Python für Naturwissenschaftler

Release 2017beta

Gert-Ludwig Ingold

19.06.2017

Inhaltsverzeichnis

1 Einführung **1**
- 1.1 Verwendete Symbole . . . 1
- 1.2 Danke an 2

2 Fortgeschrittene Aspekte von Python **3**
- 2.1 Sets . . . 3
- 2.2 Das `collections`-Modul . . . 6
- 2.3 List comprehensions . . . 9
- 2.4 Generatoren und Iteratoren . . . 13
- 2.5 Dekoratoren . . . 16
- 2.6 Ausnahmen . . . 20
- 2.7 Kontext mit `with`-Anweisung . . . 24

3 NumPy **27**
- 3.1 Python-Listen und Matrizen . . . 27
- 3.2 NumPy-Arrays . . . 28
- 3.3 Erzeugung von NumPy-Arrays . . . 33
- 3.4 Adressierung von NumPy-Arrays . . . 38
- 3.5 Universelle Funktionen . . . 45
- 3.6 Lineare Algebra . . . 48
- 3.7 Einfache Anwendungen . . . 50

4 Erstellung von Grafiken **55**
- 4.1 Erstellung von Grafiken mit matplotlib . . . 55
- 4.2 Erstellung von Grafiken mit PyX . . . 74

5 Versionskontrolle mit Git **95**
- 5.1 Vorbemerkungen . . . 95
- 5.2 Grundlegende Arbeitsschritte . . . 95
- 5.3 Verzweigen und Zusammenführen . . . 100
- 5.4 Umgang mit entfernten Archiven . . . 105

6 Testen von Programmen **109**
- 6.1 Wozu braucht man Tests? . . . 109
- 6.2 Das `doctest`-Modul . . . 110
- 6.3 Das `unittest`-Modul . . . 115
- 6.4 Testen mit NumPy . . . 122

7 Laufzeituntersuchungen **123**
- 7.1 Allgemeine Vorbemerkungen . . . 123
- 7.2 Fallstricke bei der Laufzeitmessung . . . 124

7.3 Das Modul `timeit` . . . 126
7.4 Das Modul `cProfile` . . . 128
7.5 Zeilenorientierte Laufzeitbestimmung . . . 130

8 Aspekte des parallelen Rechnens **135**
8.1 Threads, Prozesse und der GIL . . . 135
8.2 Parallelverarbeitung in Python . . . 136
8.3 Numba . . . 142

KAPITEL 1

Einführung

Die Vorlesung »Python für Naturwissenschaftler« baut auf der Vorlesung »Einführung in das Programmieren für Physiker und Naturwissenschaftler«, im Folgenden kurz Einführungsvorlesung genannt, auf. Sie setzt daher Kenntnisse der Programmiersprache Python, wie Sie in der Einführungsvorlesung vermittelt werden, voraus. Gegebenenfalls wird empfohlen, sich Grundkenntnisse von Python mit Hilfe des Manuskripts zur Einführungsvorlesung anzueignen oder das Manuskript zu verwenden, um Kenntnisse aufzufrischen.

Die Einführungsvorlesung beschränkt sich bewusst weitestgehend auf Sprachelemente von Python, die in ähnlicher Form auch in anderen, für den Naturwissenschaftler wichtigen Programmiersprachen, wie C oder Fortran, verfügbar sind. Diese Beschränkung wird in der Vorlesung »Python für Naturwissenschaftler« fallengelassen, so dass einige weitere wichtige Sprachelemente besprochen werden können. Dabei wird jedoch keine Vollständigkeit angestrebt. Vielmehr soll auch Raum bleiben, um erstens eine etwas detailliertere Einführung in die zentrale numerische Bibliothek in Python, nämlich NumPy, zu geben und zweitens einige für die Codeentwicklung relevante Werkzeuge zu besprechen.

Aus Zeitgründen werden wir uns bei Letzteren auf Versionskontrollsysteme, auf Verfahren zum systematischen und nachvollziehbaren Testen von Programmen und auf Verfahren zur Bestimmung der Laufzeiten verschiedener Programmteile beschränken. Einige dieser Techniken werden wir zwar konkret im Zusammenspiel mit Python kennenlernen, sie sind aber auch auf andere Programmiersprachen übertragbar. Auch wenn solche Techniken häufig als unnötiger Aufwand empfunden werden, können sie wesentlich zur Qualität wissenschaftlichen Rechnens beitragen.[1]

Die im Manuskript gezeigten Code-Beispiele sind für die Verwendung mit einer aktuellen Version von Python 3 vorgesehen. Ein Großteil der Beispiele ist aber auch unter Python 2.7 lauffähig oder lässt sich durch kleinere Anpassungen lauffähig machen.

1.1 Verwendete Symbole

`In [1]:` stellt den Prompt des IPython-Interpreters dar, wobei statt der `1` auch eine andere Eingabenummer stehen kann.

`Out[1]:` weist auf die Ausgabe des IPython-Interpreters zur Eingabe `In [1]:` hin.

`...:` wird im IPython-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 zuvorige Eingabe von Text verwendet.

[1] arXiv:1210.0530

$ steht für den Prompt, also die Eingabeaufforderung, der Shell beim kommandozeilenbasierten Arbeiten in einem Terminalfenster.

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

1.2 Danke an . . .

- . . . die Hörerinnen und Hörer der Vorlesung „Python für Naturwissenschaftler“, deren Fragen und Anregungen in diesem Manuskript ihren Niederschlag fanden;
- Michael Hartmann, Oliver Kanschat-Krebs und Benjamin Spreng für eine Reihe von Kommentaren zu diesem Manuskript.

KAPITEL 2

Fortgeschrittene Aspekte von Python

In diesem Kapitel sollen einige Sprachelemente von Python besprochen werden, auf die in der Vorlesung »Einführung in das Programmieren für Physiker und Materialwissenschaftler« nicht oder nur nur sehr kurz eingegangen wurde. Aus Platz- und Zeitgründen muss allerdings auch hier eine Auswahl getroffen werden.

2.1 Sets

In der Vorlesung »Einführung in das Programmieren für Physiker und Materialwissenschaftler« hatten wir uns im Kapitel über zusammengesetzte Datentypen vor allem mit Listen, Tupeln, Zeichenketten und Dictionaries beschäftigt. Sets wurden dagegen nur kurz erwähnt und sollen hier etwas ausführlicher besprochen werden.

Ein Set ist eine Menge von Python-Objekten, denen ein Hashwert zugeordnet werden kann. Insofern ist es mit einem Dictionary vergleichbar, das nur Schlüssel, aber nicht die zugehörigen Werte enthält. Die Einträge eines Sets können nicht mehrfach auftreten, so dass die Bildung eines Sets geeignet ist, um aus einer Liste Duplikate zu entfernen. Dies wird im Folgenden demonstriert.

```
In [1]: list_pts = [(0, 0), (-1, 2), (0, 0), (1, 2), (-1, 2), (0,0)]

In [2]: set_pts = set(list_pts)

In [3]: set_pts
Out[3]: {(-1, 2), (0, 0), (1, 2)}

In [4]: uniq_list_pts = list(set_pts)

In [5]: uniq_list_pts
Out[5]: [(1, 2), (0, 0), (-1, 2)]
```

Zunächst wird eine Liste erstellt, die hier Tupel enthält, um beispielsweise Punkte in der Ebene zu beschreiben. In der Eingabe 2 wird ein Set erstellt, in dem, wie man in der Ausgabe 3 sieht, tatsächlich keine Duplikate mehr vorkommen. Dabei liegen die Elemente im Set nicht in einer bestimmten Ordnung vor, ganz so wie wir es von Dictionaries kennen. Bei Bedarf kann man das Set auch wieder in eine Liste umwandeln, wie die Eingabe 4 und die Ausgabe 5 zeigen.

Statt wie im vorigen Beispiel ein Set durch Umwandlung aus einer Liste zu erzeugen, kann man die Elemente des Sets auch direkt in einer Notation mit geschweiften Klammern, die an die Verwandtschaft mit Dictionaries erinnert, eingeben.

```
In [1]: set_of_ints = {1, 3, 2, 5, 2, 1, 3, 4}

In [2]: set_of_ints
Out[2]: {1, 2, 3, 4, 5}
```

Auch hier werden natürlich eventuelle Dubletten entfernt.

Ähnlich wie Listen oder Dictionaries sind Sets auch veränderbar (*mutable*) und damit selbst nicht als Elemente von Sets oder als Schlüssel von Dictionaries verwendbar. Dafür kann man Elemente hinzufügen oder entfernen, wobei der Versuch, ein nicht vorhandenes Element zu entfernen, eine `KeyError`-Ausnahme auslöst.

```
In [1]: data = {1, 2, 4}

In [2]: data.add(3)

In [3]: data
Out[3]: {1, 2, 3, 4}

In [4]: data.remove(1)

In [5]: data
Out[5]: {2, 3, 4}

In [6]: data.remove(10)
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-13-6610e4562113> in <module>()
----> 1 data.remove(10)

KeyError: 10
```

Will man ein Set als Schlüssel verwenden und ist man dafür bereit, auf die gerade beschriebenen Möglichkeiten, ein Set zu verändern, zu verzichten, so greift man auf das `frozenset` zurück, das wie der Name schon andeutet unveränderlich (*immutable*) ist.

```
In [1]: evens = frozenset([2, 4, 6, 8])

In [2]: evens.add(10)
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-15-2c352b4e8a10> in <module>()
----> 1 evens.add(10)

AttributeError: 'frozenset' object has no attribute 'add'

In [3]: odds = frozenset([1, 3, 5, 7])

In [4]: numbers = {evens: "some even numbers", odds: "some odd numbers"}

In [5]: numbers.keys()
Out[5]: dict_keys([frozenset({8, 2, 4, 6}), frozenset({1, 3, 5, 7})])
```

Um zu überprüfen, ob ein Objekt Element einer Menge ist, ist es günstig, statt einer Liste ein Set zu verwenden, wie die folgenden Tests zeigen.[1]

```
In [1]: nmax = 1000000

In [2]: xlist = list(range(nmax))
```

[1] Hier verwenden wir `%timeit`, eine der so genannten magischen Funktionen der verbesserten Python-Shell `IPython`, die es erlaubt, Ausführungszeiten einzelner Befehle zu bestimmen. Will man die Ausführungszeit eines ganzen Befehlsblocks bestimmen, so muss die magische `%%timeit`-Funktion mit zwei Prozentzeichen verwendet werden. Wir werden hierauf im Abschnitt *Das Modul timeit* zurückkommen.

```
In [3]: xset = set(xlist)

In [4]: %timeit 1 in xlist
10000000 loops, best of 3: 37 ns per loop

In [5]: %timeit 1 in xset
10000000 loops, best of 3: 32.1 ns per loop

In [6]: %timeit nmax-1 in xlist
100 loops, best of 3: 10.6 ms per loop

In [7]: %timeit nmax-1 in xset
10000000 loops, best of 3: 85.4 ns per loop
```

Hier liegen eine Liste und ein Set mit einer Million Elementen vor. Prüft man auf Mitgliedschaft eines der ersten Listenelemente ab, so gibt es keinen wesentlichen Unterschied zwischen Liste und Set. Ganz anders sieht es aus, wenn man ein Element vom Ende der Liste auswählt. In diesem Fall muss die ganze Liste durchsucht werden und die Ausführungszeit ist in unserem Beispiel mehr als hunderttausendmal langsamer als für das Set. Dieser Unterschied ist vor allem auch dann relevant, wenn das gewählte Element nicht vorhanden ist, so dass auf jeden Fall die gesamte Liste durchsucht werden muss. Natürlich ist die Erzeugung eines Sets mit einigem Zeitaufwand verbunden. Muss man aber häufig auf Mitgliedschaft in einer bestimmten Liste testen, so kann die Umwandlung in ein Set die Ausführung entscheidend beschleunigen.

Neben dem Test auf Mitgliedschaft lässt ein Set auch noch eine Reihe von Operationen auf Mengen zu, wie zum Beispiel das Vereinigen zweier Mengen (`union` oder `|`), das Bilden der Schnittmenge (`intersection` oder `&`) und deren Komplement (`symmetric_difference` oder `^`) sowie das Bilden der Differenzmenge (`difference` oder `-`). Zudem lässt sich auf Unter- und Obermenge (`issubset` bzw. `issuperset`) sowie Schnittmengenfreiheit (`isdisjoint`) testen. Diese Möglichkeiten sind im Folgenden illustriert.

```
In [1]: a = set([1, 2, 3])

In [2]: b = set([4, 5, 6])

In [3]: a.union(b)
Out[3]: {1, 2, 3, 4, 5, 6}

In [4]: c = set([1, 3, 6])

In [5]: a.intersection(c)
Out[5]: {1, 3}

In [6]: a.symmetric_difference(c)
Out[6]: {2, 6}

In [7]: a.difference(c)
Out[7]: {2}

In [8]: d = set([1, 3])

In [9]: a.issuperset(d)
Out[9]: True

In [10]: a.issubset(d)
Out[10]: False

In [11]: a.isdisjoint(b)
Out[11]: True
```

2.2 Das `collections`-Modul

Die Standardbibliothek von Python stellt im `collections`-Modul einige interessante Container-Datentypen zur Verfügung, die es erlauben, Probleme zu lösen, die gelegentlich mit Listen, Tupeln oder Dictionaries auftreten. Im Folgenden soll eine Auswahl dieser Datentypen kurz vorgestellt werden.

Wir beginnen mit den Tupeln, deren einzelne Elemente mit Hilfe von Integern angesprochen werden können. Wenn die einzelnen Elemente eine spezielle Bedeutung haben, ist jedoch die Zuordnung zu den Indizes nicht immer offensichtlich.

Als Beispiel betrachten wir Farben, die im RGB-System durch ein Tupel von drei Integern mit Werten zwischen 0 und 255 dargestellt werden können. Eine bestimmte Farbe könnte also folgendermaßen durch ein Tupel repräsentiert sein:

```
In [1]: farbe = (135, 206, 235)

In [2]: farbe[1]
Out[2]: 206

In [3]: r, g, b = farbe

In [4]: g
Out[4]: 206
```

Hierbei muss man wissen, dass das Element mit Index 1 dem Grünwert entspricht. Um den Code verständlicher zu machen, kann man das Tupel auch wie in Eingabezeile 3 in die einzelnen Bestandteile zerlegen und diese entsprechend benennen. Es wäre jedoch praktischer, wenn man diesen Schritt nicht explizit vornehmen müsste.

In einem solchen Fall ist ein `namedtuple` nützlich, um lesbaren Code zu schreiben. In der Definition des `namedtuple` zur Darstellung einer Farbe ordnen wir den einzelnen Elementen Namen zu und können mit Hilfe dieser Namen auf die Elemente zugreifen.

```
In [5]: import collections

In [6]: Farbe = collections.namedtuple('Farbe', 'r g b')

In [7]: f1 = Farbe(135, 206, 235)

In [8]: f1[1]
Out[8]: 206

In [9]: f1.g
Out[9]: 206

In [10]: f2 = Farbe(50, 205, 50)

In [11]: f1.b > f2.b
Out[11]: True
```

Gemäß der Definition in Eingabezeile 6 erhalten die Elemente die Bezeichner `r`, `g` und `b` und können dazu verwendet werden, auf die entsprechenden Elemente zuzugreifen, wie in den Eingabezeilen 9 und 11 zu sehen ist. Es ist jedoch auch weiterhin möglich, wie in Eingabezeile 8 auf ein Element mit Hilfe seines Index zuzugreifen.

Die Bezeichner sind nicht nur auf einzelne Buchstaben beschränkt, sondern können bei Bedarf noch aussagekräftiger gewählt werden.

```
In [12]: Farbe = collections.namedtuple('Farbe', 'rot grün blau')

In [13]: f1 = Farbe(135, 206, 235)

In [14]: f1.grün
Out[14]: 206
```

Im Unterschied zum Dictionary ist das `namedtuple`, genauso wie das Tuple, *immutable*, kann also zum Beispiel als Schlüssel für ein Dictionary verwendet werden. Zudem ist es so speichereffizient wie ein normales Tuple.

```
In [15]: f1.rot = 20
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-15-ac8b7e672be5> in <module>()
----> 1 f1.rot = 20

AttributeError: can't set attribute

In [16]: f2 = Farbe(50, 205, 50)

In [17]: rgbdict = {f1: 'SkyBlue', f2: 'LimeGreen'}

In [18]: rgbdict[Farbe(135, 206, 235)]
Out[18]: 'SkyBlue'
```

Ein anderes Problem tritt in Zusammenhang mit Listen auf. Während der Zeitaufwand für das Anhängen eines neuen Elements sehr klein ist, kann das Einfügen eines Elements am Anfang der Liste sehr zeitaufwändig sein wie das folgende Beispiel zeigt.

```
In [19]: %%timeit
    ...: xlist = list()
    ...: for n in range(100000):
    ...:     xlist.append(n)
    ...:
100 loops, best of 3: 7.66 ms per loop

In [20]: %%timeit
    ...: xlist = list()
    ...: for n in range(100000):
    ...:     xlist.insert(0, n)
    ...:
1 loop, best of 3: 2.63 s per loop
```

In einem solchen Fall kann ein so genanntes `deque`, dessen Name sich von *double-ended queue*[2] ableitet, erhebliche Vorteile bringen, so lange man Elemente an einem der beiden Enden hinzufügt oder entfernt.

```
In [21]: %%timeit
    ...: xdeq = collections.deque()
    ...: for n in range(100000):
    ...:     xdeq.append(n)
    ...:
100 loops, best of 3: 7.57 ms per loop

In [22]: %%timeit
    ...: xdeq = collections.deque()
    ...: for n in range(100000):
    ...:     xdeq.appendleft(n)
    ...:
100 loops, best of 3: 7.61 ms per loop
```

Eine mögliche Anwendung ist ein FIFO (*first in, first out*), das Objekte aufnehmen kann und in dieser Reihenfolge auch wieder zurückgibt.

```
In [23]: xdeq = collections.deque([2, 1])

In [24]: xdeq.appendleft(3)

In [25]: xdeq
```

[2] Für weitere Informationen siehe z.B. den Wikipedia-Eintrag zu double-ended queue.

```
Out[25]: deque([3, 2, 1])

In [26]: xdeq.pop()
Out[26]: 1

In [27]: xdeq
Out[27]: deque([3, 2])

In [28]: xdeq.pop()
Out[28]: 2

In [29]: xdeq
Out[29]: deque([3])

In [30]: xdeq.appendleft(4)

In [31]: xdeq
Out[31]: deque([4, 3])

In [32]: xdeq.pop()
Out[32]: 3

In [33]: xdeq
Out[33]: deque([4])
```

Auch das Rotieren eines `deque` ist leicht möglich.

```
In [34]: xdeq = collections.deque(range(10))

In [35]: xdeq
Out[35]: deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [36]: xdeq.rotate(3)

In [37]: xdeq
Out[37]: deque([7, 8, 9, 0, 1, 2, 3, 4, 5, 6])

In [38]: xdeq.rotate(-5)

In [39]: xdeq
Out[39]: deque([2, 3, 4, 5, 6, 7, 8, 9, 0, 1])
```

Abschließend wollen wir noch kurz das `OrderedDict` erwähnen. Bei Dictionaries sind die Schlüssel im Allgemeinen nicht geordnet. Ein `OrderedDict` dagegen merkt sich, in welcher Reihenfolge die Einträge hinzugefügt wurden.

```
In [40]: nobelpreise = dict([('Marie Curie', 1903),
    ...:                     ('Maria Goeppert Mayer', 1963),
    ...:                     ('Klaus von Klitzing', 1985),
    ...:                     ('Albert Einstein', 1921)])

In [41]: for preis in nobelpreise:
    ...:     print(preis)
    ...:
Maria Goeppert Mayer
Marie Curie
Albert Einstein
Klaus von Klitzing

In [42]: nobelpreise = collections.OrderedDict([('Marie Curie', 1903),
    ...:                   ('Maria Goeppert Mayer', 1963),
    ...:                   ('Klaus von Klitzing', 1985),
```

```
    ...:                   ('Albert Einstein', 1921)])

In [43]: for preis in nobelpreise:
    ...:     print(preis)
    ...:
Marie Curie
Maria Goeppert Mayer
Klaus von Klitzing
Albert Einstein
```

Es gibt die Möglichkeit, mit der Methode `move_to_end()` einen einzelnen Eintrag an das Ende eines `OrderedDict` zu verschieben. Um ein bestehendes Dictionary oder `OrderedDict` umzusortieren, erzeugt man am besten ein neues `OrderedDict`.

```
In [44]: nobelpreise_sorted = collections.OrderedDict(
    ...:                           sorted(nobelpreise.items(),
    ...:                                  key=lambda x: x[1]))

In [45]: for name, jahr in nobelpreise_sorted.items():
    ...:     print(jahr, name)
    ...:
1903 Marie Curie
1921 Albert Einstein
1963 Maria Goeppert Mayer
1985 Klaus von Klitzing
```

2.3 List comprehensions

Um eine Liste aufzubauen, kann man sich zum Beispiel der folgenden Konstruktion bedienen.

```
In [1]: squares = []

In [2]: for n in range(10):
   ...:     squares.append(n*n)
   ...:

In [3]: squares
Out[3]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Hierbei wird zunächst eine leere Liste angelegt, an die anschließend in einer Schleife die Quadratzahlen angefügt werden. Etwas kompakter und damit auch übersichtlicher kann man diese Funktionalität mit Hilfe einer so genannten *list comprehension*[3] erreichen.

