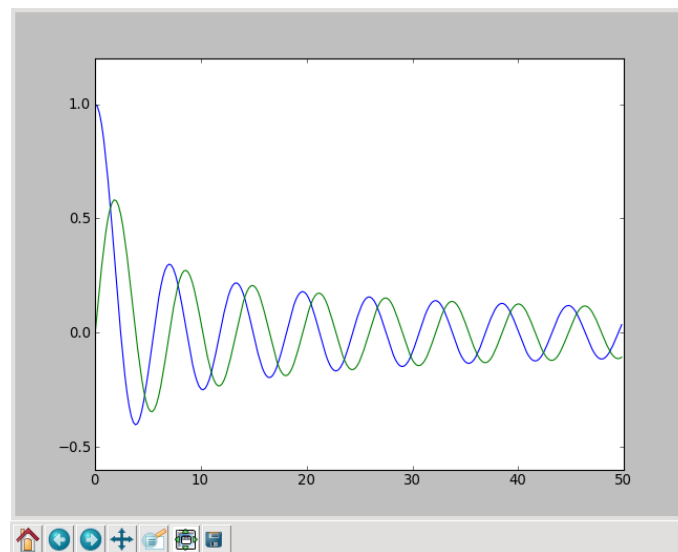
<sup>3</sup> Die Programmbibliothek zum Herunterladen und weitere Informationen findet man auf der [PyX-Webseite](#).

```

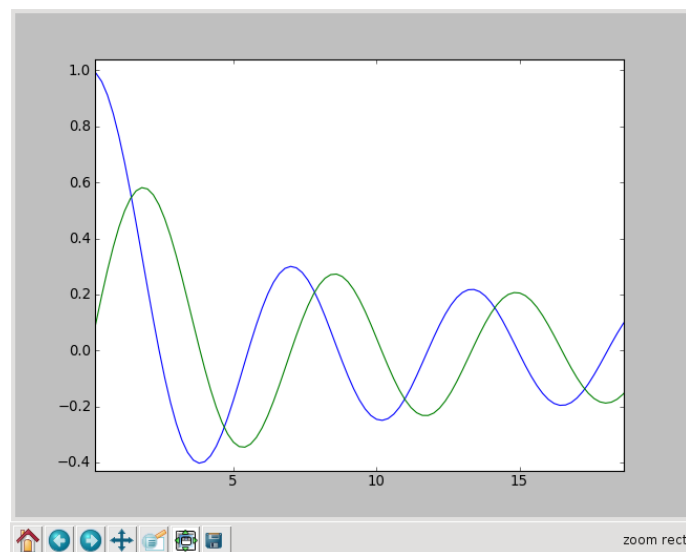
8  [<matplotlib.lines.Line2D object at 0xa273f0c>]
9  >>> plt.plot(x, y2)
10 [<matplotlib.lines.Line2D object at 0xa2732cc>]
11 >>> plt.show()

```

In Zeile 1 wird zunächst das NumPy-Paket geladen, um in Zeile 4 ein Array mit  $x$ -Werten erzeugen zu können. Anschließend wird in Zeile 2 die Plot-Funktionalität von `matplotlib`, soweit sie im Folgenden benötigt wird, geladen<sup>4</sup>. Auch hier wird, wie schon bei NumPy, eine übliche Abkürzung eingeführt, um Tipparbeit zu sparen. Schließlich werden in der Zeile 3 zwei Besselfunktionen aus SciPy importiert, die im Weiteren dargestellt werden sollen. In Zeile 4 wird nun, wie schon angedeutet, ein Array mit den  $x$ -Werten erzeugt, für die in den folgenden beiden Zeilen Arrays mit den Besselfunktionen  $J_0(x)$  und  $J_1(x)$  berechnet werden. In den Zeilen 7-10 werden dann die beiden Funktionsgraphen erzeugt. Wie die Zeilen 8 und 10 andeuten, handelt es sich dabei um `matplotlib.lines.Line2D`-Objekte. Abschließend erfolgt die Darstellung des Graphen in Zeile 11. Die Ausgabe erfolgt auf den Bildschirm, wo die Grafik in einem Fenster, wie in der folgenden Abbildung gezeigt, dargestellt wird.



Die Knöpfe am linken unteren Rand des Fensters kann man benutzen, um zum Beispiel in die Grafik hineinzu-zoomen wie es die folgende Abbildung zeigt oder den dargestellten Ausschnitt zu verschieben.



Mit dem Haussymbol kommt man immer wieder zu der ursprünglichen Darstellung zurück.

<sup>4</sup> Das Laden der Module in den ersten beiden Zeilen wird überflüssig, wenn man die erweiterte Python-Shell `ipython` mit der Option `-pylab` verwendet. Dann führt jede `plot`-Anweisung zu einer Aktualisierung der in einem externen Fenster angezeigten Grafik.

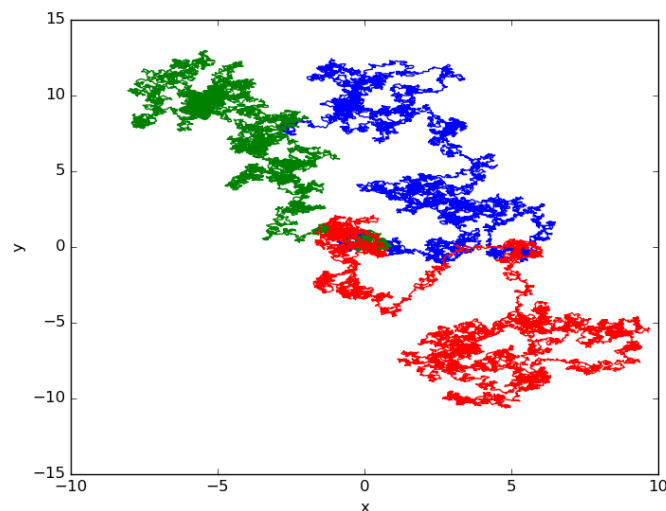
Alternativ zu dieser interaktiven Arbeitsweise kann man Grafiken auch mit Hilfe eines normalen Python-Programms erzeugen. Als Beispiel wollen wir ein Programm betrachten, das eine Zufallsbewegung berechnet und darstellt.

```

1  import numpy as np
2  from numpy.random import rand
3  import matplotlib.pyplot as plt
4  from math import pi
5
6  npts = 10000
7  r = 0.1
8  for _ in range(3):
9      x = np.zeros(npts+1)
10     y = np.zeros(npts+1)
11     richtung = 2*pi*rand(npts)
12     x[1:] = np.cumsum(r*np.cos(richtung))
13     y[1:] = np.cumsum(r*np.sin(richtung))
14     plt.plot(x, y)
15
16 plt.xlabel("x")
17 plt.ylabel("y")
18 plt.savefig("randomwalk.pdf")