```
In [1]: squares = [n*n for n in range(10)]

In [2]: squares
Out[2]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Liest man den Text in den eckigen Klammern in Eingabe 1, so bekommt man eine sehr klare Vorstellung davon, was dieser Code bewirken soll. Vor der `for`-Anweisung kann auch eine andere Anweisung stehen, die die Listenelemente erzeugt.

```
In [1]: from math import pi, sin

In [2]: [(0.1*pi*n, sin(0.1*pi*n)) for n in range(6)]
Out[2]: [(0.0, 0.0),
 (0.3141592653589793, 0.3090169943749474),
```

[3] Wir belassen es hier bei dem üblicherweise verwendeten englischen Begriff. Gelegentlich findet man den Begriff »Listenabstraktion« als deutsche Übersetzung.

```
 (0.6283185307179586, 0.5877852522924731),
 (0.9424777960769379, 0.8090169943749475),
 (1.2566370614359172, 0.9510565162951535),
 (1.5707963267948966, 1.0)]
```

List comprehensions sind nicht nur häufig übersichtlicher, sondern in der Ausführung auch etwas schneller.

```
In [1]: %%timeit result = []
   ...: for n in range(1000):
   ...:     result.append(n*n)
   ...:
10000 loops, best of 3: 91.8 µs per loop

In [2]: %timeit result = [n*n for n in range(1000)]
10000 loops, best of 3: 54.4 µs per loop
```

In unserem Fall ist die list comprehension also um fast einen Faktor 1,7 schneller.

Die Syntax von list comprehensions ist nicht auf die bisher beschriebenen einfachen Fälle beschränkt. Sie lässt zum Beispiel auch das Schachteln von Schleifen zu.

```
In [1]: [x**y for y in range(1, 4) for x in range(2, 5)]
Out[1]: [2, 3, 4, 4, 9, 16, 8, 27, 64]

In [2]: result = []

In [3]: for y in range(1, 4):
   ...:     for x in range(2, 5):
   ...:         result.append(x**y)
   ...:

In [4]: result
Out[4]: [2, 3, 4, 4, 9, 16, 8, 27, 64]
```

Wie man sieht, sind die `for`-Schleifen in der list comprehension von der äußersten zur innersten Schleife anzugeben, wobei man auch mehr als zwei Schleifen schachteln kann.

Man kann das Hinzufügen zur Liste zusätzlich noch von Bedingungen abhängig machen. Im folgenden Beispiel wird das Tupel nur in die Liste aufgenommen, wenn die erste Zahl ohne Rest durch die zweite Zahl teilbar ist.

```
In [1]: [(x, y) for x in range(1, 11) for y in range(2, x) if x % y == 0]
Out[1]: [(4, 2), (6, 2), (6, 3), (8, 2), (8, 4), (9, 3), (10, 2), (10, 5)]
```

Als kleines Anwendungsbeispiel betrachten wir den Quicksort-Algorithmus zur Sortierung von Listen. Die Idee hierbei besteht darin, ein Listenelement zu nehmen und die kleineren Elemente in einer rekursiv sortierten Liste diesem Element voranzustellen und die anderen Elemente sortiert anzuhängen.

```
In [1]: def quicksort(x):
   ...:     if len(x) < 2: return x
   ...:     return (quicksort([y for y in x[1:] if y < x[0]])
   ...:             +x[0:1]
   ...:             +quicksort([y for y in x[1:] if x[0] <= y]))

In [2]: import random

In [3]: liste = [random.randint(1, 100) for n in range(10)]

In [4]: liste
Out[4]: [51, 93, 66, 62, 46, 87, 91, 41, 3, 40]

In [5]: quicksort(liste)
Out[5]: [3, 40, 41, 46, 51, 62, 66, 87, 91, 93]
```

Das Konzept der list comprehension lässt sich auch auf Sets und Dictionaries übertragen. Letzteres ist im folgenden Beispiel gezeigt.

```
In [25]: s = 'Augsburg'

In [26]: {x: s.count(x) for x in s}
Out[26]: {'A': 1, 'b': 1, 'r': 1, 's': 1, 'u': 2, 'g': 2}
```

Wie wir gesehen haben, kann eine list comprehension zum einen aus einer Liste durch Anwendung einer Funktion eine andere Liste machen und zum anderen Listenelemente zur Aufnahme in die neue Liste mit Hilfe einer Bedingung auswählen. Diese beiden Komponenten können gemeinsam oder auch einzeln vorkommen. In letzterem Fall kann man alternativ die `map`-Funktion bzw. die `filter`-Funktion verwenden. Beide sind zentrale Elemente des so genannten funktionalen Programmierens.

`map` wendet die im ersten Argument angegebene Funktion auf die im zweiten Argument angegebene Liste an. Um eine Liste von Quadratzahlen zu erzeugen, kann man statt einer expliziten `for`-Schleife auch eine der beiden folgenden Möglichkeiten verwenden:

```
In [1]: [x*x for x in range(1, 11)]
Out[1]: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

In [2]: quadrate = map(lambda x: x*x, range(1, 11))

In [3]: list(quadrate)
Out[3]: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Eine nützliche Anwendung der `map`-Funktion besteht darin, die nach dem Einlesen numerischer Daten zunächst vorhandenen Strings in Floats umzuwandeln:

```
In [1]: s = "0.1 0.2 0.4 -0.5"

In [2]: zeilenelemente = map(float, s.split())

In [3]: list(zeilenelemente)
Out[3]: [0.1, 0.2, 0.4, -0.5]
```

Bei der `filter`-Funktion muss die als erstes Argument angegebene Funktion einen Wahrheitswert zurückgeben, der darüber entscheidet, ob ein Element der Sequenz im zweiten Argument übernommen wird oder nicht.

```
In [1]: initialen = filter(lambda x: x.isupper(), 'Albert Einstein')

In [2]: "".join(initialen)
Out[2]: 'AE'
```

Zur Abwechslung haben wir hier statt einer Liste eine Zeichenkette verwendet, die Zeichen für Zeichen abgearbeitet wird. Das Ergebnis enthält die Großbuchstaben der Zeichenkette.

Zu den wesentlichen Elementen des funktionalen Programmierens gehört auch die `reduce`-Funktion. Während sie in Python 2 noch zum normalen Sprachumfang gehörte, muss sie in Python 3 aus dem `functools`-Modul importiert werden[4]. Tatsächlich gibt es für viele Anwendungsfälle angepasste Funktionen als Ersatz, wie wir gleich noch sehen werden.

Als erstes Argument muss `reduce` eine Funktion bekommen, die zwei Argumente verarbeitet. `reduce` wendet dann die Funktion auf die ersten beiden Elemente der als zweites Argument angegebenen Liste an, verarbeitet dann entsprechend das Ergebnis und das dritte Element der Liste und fährt so fort bis das Ende der Liste erreicht ist. Die folgende Implementation der Fakultät illustriert dies.

```
In [1]: import functools

In [2]: factorial = lambda n: functools.reduce(lambda x, y: x*y, range(1, n+1))
```

[4] Guido von Rossum begründet das in einem Blog mit dem Titel *The fate of reduce() in Python 3000* aus dem Jahr 2005.

```
In [3]: factorial(6)
Out[3]: 720
```

Entsprechend lässt sich auch die Summe der Elemente einer Liste bestimmen.

```
In [1]: reduce(lambda x, y: x+y, [0.1, 0.3, 0.7])
Out[1]: 1.1

In [2]: sum([0.1, 0.3, 0.7])
Out[2]: 1.1
```

Wie die zweite Eingabe zeigt, stellt Python zu diesem Zweck auch direkt die `sum`-Funktion zur Verfügung. Ähnliches gilt für die Verwendung der Oder- und der Und-Verknüpfung in der `reduce`-Funktion, die direkt durch die `any`- bzw. `all`-Funktion abgedeckt werden.

```
In [1]: any([x % 2 for x in [2, 5, 6]])
Out[1]: True

In [2]: all([x % 2 for x in [2, 5, 6]])
Out[2]: False
```

In der ersten Eingabe wird überprüft, ob mindestens ein Element der Liste ungerade ist, während die zweite Eingabe überprüft, ob alle Elemente ungerade sind.

Zum Abschluss dieses Kapitels wollen wir noch zwei äußerst praktische eingebaute Funktionen erwähnen, die, falls sie nicht existieren würden, mit list comprehensions realisiert werden könnten. Häufig benötigt man bei der Iteration über eine Liste in einer `for`-Schleife noch den Index des betreffenden Eintrags. Dies lässt sich mit Hilfe der `enumerate`-Funktion sehr einfach realisieren.

```
In [1]: for nr, text in enumerate(['eins', 'zwei', 'drei']):
   ...:     print(nr+1, text)
   ...:
1 eins
2 zwei
3 drei
```

Die `enumerate`-Funktion gibt also für jedes Element der Liste ein Tupel zurück, das aus dem Index und dem entsprechenden Element besteht. Dabei beginnt die Zählung wie immer in Python bei Null.

Es kommt auch immer wieder vor, dass man zwei oder mehr Listen parallel in einer `for`-Schleife abarbeiten möchte. Dann ist die `zip`-Funktion von Nutzen, die aus den Einträgen mit gleichem Index nach dem Reißverschlussprinzip Tupel zusammenbaut.

```
In [1]: a = [1, 2, 3]

In [2]: b = [4, 5, 6]

In [3]: ab = zip(a, b)

In [4]: list(ab)
Out[4]: [(1, 4), (2, 5), (3, 6)]
```

Sollten die beteiligten Listen verschieden lang sein, so ist die Länge der neuen Liste durch die kürzeste der eingegebenen Listen bestimmt.

Man kann die `zip`-Funktion zum Beispiel dazu verwenden, um elegant Mittelwerte aus aufeinanderfolgenden Listenelementen zu berechnen.

```
In [1]: data = [1, 4, 5, 3, -1, 2]

In [2]: for x, y in zip(data[:-1], data[1:]):
   ...:     print((x+y)/2)
```

```
   ...:
2.5
4.5
4.0
1.0
0.5
```

2.4 Generatoren und Iteratoren

Es kommt häufig vor, dass man Listen mit einer list comprehension erzeugt, nur um anschließend über diese Liste zu iterieren. Dabei reicht es völlig aus, wenn die jeweiligen Elemente erst bei Bedarf erzeugt werden. Somit ist es nicht mehr erforderlich, die ganze Liste im Speicher bereitzuhalten, was bei großen Listen durchaus zum Problem werden könnte.

Will man die Listenerzeugung vermeiden, so kann man statt einer list comprehension einen Generatorausdruck verwenden. Die beiden unterscheiden sich syntaktisch dadurch, dass die umschließenden eckigen Klammern der list comprehension durch runde Klammern ersetzt werden.

```
In [1]: quadrate = (x*x for x in xrange(4))

In [2]: quadrate
Out[2]: <generator object <genexpr> at 0x39e4a00>

In [3]: for q in quadrate:
   ...:     print q
   ...:
0
1
4
9
```

Man kann die Werte des Generatorausdruck auch explizit durch Verwendung der zugehörigen `__next__`-Methode abrufen. Allerdings sind die Werte nach dem obigen Beispiel bereits abgearbeitet, so dass die `__next__`-Methode in einer `StopIteration`-Ausnahme resultiert. Damit wird angezeigt, dass bereits alle Wert ausgegeben wurden. Die `StopIteration`-Ausnahme war auch in der `for`-Schleife verantwortlich dafür, dass diese beendet wurde.

```
In [4]: next(quadrate)
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-4-ec579e92187a> in <module>()
----> 1 quadrate.next()

StopIteration:

In [5]: quadrate = (x*x for x in xrange(4))

In [6]: next(quadrate)
Out[6]: 0

In [7]: next(quadrate)
Out[7]: 1

In [8]: next(quadrate)
Out[8]: 4

In [9]: next(quadrate)
Out[9]: 9
```

```
In [10]: next(quadrate)
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-10-ec579e92187a> in <module>()
----> 1 quadrate.next()

StopIteration:
```

In Eingabe 5 wurde der Generatorausdruck neu initialisiert, so dass er wieder vier Werte liefern konnte. Am Ende wird dann wiederum die `StopIteration`-Ausnahme ausgelöst. Natürlich kann man diese Ausnahme auch abfangen, wie in folgendem Beispiel gezeigt wird.

```
In [1]: def q():
   ...:     try:
   ...:         return quadrate.next()
   ...:     except StopIteration:
   ...:         return "Das war's mit den Quadratzahlen."
   ...:

In [2]: quadrate = (x*x for x in xrange(4))

In [3]: [q() for n in range(5)]
Out[3]: [0, 1, 4, 9, "Das war's mit den Quadratzahlen."]
```

Aus Sequenzen kann man mit Hilfe der eingebauten `iter`-Funktion Iteratoren konstruieren.

```
In [1]: i = iter([1, 2, 3])

In [2]: next(i)
Out[2]: 1

In [3]: next(i)
Out[3]: 2

In [4]: next(i)
Out[4]: 3

In [5]: next(i)
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-5-e590fe0d22f8> in <module>()
----> 1 next(i)

StopIteration:
```

In der Eingabe 1 wird ein Iterator erzeugt, der über eine `__next__`-Methode verfügt und nach dem Abarbeiten der Liste eine `StopIteration`-Ausnahme auslöst. Iteratoren kann man auch über eine Klassendefinition erhalten, wie im folgenden Beispiel für die Fibonacci-Zahlen gezeigt ist.

```
In [1]: class Fibonacci(object):
   ...:     def __init__(self, nmax):
   ...:         self.nmax = nmax
   ...:         self.a = 0
   ...:         self.b = 1
   ...:     def __iter__(self):
   ...:         return self
   ...:     def __next__(self):
   ...:         if self.nmax == 0:
   ...:             raise StopIteration
   ...:         self.b, self.a = self.b+self.a, self.b
   ...:         self.nmax = self.nmax-1
```

```
   ...:         return self.a
   ...:

In [2]: for n in Fibonacci(10):
   ...:     print(n, end=' ')
   ...:
1 1 2 3 5 8 13 21 34 55
```

Die __iter__-Methode dieser Klasse gibt sich selbst zurück, während die __next__-Methode das jeweils nächste Element zurückgibt. Nachdem die ersten nmax Elemente der Fibonacci-Reihe erzeugt wurden, wird eine StopIteration-Ausnahme ausgelöst, die zur Beendung der for-Schleife in Eingabe 2 führt.

Normalerweise wird es einfacher sein, statt einer solchen Klassendefinition einen Generator zu schreiben. Dieser sieht auf den ersten Blick wie eine Funktionsdefinition aus. Allerdings ist die return-Anweisung durch eine yield-Anweisung ersetzt, die dafür verantwortlich ist, den jeweils nächsten Wert zurückzugeben. Bemerkenswert ist im Vergleich zu Funktionen außerdem, dass die Werte der Funktionsvariablen nicht verlorengehen. Das folgende Beispiel erzeugt die ersten Zeilen eines pascalschen Dreiecks.

```
In [1]: def pascaltriangle(n):
   ...:     coeff = 1
   ...:     yield coeff
   ...:     for m in range(n):
   ...:         coeff = coeff*(n-m)//(m+1)
   ...:         yield coeff

In [2]: for n in range(11):
   ...:     print " ".join(str(p).center(3) for p in pascaltriangle(n)).center(50)
   ...:
                        1
                      1   1
                    1   2   1
                  1   3   3   1
                1   4   6   4   1
              1   5   10  10  5   1
            1   6   15  20  15  6   1
          1   7   21  35  35  21  7   1
        1   8   28  56  70  56  28  8   1
      1   9   36  84 126 126  84  36  9   1
    1   10  45 120 210 252 210 120  45  10  1
```

In der letzten Zeile fungieren die Klammern der join-Methode gleichzeitig als Klammern für den Generatorausdruck.

+ Man kann yield auch benutzen, um Werte in die Funktion einzuspeisen. Auf diese Weise erhält man eine Koroutine. Dieses Konzept soll hier jedoch nicht weiter diskutiert werden.

Abschließend sei noch erwähnt, dass das itertools-Modul eine ganze Reihe von nützlichen Iteratoren zur Verfügung stellt. Als Beispiel mögen Permutationen dienen.