```

In den ersten vier Zeilen werden zunächst wieder wie gewohnt Module importiert, wobei in Zeile 2 eine Funktion zur Erzeugung von Zufallszahlen aus NumPy importiert wird. Die Zufallsbewegung wird nun in einer diskreten Weise durch `npts` Punkte spezifiziert, wobei jeweils ein Schritt der Länge `r` in eine Zufallsrichtung, die durch ein Element des Arrays `richtung` festgelegt ist, ausgeführt wird. Diese Schritte werden in den Zeilen 12 und 13 kumulativ aufsummiert, um die Trajektorien zu erzeugen. Durch die Zeilen 9 und 10 und die verschobene Indizierung in den Zeilen 12 und 13 wird dafür gesorgt, dass alle Trajektorien im Ursprung starten. Die durch die Werte in `x` und `y` definierte Trajektorie wird in Zeile 14 geplottet. Insgesamt wird dieser Vorgang mit Hilfe der `for`-Schleife dreimal durchgeführt. Abschließend wird noch eine Achsenbeschriftung hinzugefügt und schließlich die Abbildung in einer PDF-Datei abgespeichert, die im Folgenden dargestellt ist.



Hier stellt man fest, dass `matplotlib` automatisch die Farbe der Linien wechselt. Alternativ hätte man die Farben auch explizit spezifizieren können.

## 10.2 PyX

Eine mögliche Alternative zu `matplotlib` stellt `PyX` dar, das von André Wobst und Jörg Lehmann, in der Anfangsphase der Programmentwicklung als Doktoranden am Lehrstuhl für Theoretische Physik I der Universität

Augsburg tätig, programmiert wurde und noch weiterentwickelt wird. Etwas später kam noch Michael Schindler hinzu. Die drei Buchstaben in PyX stehen für »Postscript«, »Python« und »(La)TeX«<sup>5</sup>. Postscript wurde als Ausgabeformat inzwischen noch durch PDF erweitert. PyX ist in Python geschrieben und wird zur Erzeugung von Grafiken in Python-Programme importiert. Zudem lassen sich bei Bedarf in Python eigene Erweiterungen programmieren. (La)TeX schließlich wird für eine qualitativ hochwertige Textausgabe benutzt, wobei matplotlib ähnliche Möglichkeiten bietet. Seit Version 0.13 läuft PyX ausschließlich unter Python 3.

Mit PyX lassen sich nicht nur Graphen erstellen, sondern auch Schemazeichnungen erzeugen. So wurden die meisten Abbildungen in diesem Manuskript mit PyX erstellt.

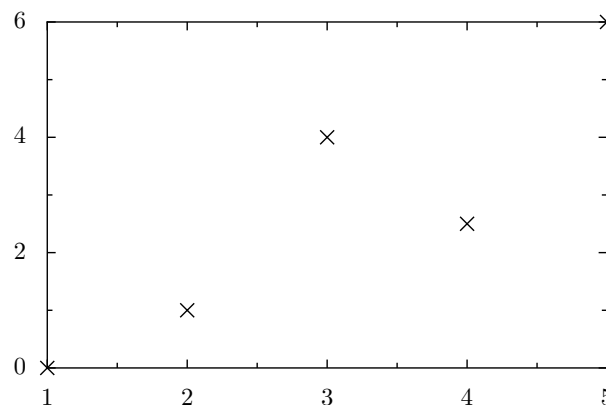
Sehen wir uns zunächst an, wie aus einem in einer Datei vorliegenden Datensatz eine graphische Darstellung erzeugt werden kann. Liegt eine Datei `foo_pyx.dat` mit folgendem Inhalt

```
1  0
2  1
3  4
4  2.5
5  6
```

vor, so erzeugt das Programm

```
1 from pyx import *
2
3 g = graph.graphxy(width=8)
4 g.plot(graph.data.file("foo_pyx.dat", x=1, y=2))
5 g.writePDFfile("foo_pyx")
```

eine PDF-Datei mit Namen `foo_pyx.pdf`, die folgendermaßen aussieht.



In Zeile 1 wird zunächst PyX importiert. Anschließend wird in Zeile 3 ein zweidimensionaler Graph der Breite 8 erzeugt. Wird die Höhe des Graphen nicht spezifiziert, so ist das Seitenverhältnis durch den goldenen Schnitt gegeben. In Zeile 4 werden die Daten aus der angegebenen Datei eingelesen und in den gerade initialisierten Graphen gezeichnet. Dabei geben die Werte der benannten Argumente `x` und `y` die Spalten an, aus denen die jeweiligen Daten zu entnehmen sind. Schließlich wird mit der `writePDFfile`-Methode die PDF-Datei erzeugt.

Um einen gewissen Einblick in die Möglichkeiten von PyX zu gewinnen, wollen wir diese Graphik nun modifizieren. Dabei erzeugt der Code

```
1 from pyx import *
2
3 unit.set(xscale=1.3)
4
5 g = graph.graphxy(width=8,
6                   x=graph.axis.linear(title="$x$"),
7                   y=graph.axis.linear(title="$y$")
8                   )
9 g.plot(graph.data.file("foo_pyx.dat", x=1, y=2),
```

<sup>5</sup> Daher benötigt PyX auch ein installiertes TeX-System. Da es sich dabei um ein sehr mächtiges Textsatzsystem handelt, das im wissenschaftlichen Umfeld stark genutzt wird, lohnt sich eine Installation auch unabhängig von PyX. Für weitere Informationen siehe zum Beispiel <http://www.tug.org/texlive/> oder auch <http://www.dante.de/tex/tl-install-windows.html>.

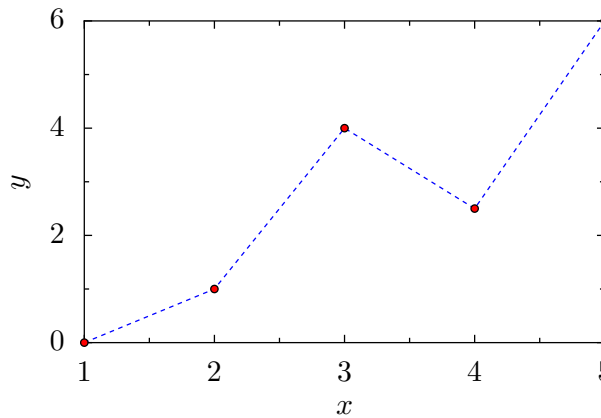


```

10     [graph.style.line([style.linestyle.dashed, color.rgb(0, 0, 1)]),
11     graph.style.symbol(graph.style.symbol.circle, size=0.1,
12                        symbolattrs=[deco.filled([color.rgb.red]),
13                                     deco.stroked([color.grey(0.5)])])])
14 g.writePDFfile("foo_pyx")

```

die folgende Grafik



Hier haben wir die folgenden Veränderungen vorgenommen. In Zeile 3 wurden alle Beschriftungen um einen Faktor 1,3 vergrößert. PyX erlaubt es, verschiedene Längen, zum Beispiel Textgrößen und Liniendicken unabhängig voneinander global zu verändern. `xscale` ist dabei für die Textgröße zuständig. In den Zeilen 6 und 7 wurden die beiden Achsen mit Beschriftungen versehen, wobei die Dollarzeichen durch `TeX` bedingt sind und ein Umschalten in den Mathematikmodus bewirken. Außerdem werden hier lineare Achsen verwendet. Mit `graph.axis.logarithmic` könnte man auch logarithmische Achsen verlangen. Zudem wäre es möglich, eigene Achsentypen zu programmieren.

In den Zeilen 9-12 wird die Darstellung der Daten festgelegt. In diesem Fall haben wir angegeben, dass wir die Datenpunkte sowohl durch Symbole darstellen als auch mit Linien verbinden wollen. Letzteres geschieht mit `graph.style.line`, das als Argument eine Liste von Linieneigenschaften erwartet. Hier haben wir einen gestrichelten Liniestil und die Farbe blau im RGB-Format<sup>6</sup> verlangt. Für die Symbole haben wir mit `graph.style.symbol.circle` Kreise ausgewählt, deren Größe durch den Wert des Arguments `size` bestimmt ist. Zudem kann eine Liste von Attributen übergeben werden. Wir verlangen beispielsweise, dass die Kreise rot gefüllt sind, wobei hier zur Abwechslung der Farbname verwendet wurde. Außerdem wird die Kontur des Kreises in grau ausgeführt, da `color.grey` mit einem Argument zwischen Null und Eins auf einen Grauwert zwischen schwarz und weiß abgebildet wird. Man sieht in diesem Programmbeispiel, dass man sich einiges an Tipparbeit durch geeignete `import`-Anweisungen zu Beginn des Programms sparen könnte.

Wir beenden dieses Kapitel mit einem Beispiel von der [PyX-Webseite](#), das die Mächtigkeit des Programmpakets bei Schemazeichnungen demonstriert.

```

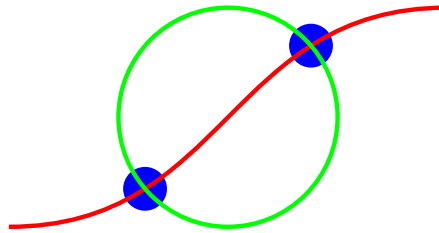
1  from pyx import *
2
3  p1 = path.curve(0, 0, 1, 0, 1, 1, 2, 1)
4  p2 = path.circle(1, 0.5, 0.5)
5
6  (a1, a2), (b1, b2) = p1.intersect(p2)
7
8  x1, y1 = p1.at(a1)
9  x2, y2 = p1.at(a2)
10
11 c = canvas.canvas()
12 c.fill(path.circle(x1, y1, 0.1), [color.rgb.blue])
13 c.fill(path.circle(x2, y2, 0.1), [color.rgb.blue])
14 c.stroke(p1, [color.rgb.red])
15 c.stroke(p2, [color.rgb.green])

```

<sup>6</sup> RGB steht für »red«, »green« und »blue«, wobei hier die Stärke jeder Komponente durch eine Zahl zwischen Null und Eins angegeben wird.

```
16 c.writePDFfile("intersect")
```

Das Ergebnis sieht folgendermaßen aus:



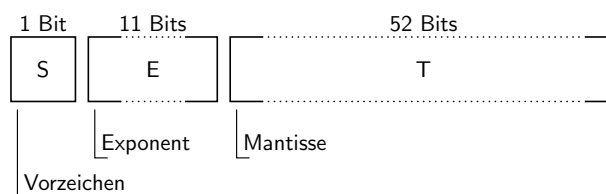
In den Zeilen 3 und 4 werden zunächst zwei Pfade definiert, und zwar eine Bézier-Kurve `p1` und ein Kreis `p2`. In Zeile 6 werden die beiden Pfade miteinander geschnitten. Das Ergebnis sind zwei Tupel, die die Lage der beiden Schnittpunkte entlang der beiden Kurven angeben. Die zugehörigen Koordinaten werden in den Zeilen 8 und 9 mit Hilfe der ersten Kurve bestimmt. In Zeile 11 beginnt das Zeichnen mit der Einrichtung eines »canvas«, also einer Leinwand, auf der gemalt werden kann. Dann werden in den Zeilen 12 und 13 an den zuvor berechneten Schnittpunkten zwei blau gefüllte Kreise mit Radius `0.1` positioniert. Schließlich werden die beiden Pfade in rot bzw. grün gezeichnet und das Ergebnis mit Hilfe der `writePDFfile`-Methode des Canvas ausgegeben.