```
In [1]: import itertools

In [2]: for s in itertools.permutations("ABC"):
   ...:     print s
   ...:
('A', 'B', 'C')
('A', 'C', 'B')
('B', 'A', 'C')
('B', 'C', 'A')
('C', 'A', 'B')
('C', 'B', 'A')
```

2.5 Dekoratoren

Dekoratoren sind ein Programmierkonstrukt, das man gelegentlich gewinnbringend einsetzen kann, wie wir im Folgenden sehen werden. Aber selbst wenn man keine eigenen Dekoratoren programmieren möchte, sollte man zumindest das Konzept kennen. Es kommt immer wieder vor, dass bei der Verwendung von fremden Python-Paketen Dekoratoren zum Einsatz kommen. Dies kann dann zum Beispiel folgendermaßen aussehen:

```
@login_required
def myfunc():
    """this function should only be executable by users properly logged in"""
    pass
```

Der Operator `@` weist hier auf die Verwendung eines Dekorators hin.

Bevor wir uns aber mit Dekoratoren beschäftigen, ist es nützlich, so genannte Closures[5] zu diskutieren. Das Konzept soll an einem einfachen Beispiel erläutert werden.

```
In [1]: def add_tax(taxrate):
   ...:     def _add_tax(value):
   ...:         return value*(1+0.01*taxrate)
   ...:     return _add_tax
   ...:

In [2]: add_mwst = add_tax(19)

In [3]: add_reduzierte_mwst = add_tax(7)

In [4]: for f in [add_mwst, add_reduzierte_mwst]:
   ...:     print('{:.2f}'.format(f(10)))
   ...:
11.90
10.70
```

Mit `add_tax` haben wir hier eine Funktion definiert, die wiederum eine Funktion zurückgibt. Das Interessante an dieser Konstruktion ist, dass die zurückgegebene Funktion sich den Kontext merkt, in dem sie erzeugt wurde. In unserem Beispiel bedeutet das, dass die Funktion `_add_tax` auf den Wert der Variable `taxrate`, also den Steuersatz, auch später noch zugreifen kann. Dies wird deutlich, wenn wir zur Addition des vollen Mehrwertsteuersatzes die Funktion `add_mwst` definieren. Hierbei wird der Variable `taxrate` der Wert 19 mitgegeben, der später beim Aufruf von `add_mwst` noch zur Verfügung steht. Entsprechend definieren wir eine Funktion zur Addition des reduzierten Mehrwertsteuersatzes. Am Beispiel der abschließenden Schleife wird deutlich, dass die Funktionen wie gewünscht funktionieren.

Kommen wir nun zurück zu den Dekoratoren. Diese erlauben es, Funktionen oder Klassen mit Zusatzfunktionalität zu versehen oder diese zu modifizieren. Wir wollen uns hier auf Funktionen beschränken. Betrachten wir als ein erstes Beispiel den folgenden Code:

```
In [1]: def register(func):
   ...:     print('{} registered'.format(func.__name__))
   ...:     return func
   ...:

In [2]: @register
   ...: def myfunc():
   ...:     print('executing myfunc')
   ...:
myfunc registered

In [3]: myfunc()
executing myfunc
```

[5] Wir belassen es auch hier wieder bei dem häufig verwendeten englischen Begriff, der als »Funktionsabschluss« zu übersetzen wäre.

```
In [4]: @register
   ...: def myotherfunc():
   ...:     print('executing myotherfunc')
   ...:
myotherfunc registered

In [5]: myotherfunc()
executing myotherfunc
```

Hier haben wir zunächst einen Dekorator `register` definiert, der als Argument eine Funktion erhält. Bevor er sie unverändert zurückgibt, registriert er die Funktion, was hier durch eine einfache Ausgabe nur angedeutet wird. Der Dekorator kann nun verwendet werden, indem vor der gewünschten Funktion die Zeile `@register` eingefügt wird. Wie schon erwähnt, gibt der Operator `@` an, dass hier ein Dekorator verwendet wird, die folgende Funktion also dekoriert wird. In der Eingabe 2 wird `myfunc` als Argument an `register` übergeben. Bei der Auswertung der Funktionsdefinition wird nun der Ausgabebefehl ausgeführt. Später erfolgt dies nicht mehr, da der Dekorator `register` die Funktion ja unverändert zurückgegeben hat. Die Eingabe 4 zeigt, dass der Dekorator mit einer beliebigen Funktion verwendet werden kann.

Wenden wir uns nun einem etwas komplexeren Beispiel zu. Wir haben in dem untenstehenden Code-Beispiel in den Zeilen 17-21 die Minimalvariante einer rekursiven Funktion für die Berechnung der Fakultät programmiert. Diese Funktion soll nun so modifiziert werden, dass Logging-Information ausgegeben wird. Zu Beginn der Funktion soll ausgegeben werden, mit welchem Argument die Funktion aufgerufen wurde und am Ende sollen zusätzlich das Ergebnis und die seit dem Aufruf verstrichene Zeit ausgegeben werden.

Natürlich könnte die entsprechende Funktionalität direkt in die Funktion programmiert werden, aber es spricht einiges dagegen, so vorzugehen. Die Fähigkeit, Logging-Information auszugeben, hat nichts mit der Berechnung der Fakultät zu tun, und daher ist es besser, die beiden Funktionalitäten sauber zu trennen. Dies wird deutlich, wenn man bedenkt, dass die Ausgabe von Logging-Information vor allem in der Entwicklungsphase erforderlich ist und später wahrscheinlich entfernt werden soll. Dann müsste man wieder in das Innere der Funktion eingreifen und die richtigen Zeilen identifizieren, die entfernt werden müssen. Außerdem ist die Ausgabe von Logging-Information etwas, was nicht nur für unsere spezielle Funktion nützlich ist, sondern auch in anderen Fällen verwendet werden kann. Dies spricht wiederum dafür, diese Funktionalität aus der eigentlichen Funktion fernzuhalten.

Genau dieses Ziel ist in dem folgenden Code realisiert, in dem die Funktion `factorial` mit einem Dekorator versehen ist.

```
1  import time
2  from itertools import chain
3
4  def logging(func):
5      def func_with_log(*args, **kwargs):
6          argumente = ', '.join(map(repr, chain(args, kwargs.items())))
7          print('calling {}({})'.format(func.__name__, argumente))
8          start = time.time()
9          result = func(*args, **kwargs)
10         elapsed = time.time()-start
11         print('got {}({}) = {} in {:5.3f} ms'.format(
12                 func.__name__, argumente, result, elapsed*1000
13                                                     ))
14         return result
15     return func_with_log
16
17 @logging
18 def factorial(n):
19     if n == 1:
20         return 1
21     else:
22         return n*factorial(n-1)
23
24 factorial(5)
```

Sehen wir uns nun also den Dekorator `logging` in den Zeilen 4-14 an. Wie schon angedeutet, wird die Funktion `factorial` als Argument `func` an die Funktion `logging` übergeben. Der wesentliche Teil von `logging` besteht darin, eine neue Funktion, die hier den Namen `func_with_log` trägt, zu definieren, die die Funktion `func` ersetzen wird. Die Definition in den Zeilen 5-13 ist absichtlich allgemeiner gehalten als es für uns Beispiel notwendig wäre. So lässt sich der Dekorator auch in anderen Fällen direkt einsetzen. Daher lassen wir in Zeile 5 eine allgemeine Übergabe von Variablen in einem Tupel `args` und einem Dictionary `kwargs` zu.

Die zugehörigen Werte werden in der Zeile 6 durch Kommas separiert zu einem String zusammengebaut. Dabei übernimmt die aus dem `itertools`-Modul importierte Funktion `chain` die Aufgabe, die Elemente des Tupels `args` und der Schlüssel-Wert-Tupel des Dictionaries `kwargs` zu einer einzigen Sequenz zusammenzufassen. Mit Hilfe der `map`-Funktion wird zu jedem Element mit Hilfe der `repr`-Funktion die zugehörige Darstellung erzeugt. Die `join`-Funktion baut diese Darstellung schließlich zu einem String zusammen. Nachdem dieses Verfahren etwas komplexer ist, sei angemerkt, dass dieses Vorgehen nichts mit dem Dekorator an sich zu tun hat. Vielmehr ist es durch unsere Anforderung bedingt, Logging-Information einschließlich der Aufrufparameter ausgeben zu können, und dies nicht nur für die `factorial`-Funktion, sondern in einem möglichst allgemeinen Fall. Die Ausgabe der Logging-Information erfolgt in Zeile 7. Zeile 8 bestimmt den Startzeitpunkt. In diesem Zusammenhang wurde in Zeile 1 das `time`-Modul importiert.

Nach diesen Vorarbeiten wird in Zeile 9 die eigentliche Funktion, die die Fakultät berechnen soll, aufgerufen. Typischerweise wird die ursprüngliche Funktion tatsächlich aufgerufen. Allerdings ist dies nicht unbedingt notwendig. Man könnte stattdessen hier einfach einen Hinweis ausgeben, dass nun die Fakultät zu berechnen wäre. Auf diese Weise würde man jedoch kein Ergebnis für die Fakultät erhalten.

Nachdem die `factorial`-Funktion ihr Ergebnis zurückgegeben hat, wird in Zeile 10 die verstrichene Zeit bestimmt und in Zeile 11 in Millisekunden gemeinsam mit dem Ergebnis ausgegeben. Abschließend soll die dekorierte Funktion das berechnete Resultat zurückgeben. Damit ist die Definition der dekorierten Funktion beendet und der Dekorator gibt diese Funktion in Zeile 14 zurück.

Ruft man in Zeile 24 nun die Funktion `factorial` auf, so wird wegen des `logging`-Dekorators in Wirklichkeit die gerade besprochene, dekorierte Funktion ausgeführt. Man erhält somit die folgende Ausgabe:

```
calling factorial(5)
calling factorial(4)
calling factorial(3)
calling factorial(2)
calling factorial(1)
got factorial(1) = 1 in 0.004 ms
got factorial(2) = 2 in 0.085 ms
got factorial(3) = 6 in 0.163 ms
got factorial(4) = 24 in 0.281 ms
got factorial(5) = 120 in 0.524 ms
```

In dieser Ausgabe ist gut zu sehen, wie durch die rekursive Abarbeitung nacheinander die Fakultät von 5, von 4, von 3, von 2 und von 1 berechnet wird. Die Ausführungen sind geschachtelt, denn die Berechnung der Fakultät von 2 kann erst beendet werden, wenn die Fakultät von 1 bestimmt wurde. Entsprechend benötigt die Berechnung der Fakultät von 5 auch mehr Zeit als die Berechnung der Fakultät von 4 usw. Der `logging`-Dekorator erlaubt somit Einblicke in die Abarbeitung der rekursiven Funktion ohne dass wir in diese Funktion direkt eingreifen mussten.

Abschließend betrachten wir noch ein weiteres Anwendungsbeispiel, das besonders dann von Interesse ist, wenn die Ausführung einer Funktion relativ aufwändig ist. Dann kann es sinnvoll sein, Ergebnisse aufzubewahren und auf diese bei einem erneuten Aufruf mit den gleichen Argumenten wieder zuzugreifen. Dies setzt natürlich eine deterministische Funktion voraus, also eine Funktion, deren Ergebnis nur von den übergebenen Argumenten abhängt. Außerdem wird der Gewinn an Rechenzeit mit Speicherplatz bezahlt. Dies ist jedoch normalerweise unproblematisch, so lange sich die Zahl verschiedener Argumentwerte in Grenzen hält.

```
1 import functools
2
3 def memoize(func):
4     cache = {}
5     @functools.wraps(func)
6     def _memoize(*args):
```

```
7           if args in cache:
8               return cache[args]
9           result = func(*args)
10          cache[args] = result
11          return result
12      return _memoize
13
14  @logging
15  @memoize
16  def factorial(n):
17      if n == 1:
18          return 1
19      else:
20          return n*factorial(n-1)
```

Im `memoize`-Dekorator ist hier eine Closure realisiert, die es der Funktion `_memoize` erlaubt, auch später auf das Dictionary `cache` zuzugreifen, in dem die Ergebnisse gespeichert werden. Hierzu eignet sich ein Dictionary, weil man die Argumente in Form des Tupels `args` als Schlüssel hinterlegen kann. Allerdings ist es nicht möglich, auch ein eventuelles Dictionary `kwargs` im Schlüssel unterzubringen. Argumente, die mit Schlüsselworten übergeben werden, sind hier somit nicht erlaubt.

Aus den Zeilen 14 und 15 ersieht man, dass Dekoratoren auch geschachtelt werden können. Die Funktion `factorial` wird zunächst mit dem `memoize`-Dekorator versehen. Die so dekorierte Funktion wird dann mit dem `logging`-Dekorator versehen. Eine Schwierigkeit besteht hier allerdings darin, dass der `logging`-Dekorator für ein korrektes Funktionieren den Namen der ursprünglichen Funktion, also `factorial` benötigt. Aus diesem Grunde verwendet man in Zeile 5 den `wraps`-Dekorator aus dem `functools`-Modul, der dafür sorgt, dass Name und Dokumentationsstring diejenigen der Funktion `func` und nicht der Funktion `_memoize` sind.

Im Folgenden ist die Funktionsweise des `memoize`-Dekorators gezeigt.

```
In [1]: factorial(4)
calling factorial(4)
calling factorial(3)
calling factorial(2)
calling factorial(1)
got factorial(1) = 1 in 0.005 ms
got factorial(2) = 2 in 0.063 ms
got factorial(3) = 6 in 0.252 ms
got factorial(4) = 24 in 0.369 ms
Out[2]: 24

In [2]: factorial(3)
calling factorial(3)
got factorial(3) = 6 in 0.007 ms
Out[3]: 6
```

Beim ersten Aufruf der Funktion `factorial` wird die Fakultät rekursiv ausgewertet wie wir das schon weiter oben gesehen haben. Dabei werden aber die berechneten Werte im Dictionary `cache` gespeichert. Ruft man nun die Funktion mit einem Argument auf, dessen Fakultät bereits berechnet wurde, kann direkt auf das Ergebnis im Cache zugegriffen werden. Dies zeigt sich daran, dass keine rekursive Berechnung mehr durchgeführt wird und die benötigte Zeit bis zur Rückgabe des Ergebnisses sehr kurz ist.

Abschließend sei noch kurz erwähnt, dass Dekoratoren auch ein Argument haben können, das wie üblich in Klammern angegeben wird. Dabei ist zu beachten, dass dem Dekorator dann nicht mehr wie in den hier diskutierten Beispielen die zu dekorierende Funktion übergeben wird, sondern das angegebene Argument. Die zu dekorierende Funktion wird dafür dann an die im Dekorator definierte Funktion übergeben.

2.6 Ausnahmen

Bereits in der »Einführung in das Programmieren« hatten wir Ausnahmen (*exceptions*) kennengelernt und gesehen, wie man mit einem `except`-Block auf eine Ausnahme reagieren kann. Zudem haben wir im Abschnitt *Generatoren und Iteratoren* eine Anwendung gesehen, in der wir selbst eine Ausnahme ausgelöst haben, nämlich eine `StopIteration`-Ausnahme. Im Folgenden sollen noch einige Aspekte von Ausnahmen diskutiert werden, die bis jetzt zu kurz kamen.

Grundsätzlich ist es sinnvoll, möglichst spezifisch auf Ausnahmen zu reagieren. Daher sollte zum einen der `try`-Block kurz gehalten werden, um einen möglichst direkten Zusammenhang zwischen Ausnahme und auslösendem Code zu garantieren. Zum anderen sollten nicht unnötig viele Ausnahmen gleichzeitig in einem `except`-Block abgefangen werden.

```
In [1]: try:
   ...:     datei = open('test.dat')
   ...: except IOError as e:
   ...:     print('abgefangener Fehler:', e)
   ...: else:
   ...:     content = datei.readlines()
   ...:     datei.close()
   ...:

In [2]: content
Out[2]: ['Das ist der Inhalt der Test-Datei.\n']

In [3]: try:
   ...:     datei = open('test.dat')
   ...: except IOError as e:
   ...:     print('abgefangener Fehler:', e)
   ...: else:
   ...:     content = datei.readlines()
   ...:     datei.close()
   ...:
abgefangener Fehler: [Errno 2] No such file or directory: 'test.dat'
```

Bei der ersten Eingabe ist die Datei `test.dat` vorhanden und kann geöffnet werden. Der `except`-Block wird daher übersprungen und der `else`-Block ausgeführt. Im Prinzip hätte man den Inhalt des `else`-Blocks auch im `try`-Block unterbringen können. Die hier gezeigte Variante hat jedoch den Vorteil, dass der Zusammenhang zwischen dem Versuch, eine Datei zu öffnen, und der eventuellen `IOError`-Ausnahme eindeutig ist. In der Eingabe 3 ist die Datei `test.dat` nicht vorhanden, und es wird der `except`-Block ausgeführt. Die Variable `e` nach dem Schlüsselwort `as` enthält dabei Informationen, die beim Auslösen der Ausnahme übergeben wurden und hier im `except`-Block zur Information des Benutzers ausgegeben werden. Der `else`-Block wird hier im Gegensatz zum ersten Fall nicht ausgeführt.

Wenn die Ausführung des Codes im `try`-Block potentiell zu verschiedenen Ausnahmen führen kann, ist es sinnvoll, mehrere `except`-Blöcke vorzusehen, wie das folgende Beispiel zeigt.

```
1  def myfunc(x):
2      mydict = {1: 'eins', 2: 'zwei'}
3      try:
4          print(mydict[int(x)])
5      except KeyError as e:
6          print('KeyError:', e)
7      except TypeError as e:
8          print('TypeError:', e)
9
10 myfunc(1.5)
11 myfunc(5.5)
12 myfunc(1+3j)
```

Während der Funktionsaufruf in Zeile 10 keine Ausnahme auslöst, führen die Aufrufe in den Zeilen 11 und 12 zu einem `KeyError`, da es den Schlüssel `5` in `mydict` nicht gibt, bzw. zu einem `TypeError` weil sich die kom-

plexe Zahl `1+3j` nicht in einen Integer umwandeln lässt. Die Ausgabe sieht dementsprechend folgendermaßen aus:

```
eins
KeyError: 5
TypeError: can't convert complex to int
```

In diesem Beispiel wird Code wiederholt. Dies lässt sich verhindern, wenn man die Funktion wie folgt definiert.