Abschließend muss noch einmal betont werden, dass sowohl die Beschreibung von `matplotlib` als auch die Beschreibung von `PyX` nur jeweils einen winzigen Ausschnitt aus den Möglichkeiten der beiden Programmpakete darstellen konnten. Einen guten Überblick bieten die oben bereits erwähnten Beispielseiten.

## 64-Bit-Gleitkommazahlen nach IEEE-Standard 754

In diesem Anhang soll kurz die Binärdarstellung von 64-Bit-Gleitkommazahlen nach dem Standard IEEE Std 754–2008<sup>1</sup> vorgestellt werden. Dabei handelt es sich nur um einen sehr kleinen Ausschnitt dieses Standards, der noch weitere Gleitkommazahlenformate sowie deren arithmetische Verarbeitung definiert.

Wie in der folgenden Abbildung zu sehen ist, besteht eine 64-Bit-Gleitkommazahl aus drei Anteilen: 1 Bit für das Vorzeichen, 11 Bits aus denen der Exponent bestimmt wird, sowie 52 Bits aus denen sich die Mantisse ergibt. Im zweiten und dritten Segment steht das höchstwertigste Bit (MSB – *most significant bit*) jeweils links und das niederwertigste Bit (LSB – *least significant bit*) rechts.



Die Interpretation der 64 Bits hängt vom Wert des Exponenten E ab, der als Integer zu verstehen ist und damit Werte zwischen 0 und  $2^{11}-1$  annehmen kann.

Wir beginnen mit dem häufigsten Fall, dass E ungleich 0 und ungleich  $2^{11}-1$  ist, also einen Wert zwischen 1 und  $2^{11}-2$  besitzt. In diesem Fall liegt eine so genannte normalisierte Zahl vor, für die die 64 Bits in folgender Weise zu interpretieren sind. Ist das Vorzeichenbit gleich 0, so handelt es sich um eine positive Zahl, andernfalls um eine negative Zahl. Um den tatsächlichen Exponenten zu erhalten, muss man von E den Wert  $2^{10}-1=1023$  abziehen. Damit resultiert im Binärsystem ein Faktor, der zwischen  $2^{-1022}$  und  $2^{1023}$  liegt. Die Mantisse ist als Nachkommaanteil im Binärsystem zu interpretieren, wobei implizit eine führende 1 vor dem Komma steht.

Ein Beispiel soll diese Kodierung verdeutlichen. Gegeben sei die 64-Bit-Darstellung C039A40000000000, deren führende Bits `1|10000000011|100110100100...` lauten. Die senkrechten Striche verdeutlichen dabei die Einteilung in die drei Bereiche S, E und T. Das erste Bit gibt an, dass es sich um eine negative Zahl handelt. Die folgenden 11 Bit ergeben  $2^{10}+2^1+2^0 = 1027$ . Subtrahiert man 1023, so ergibt sich für die kodierte Zahl ein Faktor  $2^4$ . Die restlichen Bits ergeben die Mantisse mit dem Wert  $2^{-1}+2^{-4}+2^{-5}+2^{-7}+2^{-10} = 0,6025390625$ . Berücksichtigt man die führende implizite Eins bei normalisierten Zahlen, so ergibt sich insgesamt  $-2^4 \times 1,6025390625 = -25,640625$ .

Falls der Exponent E seinen Maximalwert, für 64-Bit-Gleitkommazahlen also 2047, annimmt, hängt die Interpretation vom Mantissenfeld ab. Enthält dieses den Wert Null, so ist die Zahl unter Berücksichtigung des Vorzeichenbits als  $+\infty$  oder  $-\infty$  zu interpretieren. Ist das Mantissenfeld dagegen ungleich Null, so ergibt sich der Wert NaN (Not a Number).

Hat der Exponent E seinen Minimalwert Null und ist zugleich die Mantisse T gleich Null, so liegt je nach Wert des Vorzeichenbits die Zahl 0 oder -0 vor. Ist dagegen die Mantisse T von Null verschieden, so hat man es mit einer so genannten denormalisierten Zahl zu tun. Denormalisierte Zahlen wurden eingeführt, weil der Minimalwert einer normalisierten 64-Bit-Gleitkommazahl betragsmäßig gleich  $2^{-1022} \approx 2,225 \cdot 10^{-308}$  ist. Zwischen dieser Zahl und Null existiert somit eine Lücke, die man im Rahmen der Möglichkeiten einer 64-Bit-Gleitkommazahl zu füllen versucht. Im Gegensatz zu normalisierten Zahlen, bei denen eine implizite Eins vor den Nachkommaanteil

<sup>1</sup> IEEE Standard for Floating-Point Arithmetic ([dx.doi.org:10.1109/IEEESTD.2008.4610935](https://doi.org/10.1109/IEEESTD.2008.4610935))

zu setzen war, steht bei denormalisierten Zahlen vor dem Komma eine Null. Da damit der sich aus dem Mantisenfeld ergebende Faktor kleiner als Eins ist, eröffnet sich die Möglichkeit Zahlen darzustellen, die kleiner als die kleinst mögliche normalisierte Zahl ist. Allerdings geht dies auf Kosten der Anzahl signifikanter Ziffern, die bei denormalisierten Zahlen kleiner ist als bei normalisierten Zahlen. Damit die denormalisierten Zahlen nahtlos an die normalisierten Zahlen anschließen, ist der sich normalerweise aus dem Exponentenfeld ergebende Faktor gleich dem kleinsten Faktor für normalisierte Zahlen, also  $2^{1-1023} = 2^{-1022}$ .