```
1 def myfunc(x):
2     mydict = {1: 'eins', 2: 'zwei'}
3     try:
4         print(mydict[int(x)])
5     except (KeyError, TypeError) as e:
6         print(": ".join([type(e).__name__, str(e)]))
```

Möchte man alle Ausnahmen abfangen, so kann man das Tupel in Zeile 5 zum Beispiel durch `Exception` oder gar `BaseException` ersetzen. Auf den Hintergrund hierfür kommen wir etwas später noch zurück.

Einer Folge von `except`-Blöcken könnte sich natürlich auch wieder ein `else`-Block anschließen, wie wir dies weiter oben gesehen hatten. Allerdings gibt es nicht nur Situationen, wo abhängig vom Auftreten einer Ausnahme der eine oder andere Block abgearbeitet wird, sondern es kann auch vorkommen, dass am Ende auf jeden Fall ein gewisser Codeblock ausgeführt werden soll. Ein typischer Fall ist das Schreiben in eine Datei. Dabei muss am Ende sichergestellt werden, dass die Datei geschlossen wird. Hierzu dient der `finally`-Block. Betrachten wir ein Beispiel.

```
def myfunc(nr, x):
    datei = open('test_%i.dat' % nr, 'w')
    datei.write('ANFANG\n')
    try:
        datei.write('%g\n' % (1/x))
    except ZeroDivisionError as e:
        print('ZeroDivisionError:', e)
    finally:
        datei.write('ENDE\n')
        datei.close()

for nr, x in enumerate([1.5, 0, 'Test']):
    myfunc(nr, x)
```

In der Funktion `myfunc` soll der Kehrwert des Arguments in die Datei `test.dat` geschrieben werden. Es ergibt sich die folgende Ausgabe:

```
--- test_0.dat ---
ANFANG
0.666667
ENDE

--- test_1.dat ---
ANFANG
ENDE

--- test_2.dat ---
ANFANG
ENDE
```

Dabei wird im zweiten Fall wegen der Division durch Null kein Kehrwert ausgeben, während im dritten Fall ein `TypeError` auftritt, weil versucht wird, durch einen String zu dividieren. Diese Ausnahme wird zwar nicht abgefangen, aber es ist immerhin garantiert, dass die Ausgabedatei ordnungsgemäß geschlossen wird.

Was würde passieren, wenn man das Schließen der Datei nicht in einem `finally`-Block unterbringt, sondern einfach am Ende der Funktion ausführen lässt? Wir modifizieren unseren Code entsprechend:

```
1  def myfunc(nr, x):
2      datei = open('test_%i.dat' % nr, 'w')
3      datei.write('ANFANG\n')
4      try:
5          datei.write("%g\n" % (1/x))
6      except ZeroDivisionError as e:
7          print('ZeroDivisionError:', e)
8      datei.write('ENDE\n')
9      datei.close()
10
11 for nr, x in enumerate([1.5, 0, 'Test']):
12     myfunc(nr, x)
```

Nun erhält man die folgenden Dateiinhalte:

```
--- test_0.dat ---
ANFANG
0.666667
ENDE

--- test_1.dat ---
ANFANG
ENDE

--- test_2.dat ---
ANFANG
```

Wie man sieht, ist die letzte Datei unvollständig. Die nicht abgefangene `TypeError`-Ausnahme führt zu einem Programmabbruch, der sowohl die Ausführung des `write`-Befehls in Zeile 10 als auch das Schließen der Datei verhindert. In der ersten Variante des Programms dagegen gehört der `finally`-Block zum `try...except`-Block und wird somit auf jeden Fall ausgeführt. Erst danach führt der `TypeError` in diesem Fall zum Programmabbruch.

Selbst in obigem Beispiel ohne `finally`-Block wurde die Datei geschlossen, da dies spätestens durch das Betriebssystem beim Programmende veranlasst wird. Es ist aber dennoch kein guter Stil, sich hierauf zu verlassen. Eine Datei nicht zu schließen, kann in Python Schwierigkeiten bereiten, wenn man die Datei anschließend wieder zum Lesen öffnen will. Auch wenn man auf eine große Zahl von Dateien schreiben möchte, kann es zu Problemen kommen, wenn man Dateien nicht schließt, da die Zahl der offenen Dateien beschränkt ist. In diesem Zusammenhang gibt es Unterschiede zwischen verschiedenen Implementationen von Python. In CPython, also der standardmäßig verwendeten, in der Programmiersprache C implementierten Version von Python, sorgt ein als »garbage collection« bezeichneter Prozess, also das Einsammeln von (Daten-)Müll, dafür, dass überflüssige Objekte entfernt werden. Hierbei werden auch Dateien geschlossen, auf die nicht mehr zugegriffen wird. Allerdings wird dies nicht durch die Sprachdefinition garantiert. In Jython, einer Python-Implementation für die Java Virtual Machine, ist dies tatsächlich nicht der Fall.

Python stellt standardmäßig bereits eine große Zahl von Ausnahmen zur Verfügung, die alle als Unterklassen von einer Basisklasse, der `BaseException` abgeleitet sind. Die folgende Klassenhierarchie der Ausnahmen ist der Python-Dokumentation entnommen, wo die einzelnen Ausnahmen auch genauer beschrieben sind.[6]

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
```

[6] Siehe 6. Built-in Exceptions in der Dokumentation der Standardbibliothek von Python.

```
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
      |              +-- UnicodeDecodeError
      |              +-- UnicodeEncodeError
      |              +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
       +-- ImportWarning
       +-- UnicodeWarning
       +-- BytesWarning
```

Will man gleichzeitig Ausnahmen abfangen, die Unterklassen einer gemeinsamen Klasse sind, so kann man stattdessen auch direkt die entsprechende Ausnahmeklasse abfangen. Somit sind beispielsweise

```
except (IndexError, KeyError) as e:
```

und

```
except LookupError as e:
```

äquivalent. Man kann die vorhandenen Ausnahmeklassen, sofern sie von der Fehlerart her passend sind, auch direkt für eigene Zwecke verwenden oder Unterklassen programmieren. Beim Auslösen einer Ausnahme kann dabei auch eine entsprechende Fehlermeldung mitgegeben werden.

```
In [1]: raise ValueError('42 ist keine erlaubte Eingabe!')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-1-c6e93f8997ca> in <module>()
----> 1 raise ValueError("42 ist keine erlaubte Eingabe!")

ValueError: 42 ist keine erlaubte Eingabe!
```

Hat man eine Ausnahme abgefangen, so hat man die Möglichkeit, nach einer adäquaten Reaktion die Ausnahme

erneut auszulösen. Geschieht dies in einer Funktion, so hat das aufrufende Programm wiederum die Möglichkeit, entsprechend zu reagieren. Dies ist im folgenden Beispiel illustriert.

```
def reciprocal(x):
    try:
        return 1/x
    except ZeroDivisionError:
        msg = 'Maybe the main program knows what to do...'
        raise ZeroDivisionError(msg)

try:
    reciprocal(0)
except ZeroDivisionError as e:
    print(e)
    print("Let's just continue!")

print("That's the end of the program.")
```

Dieses Programm erzeugt die folgende Ausgabe:

```
Maybe the main program knows what to do...
Let's just continue!
That's the end of the program.
```

Die Funktion `reciprocal` fängt die Division durch Null ab. Sie verhindert damit den vorzeitigen Programmabbruch und gibt dem Hauptprogramm die Chance, in geeigneter Weise zu reagieren. Dies geschieht hier, indem wiederum der `ZeroDivisionError` abgefangen wird.

2.7 Kontext mit `with`-Anweisung

Im vorigen Abschnitt hatten wir im Zusammenhang mit dem Zugriff auf eine Datei ein typisches Szenario kennengelernt, bei dem die eigentliche Funktionalität zwischen zwei Schritte eingebettet ist, in denen zunächst Vorbereitungen getroffen werden und am Ende notwendige Aufräumarbeiten durchgeführt werden. In unserem Beispiel wäre dies das Öffnen der Datei zu Beginn und das Schließen der Datei am Ende. Eine solche Situation kann in Python mit Hilfe eines Kontextmanagers elegant bewältigt werden. Dies ist im folgenden Beispiel gezeigt.

```
with open('test.dat', 'w') as file:
    for n in range(4, -1, -1):
        file.write('{:g}\n'.format(1/n))
```

Dies entspricht einem `try...finally`-Konstrukt, bei dem im `finally`-Block unabhängig vom Auftreten einer Ausnahme die Datei wieder geschlossen wird. Die Ausgabedatei hat dann den folgenden Inhalt:

```
0.25
0.333333
0.5
1
```

Sie wurde explizit beim Verlassen des `with`-Blocks geschlossen, nachdem zuvor die Variable `n` den Wert Null erreicht hat und die Division eine `ZeroDivisionError`-Ausnahme ausgelöst hat. Um dies zu überprüfen, muss man die Ausnahme abfangen.

```
try:
    with open('test.dat', 'w') as file:
        for n in range(4, -1, -1):
            file.write('{:g}\n'.format(1/n))
except ZeroDivisionError:
    print('division by zero')

print('file is closed: {}'.format(file.closed))
```

Die zugehörige Ausgabe lautet dann wie erwartet:

```
division by zero
file is closed: True
```

Kontextmanager können unter anderem beim Arbeiten mit Cython[7] nützlich sein. Cython ermöglicht die Optimierung von Python-Skripten, indem es C-Erweiterungen anbietet. Dazu gehört unter anderem die Möglichkeit, den Datentyp von Variablen festzulegen. In Python werden Listenindizes normalerweise darauf überprüft, ob sie innerhalb des zulässigen Bereichs liegen, und es werden negative Indizes entsprechend behandelt. Dies kostet natürlich Zeit. Ist man sich sicher, dass man weder negative Indizes benutzt noch die Listengrenzen überschreitet, so kann man bei der Benutzung von Cython auf die genannte Funktionalität verzichten. Will man dies in einem begrenzten Code-Block tun, so bietet sich die Verwendung von `with cython.boundscheck(False)` an. Eine andere Anwendung besteht im Ausschalten des *Global Interpreter Locks*[8] von Python mit Hilfe des `nogil`-Kontextmanagers.

[7] Weitere Informationen zu Cython findet man unter www.cython.org. Cython sollte nicht mit CPython verwechselt werden, der C-Implementation von Python, die man standardmäßig beim `python`-Aufruf verwendet.

[8] Siehe das Glossar der Python-Dokumentation für eine kurze Erläuterung des GIL.

KAPITEL 3

NumPy

In der Vorlesung »Einführung in das Programmieren für Physiker und Naturwissenschaftler« wurde am Beispiel von NumPy und SciPy eine kurze Einführung in die Benutzung numerischer Programmbibliotheken gegeben. Dabei wurde an einigen wenigen Beispielen gezeigt, wie man in Python mit Vektoren und Matrizen arbeiten und einfache Problemstellungen der linearen Algebra lösen kann. Im Folgenden wollen wir uns etwas genauer mit NumPy beschäftigen, das die Basis für wichtige wissenschaftliche Programmbibliotheken bildet, wie das bereits genannte SciPy, Matplotlib für die Erstellung von Grafiken, Pandas für die Analyse großer Datenmengen, Scikit-image für die Bildbearbeitung, Scikit-learn für maschinenbasiertes Lernen und einige andere mehr.

Wegen des großen Umfangs der von NumPy zur Verfügung gestellten Funktionalität werden wir uns auf wesentliche Aspekte beschränken und keine vollständige Beschreibung anstreben. Bei Bedarf sollte daher die NumPy Referenzdokumentation herangezogen werden. Als Informationsquelle sind zudem die Python Scientific Lecture Notes empfehlenswert. Dort werden auch weitere Programmbibliotheken diskutiert, die in naturwissenschaftlichen Anwendungen hilfreich sein können.

3.1 Python-Listen und Matrizen

Viele naturwissenschaftliche Problemstellungen lassen sich in natürlicher Weise mit Hilfe von Vektoren und Matrizen formulieren. Dies kann entweder eine Eigenschaft des ursprünglichen Problems sein, beispielsweise bei der Beschreibung eines gekoppelten schwingenden Systems mit Hilfe von gekoppelten Differentialgleichungen. Es kann aber auch vorkommen, dass erst die numerische Umsetzung zu einer Formulierung in Vektoren und Matrizen führt, zum Beispiel bei der Diskretisierung einer partiellen Differentialgleichung.

Will man solche Problemstellungen mit den Standardmitteln bearbeiten, die von Python zur Verfügung gestellt werden, so wird man auf Listen zurückgreifen müssen. Um eine zweidimensionale Matrix zu definieren, würde man eine Liste von Listen anlegen und könnte dann durch eine doppelte Indizierung auf ein einzelnes Element zugreifen.

```
In [1]: matrix = [[1.1, 2.2, 3.3], [4.4, 5.5, 6.6], [7.7, 8.8, 9.9]]

In [2]: matrix[0]
Out[2]: [1.1, 2.2, 3.3]

In [3]: matrix[0][2]
Out[3]: 3.3
```

Das Beispiel erklärt die doppelte Indizierung. Durch den ersten Index, hier `[0]`, wird die erste Unterliste ausgewählt, aus der wiederum ein einzelnes Element, hier das dritte, ausgewählt werden kann.

Eine Zeile kann man entweder wie oben in der Eingabe 2 erhalten oder auch etwas umständlicher mit

```
In [4]: matrix[0][:]
Out[4]: [1.1, 2.2, 3.3]
```

Hier ist explizit angegeben, dass wir alle Elemente der ersten Zeile haben wollen. Ein enstprechender Zugriff auf eine Spalte funktioniert jedoch nicht:

```
In [5]: matrix[:][0]
Out[5]: [1.1, 2.2, 3.3]
```

Hier gibt `matrix[:]` eine Liste mit allen Unterlisten, also einfach die ursprüngliche Liste zurück. Somit ist `matrix[:][0]` nichts anderes als die erste Unterliste. Wir erhalten also wiederum die erste Zeile und keineswegs die erste Spalte. Auch wenn es beispielsweise mit Hilfe einer list comprehension möglich ist, eine Spalte aus einer Matrix zu extrahieren, zeigt das Beispiel, dass Zeilen und Spalten in einer durch eine Liste dargestellten Matrix nicht in gleicher Weise behandelt werden können. Für eine Matrix würde man eine Gleichbehandlung jedoch auf jeden Fall erwarten.

Ein weiterer Nachteil besteht in der Flexibilität von Listen, die ja bekanntlich beliebige Objekte enthalten können. Python muss daher einen erheblichen Aufwand bei der Verwaltung von Listen treiben. Dies betrifft alleine schon die Adressierung eines einzelnen Elements. Andererseits wird diese Flexibilität bei Matrizen überhaupt nicht benötigt, da dort alle Einträge vom gleichen Datentyp sind. Es sollte also möglich sein, erheblich effizientere Programme zu schreiben, indem man Matrizen nicht durch Listen darstellt, sondern durch einen auf diese Aufgabe zugeschnittenen Datentypen. Hierzu greift man auf das von NumPy zur Verfügung gestellte `ndarray`-Objekt, also ein N-dimensionales Array, zurück.

3.2 NumPy-Arrays

Bevor wir mit NumPy-Arrays[1] arbeiten können, müssen wir NumPy importieren. Da der Namensraum von NumPy sehr groß ist, empfiehlt es sich, diesen nicht mit `from numpy import *` zu importieren. Auch der Import einzelner Objekte empfiehlt sich nicht. Importiert man beispielsweise die Sinusfunktion aus NumPy, so ist weiter unten in einem Pythonskript nicht mehr ohne Weiteres erkennbar, ob es sich um den Sinus aus NumPy oder aus dem `math`-Modul handelt. Üblicherweise importiert man daher NumPy in folgender Weise:

```
In [1]: import numpy as np
```

Die Abkürzung `np` erspart dabei etwas Schreibarbeit, macht aber zugleich die Herkunft eines Objekts deutlich. Hält man sich an diese Konvention, so trägt man zur Verständlichkeit des Codes bei.

Um die Eigenschaften von Arrays zu untersuchen, müssen wir zunächst wissen, wie sich ein Array erzeugen lässt. Nachdem wir im vorigen Unterkapitel die Verwendung von Listen für die Darstellung von Matrizen diskutiert haben, mag man versucht sein, zu diesem Zweck die `append`-Methode von NumPy zu verwenden. Hiervon ist allerdings abzuraten, da bei jeder Ausführung von `append` eine Kopie des NumPy-Arrays erstellt wird. Vor allem bei großen Arrays kann bereits das Anlegen einer Kopie im Speicher sehr zeitaufwendig sein, und dies gilt ganz besonders, wenn dieser Kopiervorgang häufig durchgeführt werden muss. Bei der Arbeit mit NumPy-Arrays wird man also nach Möglichkeit immer darauf achten, dass keine unnötigen Kopien von Arrays angelegt werden. Für die Erzeugung von NumPy-Arrays bedeutet dies, dass man am besten die Größe bereits zu Beginn festlegt und dann aus den vielen zur Verfügung stehenden Methoden eine geeignete auswählt, um das Array mit Werten zu füllen.

In NumPy ist es sehr einfach, die Dokumentation nach einem bestimmten Text zu durchsuchen. Die zahlreichen Möglichkeiten, ein Array zu erzeugen, lassen sich folgendermaßen erhalten:

```
In [2]: np.lookfor('create array')
Search results for 'create array'
---------------------------------
```

[1] Wir verwenden im Folgenden das englische Wort *Array*, um damit den `ndarray`-Datentyp aus NumPy zu bezeichnen. Ein Grund dafür, nicht von Matrizen zu sprechen, besteht darin, dass sich Arrays nicht notwendigerweise wie Matrizen verhalten. So entspricht das Produkt von zwei Arrays im Allgemeinen nicht dem Matrixprodukt.

```
numpy.array
    Create an array.
numpy.memmap
    Create a memory-map to an array stored in a *binary* file on disk.
numpy.diagflat
    Create a two-dimensional array with the flattened input as a diagonal.
numpy.fromiter
    Create a new 1-dimensional array from an iterable object.
...
```

Dabei wurde hier nur ein Teil der Ausgabe dargestellt. Gleich der erste Eintrag verrät uns, wie man aus einer Liste von Listen ein Array erzeugen kann. Details hierzu erhält man bei Bedarf wie üblich mit `help(np.array)` oder alternativ mit `np.info(np.array)`.

```
In [3]: matrix = [[0, 1, 2],
   ...:           [3, 4, 5],
   ...:           [6, 7, 8]]

In [4]: myarray = np.array(matrix)

In [5]: myarray
Out[5]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

In [6]: type(myarray)
Out[6]: numpy.ndarray
```

Ein Array besitzt als wesentliche Bestandteile die Daten im eigentlichen Sinne, also die Werte der einzelnen Matrixelemente, sowie Information darüber, wie auf ein spezifisches Matrixelement zugegriffen werden kann. Die Daten sind im Speicher einfach hintereinander, also in eindimensionaler Form, abgelegt. Dabei gibt es die Möglichkeit, die Matrix zeilenweise oder spaltenweise abzuspeichern. Ersteres wird von der Programmiersprache C verwendet, während die zweite Variante von Fortran verwendet wird.

Nachdem die Daten strukturlos im Speicher abgelegt sind, müssen `ndarray`-Objekte, wie schon erwähnt, neben den Daten auch Informationen darüber besitzen, wie auf einzelne Matrixelemente zugegriffen wird. Auf diese Weise lässt sich sehr leicht die Adresse der Daten eines Matrixelements bestimmen. Zudem ist es möglich, die gleichen Daten im Speicher auf verschiedene Weise anzusehen. Damit ist es häufig möglich, unnötige Kopiervorgänge im Speicher zu vermeiden. Wie weiter oben schon angedeutet ist dies von großer Bedeutung für die Effizienz des Programms. Aus diesem Grunde ist es wichtig zu wissen, ob NumPy im Einzelfall nur eine andere Sicht auf die Daten zur Verfügung stellt oder tatsächlich ein neues Array erzeugt.

Um die Informationen über die Struktur eines Arrays besser zu verstehen, definieren wir uns eine Funktion, die einige Attribute des Arrays ausgibt.

```
In [7]: def array_attributes(a):
   ...:     for attr in ('ndim', 'size', 'itemsize', 'dtype', 'shape', 'strides'):
   ...:         print('{:8s}: {}'.format(attr, getattr(a, attr)))
```

Zum Experimentieren mit Arrays ist die `arange`-Methode sehr praktisch, die ähnlich wie das uns bereits bekannte `range` eine Reihe von Zahlen erzeugt, nun jedoch in einem Array.

```
In [8]: matrix = np.arange(16)

In [9]: matrix
Out[9]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])

In [10]: array_attributes(matrix)
ndim    : 1
size    : 16
itemsize: 8
```

```
dtype   : int64
shape   : (16,)
strides : (8,)
```

Das Attribut `ndim` gibt an, dass wir es mit einem eindimensionalen Array zu tun haben, während das Attribut `size` anzeigt, dass das Array insgesamt 16 Elemente besitzt. Jedes Element besitzt den Datentyp (`dtype`) `int64`. Es handelt sich also um 64-Bit-Integers, die eine Größe von 8 Byte (`itemsize`) besitzen. Die Attribute können wir auch direkt in der üblichen objektorientierten Schreibweise ansprechen. Zum Beispiel gibt

```
In [11]: matrix.nbytes
Out[11]: 128
```

den Speicherplatzbedarf des Arrays in Bytes an.

Für Arrays kommen eine ganze Reihe verschiedener Datentypen in Frage, zum Beispiel Integers verschiedener Länge (`int8`, `int16`, `int32`, `int64`) oder auch ohne Vorzeichen (`uint8`, ...), Gleitkommazahlen (`float16`, `float32`, `float64`), komplexe Zahlen (`complex64`, `complex128`), Wahrheitswerte (`bool8`) und sogar Unicode-Strings als nichtnumerischer Datentyp. Wenn der Datentyp nicht angegeben oder durch die Konstruktion des Arrays bestimmt ist, werden die im jeweiligen System standardmäßig verwendeten Gleitkommazahlen herangezogen, also meistens `float64`. Bei Integers ist zu beachten, dass es im Gegensatz zu Python-Integers wegen der endlichen Länge zu einem Überlauf kommen kann, wie das folgende Beispiel demonstriert.

```
In [12]: np.arange(1, 160, 10, dtype=np.int8)
Out[12]:
array([  1,  11,  21,  31,  41,  51,  61,  71,  81,  91, 101,
       111, 121, -125, -115, -105], dtype=int8)
```

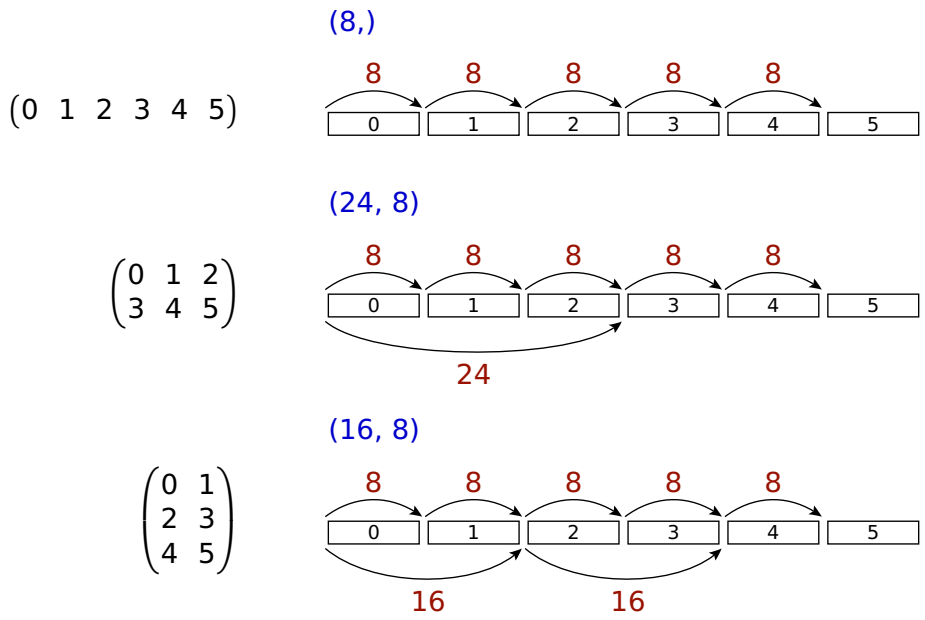
? Wie kann man diese Ausgabe verstehen?

Besonders interessant sind die beiden Attribute `shape` und `strides`. Der Wert des Attributs `shape`, in unserem Beispiel das Tupel `(16,)`, gibt die Zahl der Elemente in der jeweiligen Dimension an. Um dies besser zu verstehen, ändern wir dieses Attribut ab, wobei darauf zu achten ist, dass die Zahl der Elemente des Arrays erhalten bleibt. Wir wandeln das eindimensionale Array mit 16 Elementen in ein 4×4-Array um.

```
In [13]: matrix.shape = (4, 4)

In [14]: matrix
Out[14]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])

In [15]: matrix.strides
Out[15]: (32, 8)
```

Dabei wird deutlich, dass nicht nur die Form (`shape`) modifiziert wurde, sondern auch aus dem Tupel `(8,)` des Attributs `strides`[2] das Tupel `(32, 8)` wurde. Die *strides* geben an, um wieviel Bytes man weitergehen muss, um zum nächsten Element in dieser Dimension zu gelangen. Die folgende Abbildung zeigt dies an einem kleinen Array.

[2] Das englische Wort *stride* bedeutet Schritt.



Greifen wir speziell den mittleren Fall mit den *strides* `(24, 8)` heraus. Bewegt man sich in einer Zeile der Matrix von Element zu Element, so muss man im Speicher jeweils um 8 Bytes weitergehen, wenn ein Datentyp `int64` vorliegt. Entlang einer Spalte beträgt die Schrittweite dagegen 24 Bytes.

? Wie verändern sich die *strides* in dem 16-elementigen Array `np.arange(16)`, wenn man einen `shape` von `(2, 2, 2, 2)` wählt?

Für die Anwendung ist es wichtig zu wissen, dass die Manipulation der Attribute `shape` und `strides` nicht die Daten im Speicher verändert. Es wird also nur eine neue Sicht auf die vorhandenen Daten vermittelt. Dies ist insofern von Bedeutung als das Kopieren von größeren Datenmengen durchaus mit einem nicht unerheblichen Zeitaufwand verbunden sein kann.

Um uns davon zu überzeugen, dass tatsächlich kein neues Array erzeugt wird, generieren wir nochmals ein eindimensionales Array und daraus mit Hilfe von `reshape` ein zweidimensionales Array.

```
In [16]: m1 = np.arange(16)

In [17]: m1
Out[17]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])

In [18]: m2 = m1.reshape(4, 4)

In [19]: m2
Out[19]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Nun ändern wir das erste Element in dem eindimensionalen Array ab und stellen in der Tat fest, dass sich diese Änderung auch auf das zweidimensionale Array auswirkt.

```
In [20]: m1[0] = 99

In [21]: m1
Out[21]: array([99,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])

In [22]: m2
Out[22]:
array([[99,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
```

```
       [12, 13, 14, 15]])
```

Eine Matrix lässt sich auch transponieren, ohne dass Matrixelemente im Speicher hin und her kopiert werden müssen. Stattdessen werden nur die beiden Werte der *strides* vertauscht.

```
In [23]: m2.strides
Out[23]: (32, 8)

In [24]: m2.T
Out[24]:
array([[99,  4,  8, 12],
       [ 1,  5,  9, 13],
       [ 2,  6, 10, 14],
       [ 3,  7, 11, 15]])

In [25]: m2.T.strides
Out[25]: (8, 32)
```

Obwohl die Daten im Speicher nicht verändert wurden, kann man jetzt mit der transponierten Matrix arbeiten.

Mit Hilfe der Attribute `shape` und `strides` lässt sich die Sicht auf ein Array auf sehr flexible Weise festlegen. Allerdings ist der Benutzer selbst für die Folgen verantwortlich, wie der zweite Teil des folgenden Beispiels zeigt. Dazu gehen wir zu unserem ursprünglichen 4×4-Array zurück und verändern das Attribut `strides` mit Hilfe der `as_strided`-Methode.

```
In [26]: matrix = np.arange(16).reshape(4, 4)

In [27]: matrix1 = np.lib.stride_tricks.as_strided(matrix, strides=(16, 16))

In [28]: matrix1
Out[28]:
array([[ 0,  2,  4,  6],
       [ 2,  4,  6,  8],
       [ 4,  6,  8, 10],
       [ 6,  8, 10, 12]])

In [29]: matrix2 = np.lib.stride_tricks.as_strided(matrix, shape=(4, 4),␣
↪strides=(16, 4))

In [30]: matrix2
Out[30]:
array([[           0,  4294967296,           1,  8589934592],
       [           2, 12884901888,           3, 17179869184],
       [           4, 21474836480,           5, 25769803776],
       [           6, 30064771072,           7, 34359738368]])
```

Im ersten Fall ist der Wert der *strides* gerade das Doppelte der Datenbreite, so dass in einer Zeile von einem Wert zum nächsten jeweils ein Wert im Array übersprungen wird. Beim Übergang von einer Zeile zur nächsten wird gegenüber dem Beginn der vorherigen Zeile auch nur um zwei Werte vorangeschritten, so dass sich das gezeigte Resultat ergibt.

Im zweiten Beispiel wurde ein *stride* gewählt, der nur die Hälfte einer Datenbreite beträgt. Der berechnete Beginn eines neuen Werts im Speicher liegt damit nicht an einer Stelle, die einem tatsächlichen Beginn eines Werts entspricht. Python interpretiert dennoch die erhaltene Information und erzeugt so das obige Array. In unserem Beispiel erreicht man bei jedem zweiten Wert wieder eine korrekte Datengrenze. Die Manipulation von *strides* erfordert also eine gewisse Sorgfalt, und man ist für eventuelle Fehler selbst verantwortlich.

3.3 Erzeugung von NumPy-Arrays

NumPy-Arrays lassen sich je nach Bedarf auf verschiedene Arten erzeugen. Die Basis bildet die `ndarray`-Methode, auf die man immer zurückgreifen kann. In den meisten Fällen wird es aber praktischer sein, eine der spezialisierteren Methoden zu verwenden, die wir im Folgenden besprechen wollen.

Um ein mit Nullen aufgefülltes 2×2-Array zu erzeugen, geht man folgendermaßen vor:

```
In [1]: matrix1 = np.zeros((2, 2))

In [2]: matrix1, matrix1.dtype
Out[2]:
(array([[ 0.,  0.],
        [ 0.,  0.]]), dtype('float64'))
```

Das Tupel im Argument gibt dabei die Form des Arrays vor. Wird der Datentyp der Einträge nicht weiter spezifiziert, so werden Gleitkommazahlen mit einer Länge von 8 Bytes verwendet. Man kann aber auch explizit zum Beispiel Integereinträge verlangen:

```
In [3]: np.zeros((2, 2), dtype=np.int)
Out[3]:
array([[0, 0],
       [0, 0]])
```

Neben der `zeros`-Funktion gibt es auch noch die `empty`-Funktion, die zwar den benötigten Speicherplatz zur Verfügung stellt, diesen jedoch nicht initialisiert. Im Allgemeinen werden also die Arrayelemente von den hier im Beispiel gezeigten abweichen.

```
In [4]: np.empty((3, 3))
Out[4]:
array([[  6.91153891e-310,   2.32617410e-316,   6.91153265e-310],
       [  6.91153265e-310,   6.91153265e-310,   6.91153265e-310],
       [  6.91153265e-310,   6.91153265e-310,   3.95252517e-322]])
```

Die `empty`-Funktion sollte also nur verwendet werden, wenn die Arrayelemente später noch belegt werden.

Will man alle Elemente eines Arrays mit einem konstanten Wert ungleich Null füllen, so kann man `ones` verwenden und das sich ergebende Array mit einem Faktor multiplizieren.

```
In [5]: 2*np.ones((2, 3))
Out[5]:
array([[ 2.,  2.,  2.],
       [ 2.,  2.,  2.]])
```

Häufig benötigt man eine Einheitsmatrix, die man mit Hilfe von `identity` erhält:

```
In [6]: np.identity(3)
Out[6]:
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Hierbei wird immer eine Diagonalmatrix erzeugt. Will man dies nicht, so kann man `eye` verwenden, das nicht nur nicht quadratische Arrays erzeugen kann, sondern auch die Diagonale nach oben oder unten verschieben lässt.

```
In [7]: np.eye(2, 4)
Out[7]:
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.]])
```

Zu beachten ist hier, dass die Form des Arrays nicht als Tupel vorgegeben wird, da ohnehin nur zweidimensionale Arrays erzeugt werden können. Lässt man das zweite Argument weg, so wird ein quadratisches Array erzeugt.

Will man die Diagonaleinträge verschieben, so gibt man dies mit Hilfe des Parameters `k` an:

```
In [8]: np.eye(4, k=1)-np.eye(4, k=-1)
Out[8]:
array([[ 0.,  1.,  0.,  0.],
       [-1.,  0.,  1.,  0.],
       [ 0., -1.,  0.,  1.],
       [ 0.,  0., -1.,  0.]])
```

Eine Diagonalmatrix mit unterschiedlichen Einträgen lässt sich aus einem eindimensionalen Array folgendermaßen erzeugen:

```
In [9]: np.diag([1, 2, 3, 4])
Out[9]:
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

Dabei lässt sich wie bei der `eye`-Funktion die Diagonale verschieben.

```
In [10]: np.diag([1, 2, 3, 4], k=1)
Out[10]:
array([[0, 1, 0, 0, 0],
       [0, 0, 2, 0, 0],
       [0, 0, 0, 3, 0],
       [0, 0, 0, 0, 4],
       [0, 0, 0, 0, 0]])
```

Umgekehrt kann man mit der `diag`-Funktion auch die Diagonalelemente eines zweidimensionalen Arrays extrahieren.

```
In [11]: matrix = np.arange(16).reshape(4, 4)

In [12]: matrix
Out[12]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])

In [13]: np.diag(matrix)
Out[13]: array([ 0,  5, 10, 15])
```

Lassen sich die Arrayeinträge als Funktion der Indizes ausdrücken, so kann man die `fromfunction`-Funktion verwenden, wie in dem folgenden Beispiel zu sehen ist, das eine Multiplikationstabelle erzeugt.

```
In [14]: np.fromfunction(lambda i, j: (i+1)*(j+1), (6, 6), dtype=np.int)
Out[14]:
array([[ 1,  2,  3,  4,  5,  6],
       [ 2,  4,  6,  8, 10, 12],
       [ 3,  6,  9, 12, 15, 18],
       [ 4,  8, 12, 16, 20, 24],
       [ 5, 10, 15, 20, 25, 30],
       [ 6, 12, 18, 24, 30, 36]])
```

Diese Funktion ist nicht auf zweidimensionale Arrays beschränkt.

Bei der Konstruktion von Arrays sind auch Funktionen interessant, die als Verallgemeinerung der in Python eingebauten Funktion `range` angesehen werden können. Ihr Nutzen ergibt sich vor allem aus der Tatsache, dass man gewissen Funktionen, den universellen Funktionen oder ufuncs in NumPy, die wir im Abschnitt *Universelle Funktionen* besprechen werden, ganze Arrays als Argumente übergeben kann. Damit wird eine besonders effiziente Auswertung dieser Funktionen möglich.

Eindimensionale Arrays lassen sich mit Hilfe von `arange`, `linspace` und `logspace` erzeugen:

```
In [15]: np.arange(1, 2, 0.1)
Out[15]: array([ 1. ,  1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9])
```

Ähnlich wie bei `range` erzeugt `arange` aus der Angabe eines Start- und eines Endwerts sowie einer Schrittweite eine Folge von Werten. Allerdings können diese auch Gleitkommazahlen sein. Zudem wird statt einer Liste ein Array erzeugt. Wie bei `range` ist der Endwert hierin nicht enthalten. Allerdings können Rundungsfehler zu unerwarteten Effekten führen.

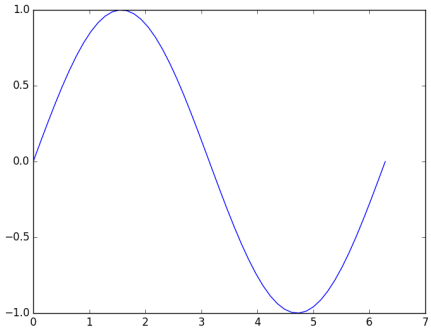
```
In [16]: np.arange(1, 1.5, 0.1)
Out[16]: array([ 1. ,  1.1,  1.2,  1.3,  1.4])

In [17]: np.arange(1, 1.6, 0.1)
Out[17]: array([ 1. ,  1.1,  1.2,  1.3,  1.4,  1.5,  1.6])

In [18]: np.arange(1, 1.601, 0.1)
Out[18]: array([ 1. ,  1.1,  1.2,  1.3,  1.4,  1.5,  1.6])
```

Dieses Problem kann man umgehen, wenn man statt der Schrittweite eine Anzahl von Punkten in dem gegebenen Intervall vorgibt. Dafür ist `linspace` eine geeignete Funktion, sofern die Schrittweite konstant sein soll. Bei Bedarf kann man sich neben dem Array auch noch die Schrittweite ausgeben lassen. Benötigt man eine logarithmische Skala, so verwendet man `logspace`, das den Exponenten linear zwischen einem Start- und einem Endwert verändert. Die Basis ist standardmäßig 10, sie kann aber durch Setzen des Parameters `base` an spezielle Erfordernisse angepasst werden.