Wir charakterisieren abschließend die 64-Bit-Gleitkommazahlen durch einige ihrer Eigenschaften. Die kleinste darstellbare Zahl größer als Null ist  $2^{-1022} \times 2^{-52} = 2^{-1074} \approx 4,941 \cdot 10^{-324}$ . Die größte darstellbare endliche Zahl ist  $2^{1023} \times (1+1-2^{-52}) \approx 1,798 \cdot 10^{308}$ . Der Abstand zwischen der Zahl Eins und der nächst größeren darstellbaren Zahl beträgt  $2^{-52} \approx 2,22 \cdot 10^{-16}$ . Die letzten beiden Angaben lassen sich in Python folgendermaßen verifizieren:

```
>>> import sys
>>> sys.float_info.max
1.7976931348623157e+308
>>> sys.float_info.epsilon
2.220446049250313e-16
```

Mit `sys.float_info.min` erhält man nicht die kleinste überhaupt darstellbare Zahl, sondern die kleinste darstellbare normalisierte Zahl  $2^{-1022}$ .

---

## Unicode

---

Auch Schriftzeichen müssen im Computer binär kodiert werden.<sup>1</sup> Bereits 1967 wurde der *American Standard Code for Information Interchange* (ASCII) als Standard veröffentlicht, wobei aber auch andere Kodierungen insbesondere auf Großrechnern im Gebrauch waren. Der ASCII ist ein 7-Bit-Code und kann somit 128 Zeichen darstellen, die auf der ersten der Codetabellen gezeigt ist, die auf den folgenden Seiten dargestellt sind. Die grau unterlegten Akronyme stellen Steuerzeichen dar, wobei SP ein Leerzeichen bezeichnet und damit auch zu den druckbaren Zeichen gerechnet werden kann. Zu den heute noch wichtigen Steuerzeichen zählen:

Hexcode	Kürzel	Bedeutung	Tastenkombination	Escape-Sequenz
08	BS	backspace	STRG-H	\b
09	HT	horizontal tabulation	STRG-I	\t
0A	LF	line feed (neue Zeile)	STRG-J	\n
0C	FF	form feed (neue Seite)	STRG-L	\f
0D	CR	carriage return (Wagenrücklauf)	STRG-M	\r
1B	ESC	escape	STRG-[	

Bei der Betrachtung der ASCII-Codetabelle fällt sofort das Fehlen von Umlauten auf, die für deutsche Texte benötigt werden, von anderen Schriftsystemen ganz zu schweigen. In einem ersten Schritt liegt daher eine Erweiterung auf einen 8-Bit-Code nahe. Hierfür existieren eine ganze Reihe von Belegungen, die sich am Bedarf bestimmter Sprachen orientieren. Von Bedeutung ist unter anderem die Normenfamilie ISO 8859. Für die deutsche Sprache ist ISO-8859-1 gebräuchlich, der in der zweiten der folgenden Codetabellen dargestellt ist und auch als »Latin-1« bekannt ist. Auch hier sind wieder eine Reihe von Steuerzeichen vertreten, unter anderem ein *nonbreakable space* (NBSP) und ein *soft hyphen* (SHY). Der zu den Zeichen gehörige Code ergibt sich aus den letzten beiden Stellen des Unicode Codepoints, der an dem vorangestellten U+ erkenntlich ist. Beispielsweise wird das »ä« durch 0xEA kodiert. Da das Eurozeichen in ISO-8859-1 nicht enthalten ist, ist auch die Norm ISO-8859-15 von Bedeutung, die sich an 8 Stellen von ISO-8859-1 unterscheidet.

In neuerer Zeit hat der Unicode-Standard<sup>2</sup> enorm an Bedeutung gewonnen, da er das Ziel hat, alle weltweit gebräuchlichen Zeichen zu kodieren. Die aktuelle Version 6.2<sup>3</sup> umfasst 110117 Zeichen und bietet noch genügend Platz für Erweiterungen. Jedes Zeichen ist einem so genannten Codepoint zugeordnet, der Werte zwischen 0x00 und 0x10FFFF annehmen kann. Da damit jedes Zeichen statt üblicherweise einem Byte nun drei Byte benötigen würde, werden die Unicode-Zeichen geschickt kodiert. Für westliche Sprachen ist die Kodierung UTF-8 besonders geeignet, da die ASCII-Zeichen im Bereich 0x00 bis 0x7F ihre Bedeutung beibehalten. Wir beschränken uns daher im Folgenden auf die Erklärung dieser Kodierung.

Die UTF-8-Kodierung ist in den folgenden Codetabellen unter dem entsprechenden Unicode Codepoint angegeben. Dabei kommen in diesen Beispielen 1-, 2- und 3-Byte-Werte vor. Im Allgemeinen können sogar 4 Bytes auftreten.

Im Bereich 0x00 bis 0x7F wird das letzte Byte des Codepoints verwendet und auf diese Weise Übereinstimmung mit ASCII erreicht. Somit lassen sich in einer westlichen Sprache verfasste Texte weitestgehend unabhängig von der tatsächlichen Kodierung lesen. Außerdem wird die Mehrzahl der vorkommenden Zeichen platzsparend kodiert.

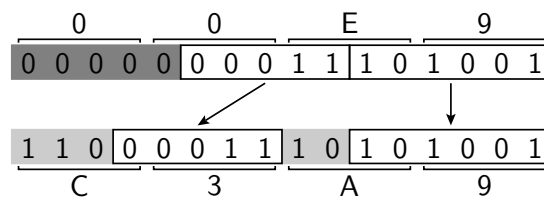
---

<sup>1</sup> Eine ausführlichere Darstellung der Entwicklungsgeschichte von Zeichenkodierungen gibt Y. Haralambous, *Fonts & Encodings* (O'Reilly, 2007).