```
In [19]: np.linspace(1, 2, 11)
Out[19]: array([ 1. ,  1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9,  2. ])

In [20]: np.linspace(1, 2, 4, retstep=True)
Out[20]:
(array([ 1.        ,  1.33333333,  1.66666667,  2.        ]),
 0.3333333333333333)

In [21]: np.logspace(0, 3, 6)
Out[21]:
array([    1.        ,     3.98107171,    15.84893192,    63.09573445,
         251.18864315,  1000.        ])

In [22]: np.logspace(0, 3, 4, base=2)
Out[22]: array([ 1.,  2.,  4.,  8.])
```

Gerade bei der graphischen Darstellung von Funktionen sind `linspace` und `logspace` besonders nützlich. Im folgenden Beispiel verwenden wir die matplotlib-Bibliothek, die im Abschnitt *Erstellung von Grafiken mit matplotlib* besprochen wird.

```
In [23]: import matplotlib.pyplot as plt

In [24]: x = np.linspace(0, 2*np.pi)

In [25]: y = np.sin(x)

In [26]: plt.plot(x, y)
Out[26]: [<matplotlib.lines.Line2D at 0x7f3ad50b76a0>]
```



Möchte man eine Funktion auf einem Gitter auswerten und benötigt man dazu separate Arrays für die x- und y-Werte, so hilft `meshgrid` weiter.

```
In [27]: xvals, yvals = np.meshgrid([-1, 0, 1], [2, 3, 4])

In [28]: xvals
Out[28]:
array([[-1,  0,  1],
       [-1,  0,  1],
       [-1,  0,  1]])

In [29]: yvals
Out[29]:
array([[2, 2, 2],
       [3, 3, 3],
       [4, 4, 4]])
```

In diesem Zusammenhang sind auch die Funktionen `mgrid` und `ogrid` von Interesse, die wir im Abschnitt *Universelle Funktionen* besprechen werden, nachdem wir die Adressierung von Arrays genauer angesehen haben.

Zur graphischen Darstellung von Daten ist es häufig erforderlich, die Daten zunächst aus einer Datei einzulesen und in einem Array zu speichern. Neben wir an, wir hätten eine Datei `x_von_t.dat` mit folgendem Inhalt:

```
# Zeit  Ort
   0.0  0.0
   0.1  0.1
   0.2  0.4
   0.3  0.9
```

Hierbei zeigt das #-Zeichen in der ersten Zeile an, dass es sich um eine Kommentarzeile handelt, die nicht in das Array übernommen werden soll. Unter Verwendung von `loadtxt` kann man die Daten nun einlesen:

```
In [30]: np.loadtxt("x_von_t.dat")
Out[30]:
array([[ 0. ,  0. ],
       [ 0.1,  0.1],
       [ 0.2,  0.4],
       [ 0.3,  0.9]])
```

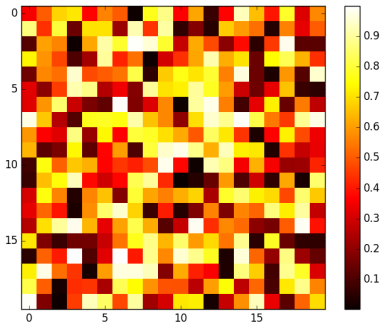
Bei der `loadtxt`-Funktion lassen sich zum Beispiel das Kommentarzeichen oder die Trennzeichen zwischen Einträgen konfigurieren. Noch wesentlich flexibler ist `genfromtxt`, das es unter anderem erlaubt, Spaltenüberschriften aus der Datei zu entnehmen oder mit fehlenden Einträgen umzugehen. Für Details wird auf die zugehörige Dokumentation verwiesen.

Abschließend betrachten wir noch kurz die Erzeugung von Zufallszahlen, die man zum Beispiel für Simulationszwecke benötigt. Statt viele einzelne Zufallszahlen zu erzeugen ist es häufig effizienter, gleich ein ganzes Array mit Zufallszahlen zu füllen. Im folgenden Beispiel erzeugen wir ein Array mit zehn im Intervall zwischen 0 und 1 gleich verteilten Pseudozufallszahlen. Reproduzierbar werden die Zahlenwerte, wenn zunächst ein Startwert für die Berechnung, ein *seed*, gesetzt wird.

```
In [31]: np.random.rand(2, 5)
Out[31]:
array([[ 0.99281469,  0.90376223,  0.81096671,  0.33726814,  0.34463236],
       [ 0.74234766,  0.05862623,  0.49005243,  0.73496906,  0.21421244]])

In [32]: np.random.rand(2, 5)
Out[32]:
array([[ 0.51071925,  0.11952145,  0.12714712,  0.98081263,  0.05736099],
       [ 0.35101524,  0.86407263,  0.80264858,  0.36629556,  0.59562485]])

In [33]: np.random.seed(1234)

In [34]: np.random.rand(2, 5)
Out[34]:
array([[ 0.19151945,  0.62210877,  0.43772774,  0.78535858,  0.77997581],
       [ 0.27259261,  0.27646426,  0.80187218,  0.95813935,  0.87593263]])

In [35]: np.random.rand(2, 5)
Out[35]:
array([[ 0.19151945,  0.62210877,  0.43772774,  0.78535858,  0.77997581],
       [ 0.27259261,  0.27646426,  0.80187218,  0.95813935,  0.87593263]])
```

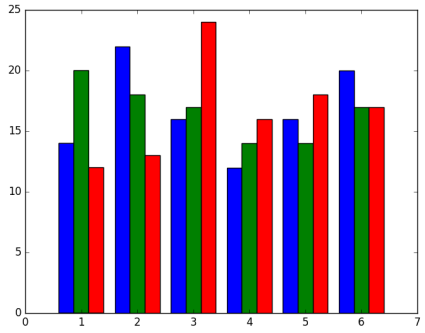
Die Zufälligkeit der Daten lässt sich graphisch darstellen.

```
In [36]: data = np.random.rand(20, 20)

In [37]: plt.imshow(data, cmap=plt.cm.hot, interpolation='none')
Out[37]: <matplotlib.image.AxesImage at 0x7f4eacf147b8>

In [38]: plt.colorbar()
Out[38]: <matplotlib.colorbar.Colorbar at 0x7f4eac6a60f0>
```



Als Anwendung betrachten wir drei Realisierungen von hundert Würfen eines Würfels. Dazu erzeugen wir mit `randint(1, 7)` ganzzahlige Pseudozufallszahlen zwischen 1 und 6 in einem zweidimensionalen Array der Form `(100, 3)`. Diese drei Spalten zu je 100 Zahlen werden jeweils als Histogramm dargestellt.

```
In [14]: wuerfe = np.random.randint(1, 7, (100, 3))

In [15]: plt.hist(wuerfe, np.linspace(0.5, 6.5, 7))
Out[15]:
([array([ 14.,  22.,  16.,  12.,  16.,  20.]),
  array([ 20.,  18.,  17.,  14.,  14.,  17.]),
  array([ 12.,  13.,  24.,  16.,  18.,  17.])],
 array([ 0.5,  1.5,  2.5,  3.5,  4.5,  5.5,  6.5]),
 <a list of 3 Lists of Patches objects>)
```



3.4 Adressierung von NumPy-Arrays

Die Adressierungsmöglichkeiten für NumPy-Arrays basieren auf der so genannten *slice*-Syntax, die wir von Python-Listen her kennen und uns hier noch einmal kurz in Erinnerung rufen wollen. Einen Ausschnitt aus einer Liste, ein *slice*, erhält man durch die Notation `[start:stop:step]`. Hierbei werden ausgehend von dem Element mit dem Index `start` die Elemente bis vor das Element mit dem Index `stop` mit einer Schrittweite `step` ausgewählt. Wird die Schrittweite nicht angegeben, so nimmt `step` den Defaultwert `1` an. Negative Schrittweiten führen in der Liste von hinten nach vorne. Fehlen `start` und/oder `stop` so beginnen die ausgewählten Elemente mit dem ersten Element bzw. enden mit dem letzten Element. Negative Indexwerte werden vom Ende der Liste her genommen. Das letzte Element kann also mit dem Index `-1`, das vorletzten Element mit dem Index `-2` usw. angesprochen werden. Diese Indizierung funktioniert so auch für NumPy-Arrays wie die folgenden Beispiele zeigen.

```
In [1]: a = np.arange(10)

In [2]: a
Out[2]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [3]: a[:]
Out[3]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [4]: a[::2]
Out[4]: array([0, 2, 4, 6, 8])

In [5]: a[1:4]
Out[5]: array([1, 2, 3])

In [6]: a[6:-2]
Out[6]: array([6, 7])

In [7]: a[::-1]
Out[7]: array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

Für mehrdimensionale Arrays wird die Notation direkt verallgemeinert. Im Gegensatz zu der im Abschnitt *Python-Listen und Matrizen* beschriebenen Notation für Listen von Listen werden hier die diversen Indexangaben durch Kommas getrennt zusammengefasst. Einige Beispiele für zweidimensionale Arrays sollen das illustrieren.

```
In [8]: a = np.arange(36).reshape(6, 6)

In [9]: a
Out[9]:
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29],
```

```
       [30, 31, 32, 33, 34, 35]])

In [10]: a[:, :]
Out[10]:
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35]])

In [11]: a[2:4, 2:4]
Out[11]:
array([[14, 15],
       [20, 21]])

In [12]: a[2:4, 3:5]
Out[12]:
array([[15, 16],
       [21, 22]])

In [13]: a[::2, ::2]
Out[13]:
array([[ 0,  2,  4],
       [12, 14, 16],
       [24, 26, 28]])

In [14]: a[2::2, ::2]
Out[14]:
array([[12, 14, 16],
       [24, 26, 28]])

In [15]: a[2:4]
Out[15]:
array([[12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])
```

Wie das letzte Beispiel zeigt, ergänzt NumPy bei fehlenden Indexangaben jeweils einen Doppelpunkt, so dass alle Elemente ausgewählt werden, die mit den explizit gemachten Indexangaben konsistent sind.

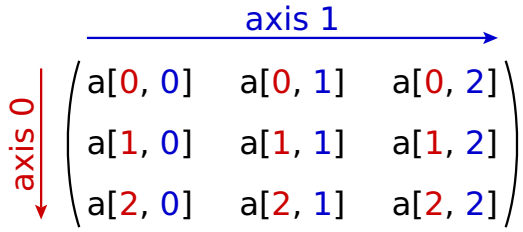
Will man eine Spalte (oder auch eine Zeile) in einer zweidimensionalen Array auswählen, so hat man zwei verschiedene Möglichkeiten:

```
In [16]: a[:, 0:1]
Out[16]:
array([[ 0],
       [ 6],
       [12],
       [18],
       [24],
       [30]])

In [17]: a[:, 0]
Out[17]: array([ 0,  6, 12, 18, 24, 30])
```

Im ersten Fall sorgt die für beide Dimensionen vorhandene Indexnotation dafür, dass ein zweidimensionales Array erzeugt wird, das die Elemente der ersten Spalte enthält. Im zweiten Fall wird für die zweite Dimension ein fester Index angegeben, so dass nun ein eindimensionales Array erzeugt wird, die wiederum aus den Elementen der ersten Spalte besteht.

In einigen NumPy-Methoden gibt es einen Parameter `axis`, der die Richtung in dem Array angibt, in der die Methode ausgeführt werden soll. Die Achsennummer ergibt sich aus der Position der zugehörigen Indexangabe. Wie man aus den obigen Beispielen entnehmen kann, verläuft die Achse 0 von oben nach unten, während die

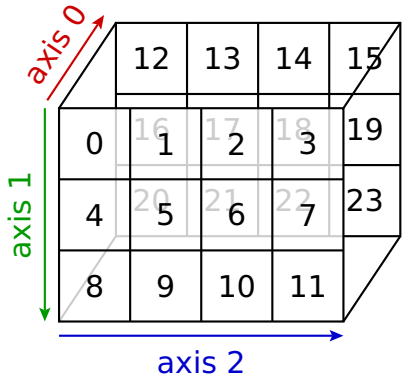
Achse 1 von links nach rechts verläuft. Dies wird auch durch die folgende Abbildung veranschaulicht.



Das Aufsummieren von Elementen unserer Beispielmatrix erfolgt dann mit Hilfe der `sum`-Methode entweder von oben nach unten, von links nach rechts oder über alle Elemente.

```
In [18]: a.sum(axis=0)
Out[18]: array([ 90,  96, 102, 108, 114, 120])

In [19]: a.sum(axis=1)
Out[19]: array([ 15,  51,  87, 123, 159, 195])

In [20]: a.sum()
Out[20]: 630
```

Zur Verdeutlichung betrachten wir noch ein dreidimensionales Array das im Folgenden graphisch dargestellt ist.




```
In [21]: b = np.arange(24).reshape(2, 3, 4)

In [22]: b
Out[22]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])

In [23]: b[0:1]
Out[23]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]]])

In [24]: b[:, 0:1]
Out[24]:
array([[[ 0,  1,  2,  3]],

       [[12, 13, 14, 15]]])

In [25]: b[:, :, 0:1]
```

```
Out[25]:
array([[[ 0],
        [ 4],
        [ 8]],

       [[12],
        [16],
        [20]]])

In [26]: b[..., 0:1]
Out[26]:
array([[[ 0],
        [ 4],
        [ 8]],

       [[12],
        [16],
        [20]]])
```

Man sieht hier deutlich, wie je nach Wahl der Achse ein entsprechender Schnitt durch das als Würfel vorstellbare Array gemacht wird. Das letzte Beispiel zeigt die Benutzung des Auslassungszeichens `...` (im Englischen *ellipsis* genannt). Es steht für die Anzahl von Doppelpunkten, die nötig sind, um die Indizes für alle Dimensionen zu spezifizieren. Allerdings funktioniert dies nur beim ersten Auftreten des Auslassungszeichens, da sonst nicht klar ist, wie viele Indexspezifikation für jedes Auslassungszeichen einzusetzen sind. Alle weiteren Auslassungszeichen werden daher durch einen einzelnen Doppelpunkt ersetzt.

Weiter oben hatten wir in einem Beispiel gesehen, dass die Angabe eines festen Index die Dimension des Arrays effektiv um Eins vermindert. Umgekehrt ist es auch möglich, eine zusätzliche Dimension der Länge Eins hinzuzufügen. Hierzu dient `newaxis`, das an der gewünschten Stelle als Index eingesetzt werden kann. Die folgenden Beispiele zeigen, wie aus einem eindimensionalen Array so zwei verschiedene zweidimensionale Arrays konstruiert werden können.

```
In [27]: c = np.arange(5)

In [28]: c
Out[28]: array([0, 1, 2, 3, 4])

In [29]: c[:, np.newaxis]
Out[29]:
array([[0],
       [1],
       [2],
       [3],
       [4]])

In [30]: c[np.newaxis, :]
Out[30]: array([[0, 1, 2, 3, 4]])
```

Eine Anwendung hiervon werden wir weiter unten in diesem Kapitel kennenlernen, wenn wir uns mit der Erweiterung von Arrays auf eine Zielgröße, dem so genannten *broadcasting* beschäftigen.

Zunächst wollen wir aber noch eine weitere Indizierungsmethode, das so genannte *fancy indexing*, ansprechen. Obwohl es sich hierbei um ein sehr flexibles und mächtiges Verfahren handelt, sollte man bedenken, dass hier immer eine Kopie des Arrays erzeugt wird und nicht einfach nur eine neue Sicht auf bereits vorhandene Daten. Da Letzteres effizienter ist, sollte man *fancy indexing* in erster Linie in Situationen einsetzen, in denen das normale Indizieren nicht ausreicht.

Beim *fancy indexing* werden die möglichen Indizes als Arrays oder zum Beispiel als Liste, nicht jedoch als Tupel, angegeben. Die Elemente können dabei *Integer* oder *Boolean* sein. Beginnen wir mit dem ersten Fall, wobei wir zunächst von einem eindimensionalen Array ausgehen.

```
In [31]: a = np.arange(10, 20)

In [32]: a[[0, 3, 0, 5]]
Out[32]: array([10, 13, 10, 15])

In [33]: a[np.array([[0, 2], [1, 4]])]
Out[33]:
array([[10, 12],
       [11, 14]])
```

Im ersten Fall werden einzelne Arrayelemente durch Angabe der Indizes ausgewählt, wobei auch Wiederholungen sowie eine nichtmonotone Wahl von Indizes möglich sind. Sind die Indizes als Array angegeben, so wird ein Array der gleichen Form erzeugt.

Bei der Auswahl von Elementen aus einem mehrdimensionalen Arrays muss man gegebenenfalls weitere Indexlisten oder -arrays angeben.

```
In [34]: a = np.arange(16).reshape(4, 4)

In [35]: a
Out[35]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])

In [36]: a[[0, 1, 2]]
Out[36]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [37]: a[[0, 1, 2], [1, 2, 3]]
Out[37]: array([ 1,  6, 11])
```

Interessant ist die Verwendung von Indexarrays mit Elementen vom Typ *Boolean*. Ein solches Indexarray lässt sich zum Beispiel mit Hilfe einer logischen Operation auf einem Array erzeugen, wie das folgende Beispiel demonstriert. Eine Reihe von Zufallszahlen soll dabei bei einem Schwellenwert nach unten abgeschnitten werden.

```
1 threshold = 0.3
2 a = np.random.random(12)
3 print a
4 print "-"*30
5 indexarray = a < threshold
6 print indexarray
7 print "-"*30
8 a[indexarray] = threshold
9 print a
```

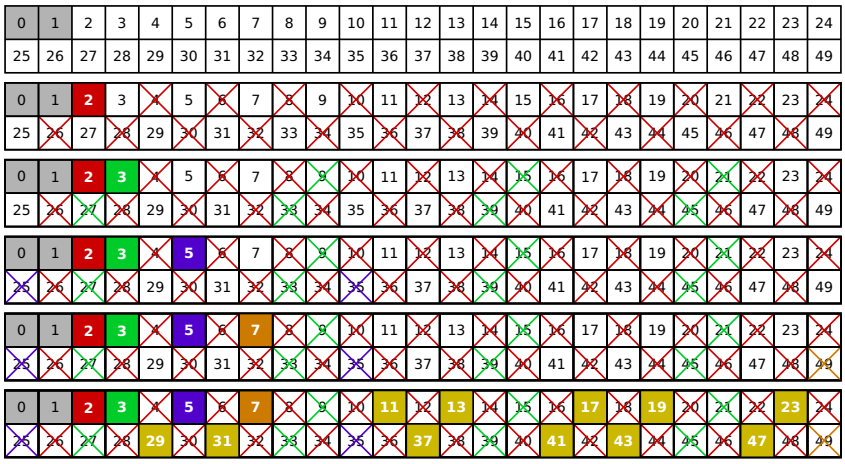
Damit ergibt sich beispielsweise die folgende Ausgabe:

```
[ 0.11859559  0.49034494  0.08552061  0.69204077  0.18406457  0.06819091
  0.36785529  0.16873423  0.44615435  0.57774615  0.54327126  0.57381642]
------------------------------
[ True False  True False  True  True False  True False False False False]
------------------------------
[ 0.3         0.49034494  0.3         0.69204077  0.3         0.3
  0.36785529  0.3         0.44615435  0.57774615  0.54327126  0.57381642]
```

In Zeile 5 wird ein Array `indexarray` erzeugt, das an den Stellen, an denen die Elemente des Arrays `a` kleiner als der Schwellwert sind, den Wahrheitswert `True` besitzt. In Zeile 8 werden die auf diese Weise indizierten Elemente dann auf den Schwellwert gesetzt. Es sei noch angemerkt, dass sich diese Funktionalität auch direkt mit

der `clip`-Funktion erreichen lässt.

Als Anwendungsbeispiel für die Indizierung von Arrays durch *slicing* und durch *fancy indexing* betrachten wir das Sieb des Eratosthenes zur Bestimmung von Primzahlen. Die folgende Abbildung illustriert das Prinzip.



Ausgehend von der Zwei als kleinster Primzahl werden in aufsteigender Reihenfolge für alle Primzahlen deren Vielfache als Nichtprimzahlen identifiziert. Dies ist in der Abbildung durch Kreuze in der entsprechenden Farbe angedeutet. Beim Durchstreichen genügt es, mit dem Quadrat der jeweiligen Primzahl zu beginnen, da kleinere Vielfache bereits bei der Betrachtung einer kleineren Primzahl berücksichtigt wurden. So werden nacheinander alle Zahlen in der Liste identifiziert, die keine Primzahlen sind. Übrig bleiben somit die gesuchten Primzahlen. Eine Realisierung dieses Verfahrens unter Verwendung der Möglichkeiten von NumPy könnte folgendermaßen aussehen.

```
1 nmax = 50
2 integers = np.arange(nmax)
3 is_prime = np.ones(nmax, dtype=bool)
4 is_prime[:2] = False
5 for j in range(2, int(np.sqrt(nmax))+1):
6     if is_prime[j]:
7         is_prime[j*j::j] = False
8 print(integers[is_prime])
```

Als Ergebnis wird am Ende die Liste

```
[ 2  3  5  7 11 13 17 19 23 29 31 37 41 43 47]
```

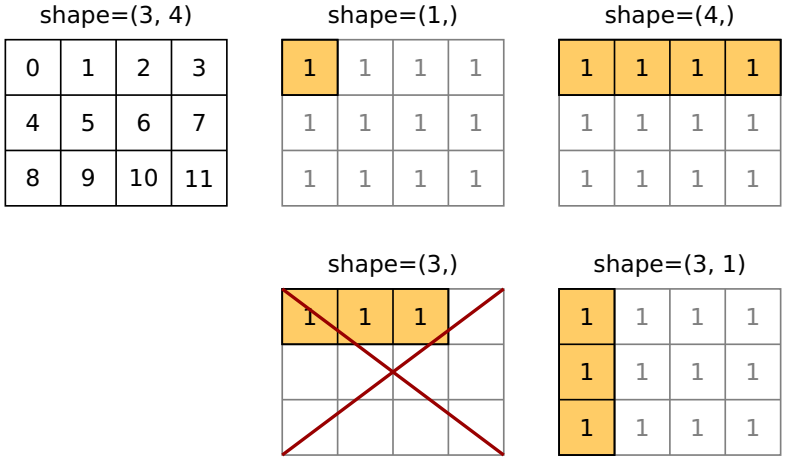
ausgegeben. Um die Indizierung leicht nachvollziehbar zu machen, enthält das Array `integers` der zu untersuchenden Zahlen auch die Null und die Eins. Nachdem in Zeile 3 zunächst alle Zahlen als potentielle Primzahlen markiert werden, wird dies in Zeile 4 für die Null und die Eins gleich wieder rückgängig gemacht. Da das Wegstreichen von Zahlen erst mit dem Quadrat einer Primzahl beginnt, müssen nur Primzahlen bis zur Wurzel aus der maximalen Zahl `nmax` betrachtet werden. In der Schleife der Zeilen 6 und 7 werden für jede dieser Primzahlen beginnend bei deren Quadrat die Vielfachen der Primzahl bis zum Ende der Liste zu Nichtprimzahlen erklärt. Die Ausgabe in Zeile 8 benutzt dann *fancy indexing* mit Hilfe des booleschen Arrays `is_prime`, um die tatsächlichen Primzahlen aus der Liste der potentiellen Primzahlen `integers` auszuwählen.

In einem Beispiel zum *fancy indexing* haben wir in der Vergleichsoperation `a < threshold` ein Array (`a`) und ein Skalar (`threshold`) miteinander verglichen. Wie kann dies funktionieren? Den Vergleich zweier Arrays derselben Form kann man sinnvoll elementweise definieren. Soll ein Array mit einem Skalar verglichen werden, so wird der Skalar von NumPy zunächst mit gleichen Elementen so erweitert, das ein Array mit der benötigten Form entsteht. Dieser als *broadcasting* bezeichnete Prozess kommt beispielsweise auch bei arithmetischen Operationen zum Einsatz. Die beiden folgenden Anweisungen sind daher äquivalent:

```
In [38]: a = np.arange(5)

In [39]: a*3
Out[39]: array([ 0,  3,  6,  9, 12])
```

```
In [40]: a*np.array([3, 3, 3, 3, 3])
Out[40]: array([ 0,  3,  6,  9, 12])
```

Broadcasting ist genau dann möglich, wenn beim Vergleich der Achsen der beiden beteiligten Arrays von der letzten Achse beginnend die Länge der Achsen jeweils gleich ist oder eine Achse die Länge Eins besitzt. Eine Achse der Länge Eins wird durch Wiederholen der Elemente im erforderlichen Umfang verlängert. Entsprechendes geschieht beim Hinzufügen von Achsen von vorne, um die Dimensionen der Arrays identisch zu machen. Die folgende Abbildung illustriert das *broadcasting*.



Hier ist ein Array der Form `(3, 4)` vorgegeben. Für ein Array der Form `(1,)` wird die Länge auf die Länge der Achse 1 des ersten Array, also 4, erweitert. Zudem wird eine weitere Achse 0 mit der gleichen Länge wie im ursprünglichen Array hinzugefügt. Geht man von einem Array der Form `(4,)` aus, so muss nur noch in gleicher Weise die Achse 0 hinzugefügt werden. Dagegen genügt ein Array der Form `(3,)` nicht den Bedingungen des *broadcasting*, da die Achse weder die Länge Eins noch die Länge der Achse 1 des ursprünglichen Arrays besitzt. Anders ist dies bei einem Array der Form `(3, 1)`, bei dem nur die Länge der Achse 1 auf 4 erhöht werden muss.

Betrachten wir abschließend noch entsprechende Codebeispiele.

```
In [41]: a = np.arange(12.).reshape(3, 4)

In [42]: a
Out[42]:
array([[  0.,   1.,   2.,   3.],
       [  4.,   5.,   6.,   7.],
       [  8.,   9.,  10.,  11.]])

In [43]: a+1.
Out[43]:
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.]])

In [44]: a+np.ones(4)
Out[44]:
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.]])

In [45]: a+np.ones(3)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-7-1b5c4daa3b16> in <module>()
----> 1 a+np.ones(3)

ValueError: operands could not be broadcast together with shapes (3,4) (3,)
```

```
In [46]: a+np.ones(3)[:, np.newaxis]
Out[46]:
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.]])

In [47]: a+np.ones(3).reshape(3, 1)
Out[47]:
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.]])
```

3.5 Universelle Funktionen

Im Abschnitt *Erzeugung von NumPy-Arrays* hatten wir in einem Beispiel bereits eine Funktion auf ein Array angewandt. Um dieses Vorgehen besser zu verstehen, importieren wir zusätzlich zum `numpy`-Paket, das in diesem Kapitel immer importiert sein sollte, noch das `math`-Modul und versuchen dann, den Sinus eines Arrays auszuwerten.

```
In [1]: import math

In [2]: math.sin(np.linspace(0, math.pi, 11))
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

<ipython console> in <module>()

TypeError: only length-1 arrays can be converted to Python scalars
```

Dabei scheitern wir jedoch, da der Sinus aus dem `math`-Modul nur mit skalaren Größen umgehen kann. Hätte unser Array nur ein Element enthalten, so wären wir noch erfolgreich gewesen. Im Beispiel hatten wir jedoch mehr als ein Element, genauer gesagt elf Elemente, und somit kommt es zu einer `TypeError`-Ausnahme.

Den Ausweg bietet in diesem Fall das `numpy`-Paket selbst, das neben einer ganzen Reihe weiterer Funktionen auch eine eigene Sinusfunktion zur Verfügung stellt. Diese ist in der Lage, mit Arrays beliebiger Dimension umzugehen. Dabei wird die Funktion elementweise angewandt und wieder ein Array der ursprünglichen Form erzeugt.

```
In [3]: np.sin(np.linspace(0, math.pi, 11))
Out[3]:
array([  0.00000000e+00,   3.09016994e-01,   5.87785252e-01,
         8.09016994e-01,   9.51056516e-01,   1.00000000e+00,
         9.51056516e-01,   8.09016994e-01,   5.87785252e-01,
         3.09016994e-01,   1.22460635e-16])

In [4]: np.sin(math.pi/6*np.arange(12).reshape(2, 6))
Out[4]:
array([[  0.00000000e+00,   5.00000000e-01,   8.66025404e-01,
          1.00000000e+00,   8.66025404e-01,   5.00000000e-01],
       [  1.22460635e-16,  -5.00000000e-01,  -8.66025404e-01,
         -1.00000000e+00,  -8.66025404e-01,  -5.00000000e-01]])
```

Statt die Kreiszahl aus dem `math`-Modul zu nehmen, hätten wir sie genauso gut aus dem `numpy`-Paket nehmen können.

Funktionen wie die gerade benutzte Sinusfunktion aus dem `numpy`-Paket, die Arrays als Argumente akzeptieren, werden universelle Funktionen (*universal function* oder kurz *ufunc*) genannt. Die im `numpy`-Paket verfügbaren universellen Funktionen sind in der NumPy-Dokumentation zu ufuncs aufgeführt. Implementationen von speziellen Funktionen als universelle Funktion sind im `scipy`-Paket zu finden. Viele Funktionen in `scipy.special`,

jedoch nicht alle, sind als *ufuncs* implementiert. Als nur eines von vielen möglichen Beispielen wählen wir die Gammafunktion:

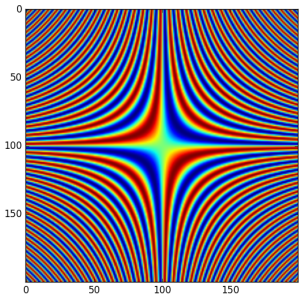
```
In [5]: import scipy.special

In [6]: scipy.special.gamma(np.linspace(1, 5, 9))
Out[6]:
array([  1.        ,   0.88622693,   1.        ,   1.32934039,
         2.        ,   3.32335097,   6.        ,  11.6317284 ,  24.        ])
```

Gelegentlich benötigt man eine Funktion von zwei Variablen auf einem Gitter. Man könnte hierzu die `meshgrid`-Funktion heranziehen, die wir im Abschnitt *Erzeugung von NumPy-Arrays* erwähnt hatten. Dabei muss allerdings entweder die Gitterpunkte explizit angeben oder beispielsweise mit `linspace` erzeugen. Dann ist es häufig einfacher, ein `mgrid`-Gitter zu verwenden.

```
In [7]: np.mgrid[0:3, 0:3]
Out[7]:
array([[[0, 0, 0],
        [1, 1, 1],
        [2, 2, 2]],

       [[0, 1, 2],
        [0, 1, 2],
        [0, 1, 2]]])

In [8]: np.mgrid[0:3:7j, 0:3:7j]
Out[8]:
array([[[ 0. ,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ],
        [ 0.5,  0.5,  0.5,  0.5,  0.5,  0.5,  0.5],
        [ 1. ,  1. ,  1. ,  1. ,  1. ,  1. ,  1. ],
        [ 1.5,  1.5,  1.5,  1.5,  1.5,  1.5,  1.5],
        [ 2. ,  2. ,  2. ,  2. ,  2. ,  2. ,  2. ],
        [ 2.5,  2.5,  2.5,  2.5,  2.5,  2.5,  2.5],
        [ 3. ,  3. ,  3. ,  3. ,  3. ,  3. ,  3. ]],

       [[ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ],
        [ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ],
        [ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ],
        [ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ],
        [ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ],
        [ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ],
        [ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ]]])
```

Man beachte, dass im zweiten Fall das dritte Element in der *slice*-Syntax imaginär ist. Damit wird angedeutet, dass nicht die Schrittweite gemeint ist, sondern die Anzahl der Werte im durch die ersten beiden Zahlen spezifizierten Intervall. Das folgende Beispiel zeigt eine weitere Anwendung. Man sieht hier, dass die Schrittweite in `mgrid` auch durch eine Gleitkommazahl gegeben sein kann.

```
In [9]: x, y = np.mgrid[-10:10:0.1, -10:10:0.1]

In [10]: plt.imshow(np.sin(x*y))
```

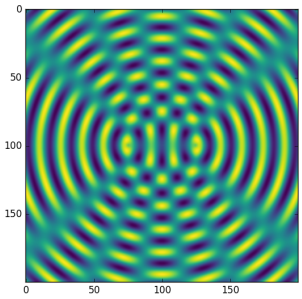


Unter Verwendung des *broadcasting* genügt auch ein mit `ogrid` erzeugtes Gitter, das wesentlich weniger Speicherplatz erfordert.

```
In [11]: np.ogrid[0:3:7j, 0:3:7j]
Out[11]:
[array([[ 0. ],
        [ 0.5],
        [ 1. ],
        [ 1.5],
        [ 2. ],
        [ 2.5],
        [ 3. ]]),
 array([[ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ]])]
```

Alternativ kann man die Werte für `x` und `y` auch expliziter wie folgt erzeugen.

```
In [12]: x = np.linspace(-40, 40, 200)

In [13]: y = x[:, np.newaxis]

In [14]: z = np.sin(np.hypot(x-10, y))+np.sin(np.hypot(x+10, y))

In [15]: plt.imshow(z, cmap='viridis')
```

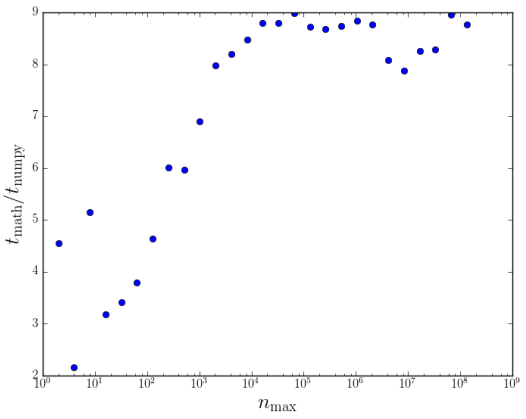


In Eingabe 13 ist es wichtig, dass eine weitere Achse hinzugefügt wird. Erst dann spannen `x` und `y` durch *broadcasting* ein zweidimensionales Gitter auf. In Eingabe 14 berechnet `hypot` die Länge der Hypotenuse eines rechtwinkligen Dreiecks mit den durch die Argumente gegebenen Kathetenlängen.

Es ist nicht nur praktisch, Funktionen von Arrays direkt berechnen zu können, sondern es spart häufig auch Rechenzeit. Wir wollen dies an einem Beispiel illustrieren, in dem wir den Sinus entweder einzeln in einer Schleife oder mit Hilfe einer universellen Funktion berechnen.

```
1 import math
2 import matplotlib.pyplot as plt
3 import numpy as np
4 import time
5
6 def sin_math(nmax):
```

```
    xvals = np.linspace(0, 2*np.pi, nmax)
    start = time.time()
    for x in xvals:
        y = math.sin(x)
    return time.time()-start

def sin_numpy(nmax):
    xvals = np.linspace(0, 2*np.pi, nmax)
    start = time.time()
    yvals = np.sin(xvals)
    return time.time()-start

maxpower = 27
nvals = np.empty(maxpower)
tvals = np.empty_like(nvals)
for nr, nmax in enumerate(np.logspace(1, maxpower, maxpower, base=2)):
    nvals[nr] = nmax
    tvals[nr] = sin_math(nmax)/sin_numpy(nmax)
plt.rc('text', usetex=True)
plt.xscale('log')
plt.xlabel('$n_\mathrm{max}$', fontsize=20)
plt.ylabel('$t_\mathrm{math}/t_\mathrm{numpy}$', fontsize=20)
plt.plot(nvals, tvals, 'o')
plt.show()
```



Ist die jeweilige Funktion häufig zu berechnen, so kann man etwa eine Größenordnung an Rechenzeit einsparen. Der Vorteil der universellen Funktion wird noch etwas größer, wenn man verlangt, dass das Ergebnis in einem Array oder in einer Liste abgespeichert wird. In der Funktion `sin_numpy` ist das bereits der Fall, nicht jedoch in der Funktion `sin_math`.

Wegen der genannten Rechenzeitvorteile lohnt es sich, einen Blick in die Liste der von NumPy zur Verfügung gestellten mathematischen Funktionen zu werfen.