<sup>2</sup> [www.unicode.org](http://www.unicode.org)

<sup>3</sup> [www.unicode.org/versions/Unicode6.2.0/](http://www.unicode.org/versions/Unicode6.2.0/)

Ist das führende Bit eine 1, so handelt es sich um einen Mehrbytecode. Im Bereich  $0x0080$  bis  $0x07FF$  werden zwei Bytes zur Kodierung verwendet. Die elf relevanten Bytes werden dabei wie in folgendem Beispiel gezeigt auf die zwei Bytes verteilt:

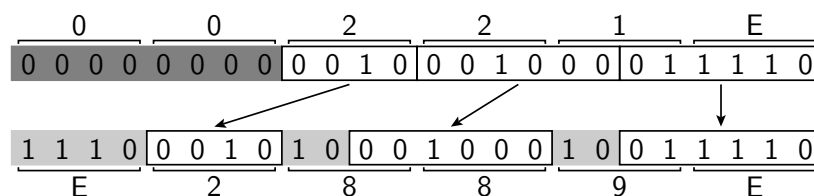


Dabei wird der Codepoint  $0xE9$  des Zeichens »é« auf den UTF-8-Code  $0xC3A9$  abgebildet.

**?** Was würde ein Programm, das von einer Latin-1-Kodierung ausgeht, in diesem Fall anzeigen?

Beispiele von 2-Byte-Codes sind in der zweiten und dritten der im Folgenden abgebildeten Codetabellen zu sehen. Dabei handelt es sich zum einen um die oberen 128 Zeichen der ISO-8859-1-Norm und zum anderen um griechische und koptische Zeichen im Unicode-Standard.

Die in der vierten Codetabelle gezeigten mathematischen Symbole erfordern einen 3-Byte-Code, der sich wie im folgenden Beispiel für das Zeichen »∞« gezeigt aus dem Unicode Codepoint ergibt:



Aus 4 Bytes bestehende Codes ergeben sich durch entsprechende Verallgemeinerung für Codepoints zwischen  $0x010000$  und  $0x10FFFF$ , wobei der UTF-8-Code dann mit  $0xF$  beginnt.

U+0000	U+0010	U+0020	U+0030	U+0040	U+0050	U+0060	U+0070
00	10	20	30	40	50	60	70
NUL	DLE	SP	0	@	P	‘	p
U+0001	U+0011	U+0021	U+0031	U+0041	U+0051	U+0061	U+0071
01	11	21	31	41	51	61	71
SOH	DC1	!	1	A	Q	a	q
U+0002	U+0012	U+0022	U+0032	U+0042	U+0052	U+0062	U+0072
02	12	22	32	42	52	62	72
STX	DC2	”	2	B	R	b	r
U+0003	U+0013	U+0023	U+0033	U+0043	U+0053	U+0063	U+0073
03	13	23	33	43	53	63	73
ETX	DC3	#	3	C	S	c	s
U+0004	U+0014	U+0024	U+0034	U+0044	U+0054	U+0064	U+0074
04	14	24	34	44	54	64	74
EOT	DC4	\$	4	D	T	d	t
U+0005	U+0015	U+0025	U+0035	U+0045	U+0055	U+0065	U+0075
05	15	25	35	45	55	65	75
ENQ	NAK	%	5	E	U	e	u
U+0006	U+0016	U+0026	U+0036	U+0046	U+0056	U+0066	U+0076
06	16	26	36	46	56	66	76
ACK	SYN	&	6	F	V	f	v
U+0007	U+0017	U+0027	U+0037	U+0047	U+0057	U+0067	U+0077
07	17	27	37	47	57	67	77
BEL	ETB	,	7	G	W	g	w
U+0008	U+0018	U+0028	U+0038	U+0048	U+0058	U+0068	U+0078
08	18	28	38	48	58	68	78
BS	CAN	(	8	H	X	h	x
U+0009	U+0019	U+0029	U+0039	U+0049	U+0059	U+0069	U+0079
09	19	29	39	49	59	69	79
HT	EM	)	9	I	Y	i	y
U+000A	U+001A	U+002A	U+003A	U+004A	U+005A	U+006A	U+007A
0A	1A	2A	3A	4A	5A	6A	7A
LF	SUB	*	:	J	Z	j	z
U+000B	U+001B	U+002B	U+003B	U+004B	U+005B	U+006B	U+007B
0B	1B	2B	3B	4B	5B	6B	7B
VT	ESC	+	;	K	[	k	{
U+000C	U+001C	U+002C	U+003C	U+004C	U+005C	U+006C	U+007C
0C	1C	2C	3C	4C	5C	6C	7C
FF	FS	,	<	L	\	l	
U+000D	U+001D	U+002D	U+003D	U+004D	U+005D	U+006D	U+007D
0D	1D	2D	3D	4D	5D	6D	7D
CR	GS	-	=	M	]	m	}
U+000E	U+001E	U+002E	U+003E	U+004E	U+005E	U+006E	U+007E
0E	1E	2E	3E	4E	5E	6E	7E
SO	RS	.	>	N	^	n	~
U+000F	U+001F	U+002F	U+003F	U+004F	U+005F	U+006F	U+007F
0F	1F	2F	3F	4F	5F	6F	7F
SI	US	/	?	O	_	o	DEL

U+0080 C280	U+0090 C290	U+00A0 C2A0	U+00B0 C2B0	U+00C0 C380	U+00D0 C390	U+00E0 C3A0	U+00F0 C3B0
XXX	DCS	NBSP	°	À	Ð	à	ð
U+0081 C281	U+0091 C291	U+00A1 C2A1	U+00B1 C2B1	U+00C1 C381	U+00D1 C391	U+00E1 C3A1	U+00F1 C3B1
XXX	PU1	¡	±	Á	Ñ	á	ñ
U+0082 C282	U+0092 C292	U+00A2 C2A2	U+00B2 C2B2	U+00C2 C382	U+00D2 C392	U+00E2 C3A2	U+00F2 C3B2
BPH	PU2	¢	²	Â	Ò	â	ò
U+0083 C283	U+0093 C293	U+00A3 C2A3	U+00B3 C2B3	U+00C3 C383	U+00D3 C393	U+00E3 C3A3	U+00F3 C3B3
NBH	STS	£	³	Ã	Ó	ã	ó
U+0084 C284	U+0094 C294	U+00A4 C2A4	U+00B4 C2B4	U+00C4 C384	U+00D4 C394	U+00E4 C3A4	U+00F4 C3B4
IND	CCH	¤	´	Ä	Ô	ä	ô
U+0085 C285	U+0095 C295	U+00A5 C2A5	U+00B5 C2B5	U+00C5 C385	U+00D5 C395	U+00E5 C3A5	U+00F5 C3B5
NEL	MW	¥	µ	Å	Õ	å	õ
U+0086 C286	U+0096 C296	U+00A6 C2A6	U+00B6 C2B6	U+00C6 C386	U+00D6 C396	U+00E6 C3A6	U+00F6 C3B6
SSA	SPA	¦	¶	Æ	Ö	æ	ö
U+0087 C287	U+0097 C297	U+00A7 C2A7	U+00B7 C2B7	U+00C7 C387	U+00D7 C397	U+00E7 C3A7	U+00F7 C3B7
ESA	EPA	§	·	Ç	×	ç	÷
U+0088 C288	U+0098 C298	U+00A8 C2A8	U+00B8 C2B8	U+00C8 C388	U+00D8 C398	U+00E8 C3A8	U+00F8 C3B8
HTS	SOS	¨	¸	È	Ø	è	ø
U+0089 C289	U+0099 C299	U+00A9 C2A9	U+00B9 C2B9	U+00C9 C389	U+00D9 C399	U+00E9 C3A9	U+00F9 C3B9
HTJ	XXX	©	¹	É	Ù	é	ù
U+008A C28A	U+009A C29A	U+00AA C2AA	U+00BA C2BA	U+00CA C38A	U+00DA C39A	U+00EA C3AA	U+00FA C3BA
VTS	SCI	ª	º	Ê	Ú	ê	ú
U+008B C28B	U+009B C29B	U+00AB C2AB	U+00BB C2BB	U+00CB C38B	U+00DB C39B	U+00EB C3AB	U+00FB C3BB
PLD	CSI	«	»	Ë	Û	ë	û
U+008C C28C	U+009C C29C	U+00AC C2AC	U+00BC C2BC	U+00CC C38C	U+00DC C39C	U+00EC C3AC	U+00FC C3BC
PLU	ST	¬	¼	Ì	Ü	ì	ü
U+008D C28D	U+009D C29D	U+00AD C2AD	U+00BD C2BD	U+00CD C38D	U+00DD C39D	U+00ED C3AD	U+00FD C3BD
RI	OSC	SHY	½	Í	Ý	í	ý
U+008E C28E	U+009E C29E	U+00AE C2AE	U+00BE C2BE	U+00CE C38E	U+00DE C39E	U+00EE C3AE	U+00FE C3BE
SS2	PM	®	¾	Î	Þ	î	þ
U+008F C28F	U+009F C29F	U+00AF C2AF	U+00BF C2BF	U+00CF C38F	U+00DF C39F	U+00EF C3AF	U+00FF C3BF
SS3	APC	-	¿	Ï	ß	ï	ÿ



U+0380 CE80	U+0390 CE90	U+03A0 CEA0	U+03B0 CEB0	U+03C0 CF80	U+03D0 CF90	U+03E0 CFA0	U+03F0 CFB0
̀	́	͂	̓	̈́	ͅ	͆	͇
U+0381 CE81	U+0391 CE91	U+03A1 CEA1	U+03B1 CEB1	U+03C1 CF81	U+03D1 CF91	U+03E1 CFA1	U+03F1 CFB1
͈	͉	͊	͋	͌	͍	͎	͏
U+0382 CE82	U+0392 CE92	U+03A2 CEA2	U+03B2 CEB2	U+03C2 CF82	U+03D2 CF92	U+03E2 CFA2	U+03F2 CFB2
͐	͑	͒	͓	͔	͕	͖	͗
U+0383 CE83	U+0393 CE93	U+03A3 CEA3	U+03B3 CEB3	U+03C3 CF83	U+03D3 CF93	U+03E3 CFA3	U+03F3 CFB3
͘	͙	͚	͛	͜	͝	͞	͟
U+0384 CE84	U+0394 CE94	U+03A4 CEA4	U+03B4 CEB4	U+03C4 CF84	U+03D4 CF94	U+03E4 CFA4	U+03F4 CFB4
͠	͡	͢	ͣ	ͤ	ͥ	ͦ	ͧ
U+0385 CE85	U+0395 CE95	U+03A5 CEA5	U+03B5 CEB5	U+03C5 CF85	U+03D5 CF95	U+03E5 CFA5	U+03F5 CFB5
ͨ	ͩ	ͪ	ͫ	ͬ	ͭ	ͮ	ͯ
U+0386 CE86	U+0396 CE96	U+03A6 CEA6	U+03B6 CEB6	U+03C6 CF86	U+03D6 CF96	U+03E6 CFA6	U+03F6 CFB6
Ͱ	ͱ	Ͳ	ͳ	ʹ	͵	Ͷ	ͷ
U+0387 CE87	U+0397 CE97	U+03A7 CEA7	U+03B7 CEB7	U+03C7 CF87	U+03D7 CF97	U+03E7 CFA7	U+03F7 CFB7
͸	͹	ͺ	ͻ	ͼ	ͽ	Ϳ	̀
U+0388 CE88	U+0398 CE98	U+03A8 CEA8	U+03B8 CEB8	U+03C8 CF88	U+03D8 CF98	U+03E8 CFA8	U+03F8 CFB8
́	͂	̓	̈́	ͅ	͆	͇	͈
U+0389 CE89	U+0399 CE99	U+03A9 CEA9	U+03B9 CEB9	U+03C9 CF89	U+03D9 CF99	U+03E9 CFA9	U+03F9 CFB9
͉	͊	͋	͌	͍	͎	͏	͐
U+038A CE8A	U+039A CE9A	U+03AA CEAA	U+03BA CEBA	U+03CA CF8A	U+03DA CF9A	U+03EA CFAA	U+03FA CFBA
͑	͒	͓	͔	͕	͖	͗	͘
U+038B CE8B	U+039B CE9B	U+03AB CEAB	U+03BB CEBB	U+03CB CF8B	U+03DB CF9B	U+03EB CFAB	U+03FB CFBB
͙	͚	͛	͜	͝	͞	͟	͠
U+038C CE8C	U+039C CE9C	U+03AC CEAC	U+03BC CEBC	U+03CC CF8C	U+03DC CF9C	U+03EC CFAC	U+03FC CFBC
͡	͢	ͣ	ͤ	ͥ	ͦ	ͧ	ͨ
U+038D CE8D	U+039D CE9D	U+03AD CEAD	U+03BD CEBD	U+03CD CF8D	U+03DD CF9D	U+03ED CFAD	U+03FD CFBD
ͩ	ͪ	ͫ	ͬ	ͭ	ͮ	ͯ	Ͱ
U+038E CE8E	U+039E CE9E	U+03AE CEAE	U+03BE CEBE	U+03CE CF8E	U+03DE CF9E	U+03EE CFAE	U+03FE CFBE
ͱ	Ͳ	ͳ	ʹ	͵	Ͷ	ͷ	͸
U+038F CE8F	U+039F CE9F	U+03AF CEAF	U+03BF CEBF	U+03CF CF8F	U+03DF CF9F	U+03EF CFAF	U+03FF CFBF
͹	ͺ	ͻ	ͼ	ͽ	Ϳ	̀	́

U+2200 E28880	U+2210 E28890	U+2220 E288A0	U+2230 E288B0	U+2240 E28980	U+2250 E28990	U+2260 E289A0	U+2270 E289B0
∇	Π	∠	ℳ	ℓ	≡	≠	≠
U+2201 E28881	U+2211 E28891	U+2221 E288A1	U+2231 E288B1	U+2241 E28981	U+2251 E28991	U+2261 E289A1	U+2271 E289B1
℄	Σ	∠	ℳ	ℓ	≡	≡	≠
U+2202 E28882	U+2212 E28892	U+2222 E288A2	U+2232 E288B2	U+2242 E28982	U+2252 E28992	U+2262 E289A2	U+2272 E289B2
∂	−	∠	ℳ	ℓ	≡	≠	≤
U+2203 E28883	U+2213 E28893	U+2223 E288A3	U+2233 E288B3	U+2243 E28983	U+2253 E28993	U+2263 E289A3	U+2273 E289B3
∃	∓		ℳ	ℓ	≡	≡	≥
U+2204 E28884	U+2214 E28894	U+2224 E288A4	U+2234 E288B4	U+2244 E28984	U+2254 E28994	U+2264 E289A4	U+2274 E289B4
ℳ	+	+	∴	≠	:=	≤	≠
U+2205 E28885	U+2215 E28895	U+2225 E288A5	U+2235 E288B5	U+2245 E28985	U+2255 E28995	U+2265 E289A5	U+2275 E289B5
∅	/		∴	≡	=:	≥	≠
U+2206 E28886	U+2216 E28896	U+2226 E288A6	U+2236 E288B6	U+2246 E28986	U+2256 E28996	U+2266 E289A6	U+2276 E289B6
Δ	\	ℳ	:	≠	≡	≤	≤
U+2207 E28887	U+2217 E28897	U+2227 E288A7	U+2237 E288B7	U+2247 E28987	U+2257 E28997	U+2267 E289A7	U+2277 E289B7
∇	*	∧	∴	≠	≡	≥	≥
U+2208 E28888	U+2218 E28898	U+2228 E288A8	U+2238 E288B8	U+2248 E28988	U+2258 E28998	U+2268 E289A8	U+2278 E289B8
∈	◦	∨	÷	≈	≡	≤	\$
U+2209 E28889	U+2219 E28899	U+2229 E288A9	U+2239 E288B9	U+2249 E28989	U+2259 E28999	U+2269 E289A9	U+2279 E289B9
∉	•	∩	−:	≠	≡	≥	≠
U+220A E2888A	U+221A E2889A	U+222A E288AA	U+223A E288BA	U+224A E2898A	U+225A E2899A	U+226A E289AA	U+227A E289BA
€	√	U	∴	≈	≡	≪	<
U+220B E2888B	U+221B E2889B	U+222B E288AB	U+223B E288BB	U+224B E2898B	U+225B E2899B	U+226B E289AB	U+227B E289BB
∋	∛	∫	≈	≈	≡	≫	>
U+220C E2888C	U+221C E2889C	U+222C E288AC	U+223C E288BC	U+224C E2898C	U+225C E2899C	U+226C E289AC	U+227C E289BC
∅	∛	∫	~	≡	≡	∅	≤
U+220D E2888D	U+221D E2889D	U+222D E288AD	U+223D E288BD	U+224D E2898D	U+225D E2899D	U+226D E289AD	U+227D E289BD
∋	α	∫	~	×	≡	*	≥
U+220E E2888E	U+221E E2889E	U+222E E288AE	U+223E E288BE	U+224E E2898E	U+225E E2899E	U+226E E289AE	U+227E E289BE
■	∞	ℳ	∞	≈	≡	≠	≤
U+220F E2888F	U+221F E2889F	U+222F E288AF	U+223F E288BF	U+224F E2898F	U+225F E2899F	U+226F E289AF	U+227F E289BF
Π	⊥	ℳ	~	≈	≡	≠	≥