3.6 Lineare Algebra

Physikalische Fragestellungen, die sich mit Hilfe von Vektoren und Matrizen formulieren lassen, benötigen zur Lösung sehr häufig Methoden der linearen Algebra. NumPy leistet hierbei Unterstützung, insbesondere mit dem `linalg`-Paket. Im Folgenden gehen wir auf einige Aspekte ein, ohne Vollständigkeit anzustreben. Daher empfiehlt es sich, auch einen Blick in den entsprechenden Abschnitt der Dokumentation zu werfen. Zunächst importieren wir die Module, die wir für die Beispiele dieses Kapitels benötigen:

```
In [1]: import numpy as np

In [2]: import numpy.linalg as LA
```

Beim Arbeiten mit Matrizen und NumPy muss man immer bedenken, dass der Multiplikationsoperator * nicht für eine Matrixmultiplikation steht. Vielmehr wird damit eine elementweise Multiplikation ausgeführt:

```
In [3]: a1 = np.array([[1, -3], [-2, 5]])

In [4]: a1
Out[4]:
array([[ 1, -3],
       [-2,  5]])

In [5]: a2 = np.array([[3, -6], [2, -1]])

In [6]: a2
Out[6]:
array([[ 3, -6],
       [ 2, -1]])

In [7]: a1*a2
Out[7]:
array([[ 3, 18],
       [-4, -5]])
```

Möchte man dagegen eine Matrixmultiplikation ausführen, so verwendet man das `dot`-Produkt:

```
In [8]: np.dot(a1, a2)
Out[8]:
array([[-3, -3],
       [ 4,  7]])
```

Ab Python 3.5 und NumPy 1.10 steht hierfür auch ein spezieller Operator zur Verfügung.

```
In [9]: a1 @ a2
Out[9]:
array([[-3, -3],
       [ 4,  7]])
```

Man könnte die Norm eines Vektors ebenfalls mit Hilfe des `dot`-Produkts bestimmen. Es bietet sich jedoch an, hierzu direkt die `norm`-Funktion zu verwenden:

```
In [10]: vec = np.array([1, -2, 3])

In [11]: LA.norm(vec)
Out[11]: 3.7416573867739413

In [12]: LA.norm(vec)**2
Out[12]: 14.0
```

Als nächstes wollen wir ein inhomogenes lineares Gleichungssystem `ax = b` lösen, wobei die Matrix `a` und der Vektor `b` gegeben sind und der Vektor `x` gesucht ist.

```
In [13]: a = np.array([[2, -1], [-3, 2]])

In [14]: b = np.array([1, 2])

In [15]: LA.det(a)
Out[15]: 0.99999999999999978

In [16]: np.dot(LA.inv(a), b)
Out[16]: array([ 4.,  7.])
```

In Eingabe 15 haben wir zunächst überprüft, dass die Determinante der Matrix `a` ungleich Null ist, so dass die invertierte Matrix existiert. Anschließend haben wir den Vektor `b` von links mit der Inversen von `a` multipliziert, um den Lösungsvektor zu erhalten. Allerdings erfolgt die numerische Lösung eines inhomogenen linearen

Gleichungssystems normalerweise nicht über eine Inversion der Matrix, sondern mit Hilfe einer geeignet durchgeführten Gauß-Elimination. NumPy stell hierzu die `solve`-Funktion zur Verfügung:

```
In [17]: LA.solve(a, b)
Out[17]: array([ 4.,  7.])
```

Eine nicht invertierbare Matrix führt hier wie auch bei der Bestimmung der Determinante auf eine `LinAlgError`-Ausnahme mit dem Hinweis auf eine singuläre Matrix.

Eine häufig vorkommende Problemstellung im Bereich der linearen Algebra sind Eigenwertprobleme. Die `eig`-Funktion bestimmt rechtsseitige Eigenvektoren und die zugehörigen Eigenwerte für beliebige quadratische Matrizen:

```
In [18]: a = np.array([[1, 3], [4, -1]])

In [19]: evals, evecs = LA.eig(a)

In [20]: evals
Out[20]: array([ 3.60555128, -3.60555128])

In [21]: evecs
Out[21]:
array([[ 0.75499722, -0.54580557],
       [ 0.65572799,  0.83791185]])

In [22]: for n in range(evecs.shape[0]):
    print(np.dot(a, evecs[:, n]), evals[n]*evecs[:, n])
Out[22]:
[ 2.72218119  2.36426089] [ 2.72218119  2.36426089]
[ 1.96792999 -3.02113415] [ 1.96792999 -3.02113415]
```

Die Ausgabe am Ende zeigt, dass die Eigenvektoren und -werte in der Tat korrekt sind. Zudem wird hier deutlich, dass die Eigenvektoren als Spaltenvektoren in der Matrix `evecs` gespeichert sind. Benötigt man nur die Eigenwerte einer Matrix, so kann man durch Benutzung der `eigvals`-Funktion Rechenzeit sparen.

Für die Lösung eines Eigenwertproblems von symmetrischen oder hermiteschen[3] Matrizen gibt es die Funktionen `eigh` und `eigvalsh`, bei denen es genügt, nur die obere oder die untere Hälfte der Matrix zu spezifizieren. Viel wichtiger ist jedoch, dass diese Funktionen einen erheblichen Zeitvorteil bieten können:

```
In [23]: a = np.random.random(250000).reshape(500, 500)

In [24]: a = a+a.T

In [25]: %timeit LA.eig(a)
1 loop, best of 3: 209 ms per loop

In [26]: %timeit LA.eigh(a)
10 loops, best of 3: 28.2 ms per loop
```

Hier wird in Eingabe 24 durch Addition der Transponierten eine symmetrische Matrix erzeugt, so dass die beiden Funktionen `eig` und `eigh` mit der gleichen Matrix arbeiten. Die Funktion `eigh` ist in diesem Beispiel etwa siebenmal so schnell.

3.7 Einfache Anwendungen

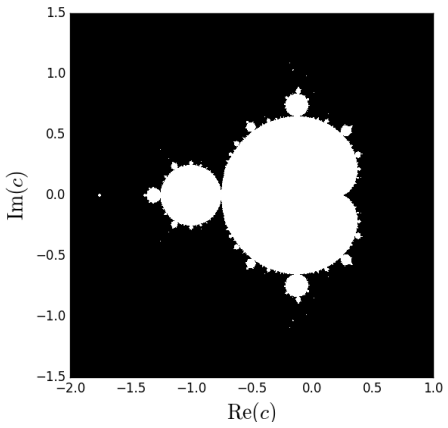
In diesem Abschnitt stellen wir einige einfache Problemstellungen vor, die sich mit Hilfe von NumPy gut bearbeiten lassen. Wir verzichten dabei bewusst auf die Angabe der Lösung, zeigen jedoch die zu erwartenden Resultate.

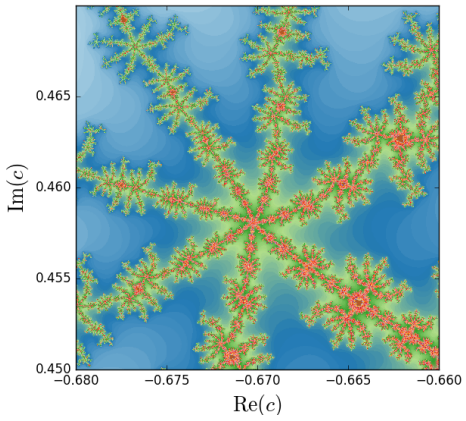
[3] Eine hermitesche Matrix geht beim Transponieren in die konjugiert komplexe Matrix über: $a_{ij} = a^*_{ji}$.

3.7.1 Mandelbrot-Menge

Betrachtet man die Rekursionsformel $z_{n+1} = z_n^2 + c$ mit $z_0 = 0$, so ist die Mandelbrot-Menge durch die komplexen Zahlen c definiert, für die Folge der z_n beschränkt bleibt. Überschreitet der Betrag von z die Schwelle 2, so divergiert die Folge. In der Praxis wird man natürlich nur eine endliche Zahl von Iterationen ausführen können.

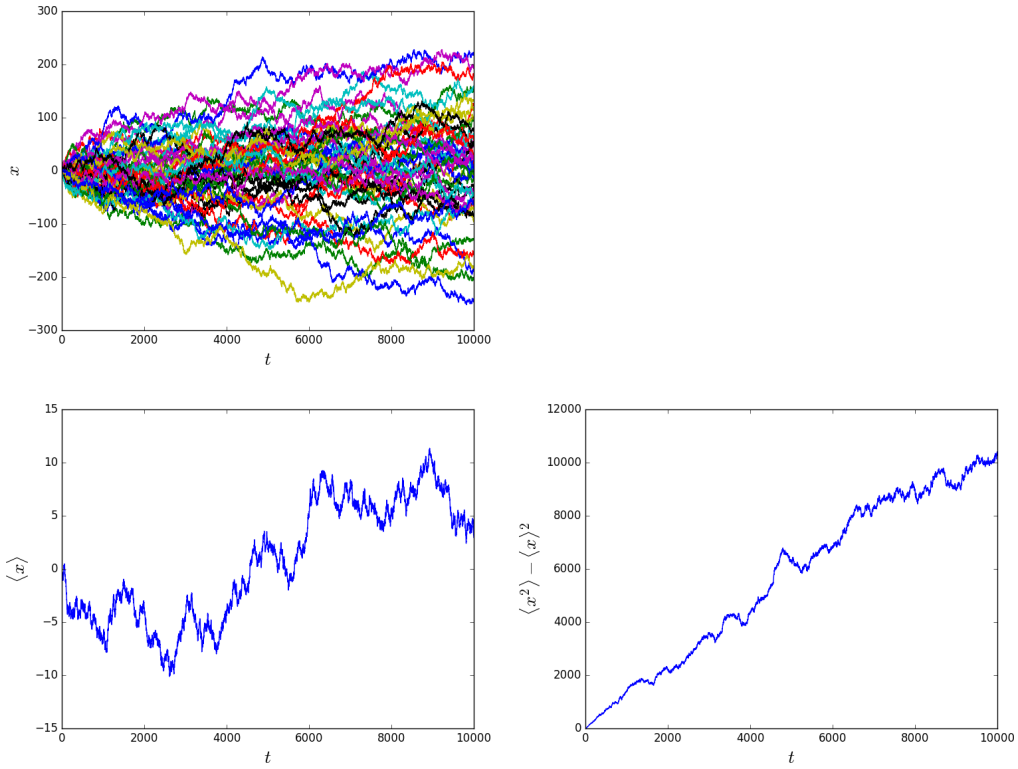
In der linken der beiden folgenden Abbildung ist die Mandelbrot-Menge in weiß dargestellt. Bei der Berechnung wird man möglicherweise Überlaufwarnungen erhalten. Diese kann man vermeiden, wenn man die Berechnung nur für die Werte von c fortsetzt, für die die Schwelle von Zwei noch nicht überschritten wurde. Dann eröffnet sich auch die Möglichkeit, die Zahl der Iterationen bis zum Erreichen der Schwelle abzuspeichern und in einer Farbabbildung darzustellen. Dies ist im rechten Bild für einen Ausschnitt gezeigt.





3.7.2 Brownsche Bewegung

Eine Zufallsbewegung in einer Dimension kann man durch Zufallszahlen darstellen, die aus der Menge der beiden Zahlen -1 und 1 gezogen werden. Jede Zufallszahl gibt die Richtung an, in der zu dem entsprechenden Zeitpunkt ein Schritt ausgeführt wird. Es sollen nun mehrere Realisierung erzeugt und graphisch dargestellt werden. Aus den erzeugten Daten soll auch der Mittelwert und die Varianz des Orts als Funktion der Zeit berechnet und dargestellt werden. Für die Berechnung ist es praktisch, dass NumPy eine Funktion zur Verfügung stellt, die für ein Array sukzessive kumulative Summen berechnen kann.



3.7.3 Fibonacci-Zahlen und goldener Schnitt

Die Fibonacci-Folge ist durch die Rekursionsformel $f_{n+1} = f_n + f_{n-1}$ mit den Anfangswerten $f_1 = f_2 = 1$ bestimmt. Das Verhältnis f_{n+1}/f_n geht für große n gegen einen Grenzwert, der mit Hilfe des `linalg`-Moduls von NumPy berechnet werden soll. Dies gelingt, wenn man die Rekursionsformel als Abbildung des Tupels (f_n, f_{n-1}) auf das Tupel (f_{n+1}, f_n) interpretiert.

3.7.4 Bildbearbeitung

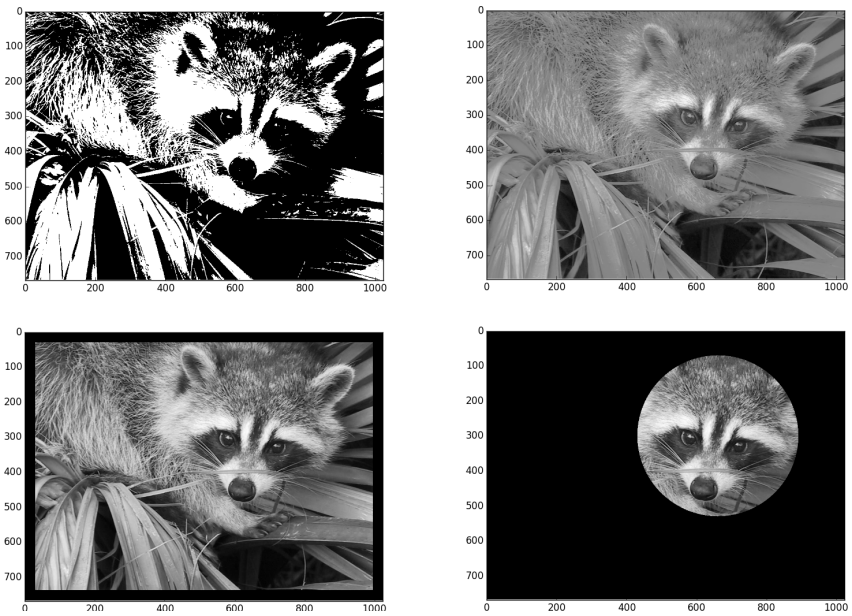
NumPy-Arrays können auch Bilddaten repräsentieren, so dass sich die Bearbeitung eines Bildes auf die Manipulation eines NumPy-Arrays zurückführen lässt. Der folgende Code zeigt, wie man ein in der SciPy-Bibliothek verfügbares Bild als Array interpretieren und graphisch darstellen kann.

```
In [1]: from scipy import misc

In [2]: face = misc.face(gray=True)

In [3]: face
Out[3]:
array([[114, 130, 145, ..., 119, 129, 137],
       [ 83, 104, 123, ..., 118, 134, 146],
       [ 68,  88, 109, ..., 119, 134, 145],
       ...,
       [ 98, 103, 116, ..., 144, 143, 143],
       [ 94, 104, 120, ..., 143, 142, 142],
       [ 94, 106, 119, ..., 142, 141, 140]], dtype=uint8)
```

Wir haben es hier mit einem Array zu tun, dessen Elemente durch vorzeichenlose 8-Bit-Integers dargestellt sind, also Zahlen zwischen 0 und 255 repräsentieren. Diese können wir als Grauwerte darstellen.

```
In [4]: plt.imshow(face, cmap=plt.cm.gray)
```

Durch geeignete Manipulation des NumPy-Arrays `face` lässt sich das Bild in ein Schwarz-Weiß-Bild verwandeln oder man kann die Kontrastkurve verändern. Des Weiteren kann man das Bild zum Beispiel mit verschiedenen Rahmen versehen. Einige Möglichkeiten zeigen die folgenden Bilder.



KAPITEL 4

Erstellung von Grafiken

Hat man mit Hilfe eines Pythonskripts Daten erzeugt, so möchte man diese häufig grafisch darstellen. Eine Möglichkeit hierfür besteht darin, die Daten in einer Datei abzuspeichern, um sie anschließend mit einem unabhängigen Programm grafisch aufzubereiten. Ein Programm, das im wissenschaftlichen Bereich gerne benutzt wird, ist beispielsweise `gnuplot`[1].

Ein anderer Zugang besteht darin, die Grafik direkt in dem Programm zu erzeugen, das die Daten berechnet. Hierfür gibt es eine Reihe von Programmpaketen. Am häufigsten benutzt wird wohl `matplotlib`[2], das eng mit den wissenschaftlichen Paketen `NumPy` und `SciPy` zusammenhängt. Ein weiteres Paket, das zudem sehr gut geeignet ist, um Schemazeichnungen und ähnliche Abbildungen zu erzeugen, ist `PyX`[3]. In diesem Kapitel werden wir eine Einführung in diese beiden Programmpakete geben. Beide sind jedoch so mächtig, dass es nicht möglich ist, alle Aspekte zu besprechen. Um eine Vorstellung von den jeweiligen Möglichkeiten zu bekommen, empfiehlt es sich, die entsprechenden Projektseiten im Internet zu besuchen und einen Blick auf die dort vorhandenen Beispielgalerien zu werfen.

Möchte man seine Daten mit einem Pythonprogramm grafisch aufbereiten, so sollte man sich Gedanken darüber machen, ob man dies innerhalb eines einzigen Programms tun möchte. Dies gilt insbesondere dann, wenn die Erzeugung der Daten zeitlich sehr aufwändig ist. Ist die Grafikerstellung nämlich fehlerhaft, so hat man nicht nur nicht die gewünschte Grafik, sondern zudem die erzeugten Daten verloren. In einem solchen Fall ist es sinnvoll, die Erzeugung von Daten und Grafiken zu trennen. Zumindest sollte man aber die Daten nach ihrer Erzeugung in einer Datei sichern.

4.1 Erstellung von Grafiken mit matplotlib

Bei matplotlib gibt es verschiedene Wege zur Erstellung einer Grafik. Man kann entweder das zu matplotlib gehörige `pylab`-Modul laden oder in IPython das magische Kommando `%pylab` verwenden. Dieser Weg führt dazu, dass umfangreiche Namensräume importiert werden, was einerseits der Bequemlichkeit dient, andererseits aber den Nachteil besitzt, dass häufig schwer nachzuvollziehen ist, woher eine bestimmte Funktion stammt. Daher ist von der Benutzung des `pylab`-Moduls eher abzuraten, und wir werden es im Folgenden auch nicht verwenden.

Eine zweite Variante besteht in der Benutzung des `pyplot`-Moduls aus matplotlib. Dieses Modul erlaubt es, den Zustand einer Grafik zu verändern und eignet sich daher besonders für die interaktive Entwicklung von Grafiken, zum Beispiel in einer IPython-Shell oder einem IPython-Notebook. Die Verwendung von pyplot ist aber genauso

[1] Für weitere Informationen siehe die Gnuplot-Webseite.
[2] Die Programmbibliothek zum Herunterladen und weitere Informationen findet man auf der matplotlib-Webseite.
[3] Die Programmbibliothek zum Herunterladen und weitere Informationen findet man auf der PyX-Webseite.

in einem normalen Python-Skript möglich. Die meisten der im Folgenden besprochenen Beispiele verwenden pyplot.

Schließlich bietet matplotlib einen objektorientierten Zugang, den wir in den letzten Beispiele dieses Kapitels etwas kennenlernen werden.

Um mit matplotlib arbeiten zu können, müssen zunächst die benötigten Module importiert werden:

```
In [1]: import numpy as np
   ...: import matplotlib as mpl
   ...: import matplotlib.pyplot as plt
   ...: %matplotlib
Using matplotlib backend: Qt5Agg
```

In den meisten Fällen wird hierzu das NumPy-Modul gehören, das wir wie gewohnt unter der Abkürzung `np` in der ersten Zeile importieren. Für die Arbeit mit pyplot wird der Importbefehl in der dritten Zeile benötigt, wobei wir uns hier an die Konvention halten, für pyplot die Abkürzung `plt` zu verwenden. Da wir gelegentlich das matplotlib-Modul selbst benötigen, um die Fähigkeiten von matplotlib zu demonstrieren, importieren wir es in der zweiten Zeile. Hierauf kann man in vielen Fällen verzichten.

Für die interaktive Arbeit in der IPython-Shell wird abschließend noch das magische Kommando `%matplotlib` verwendet. Arbeitet man mit einem Python-Skript, so entfällt dieser Schritt. Wenn man mit einem IPython-Notebook arbeitet, so hat man noch die Möglichkeit, Grafiken direkt in das Notebook einzubetten. In diesem Fall lautet das magische Kommando `%matplotlib inline`.

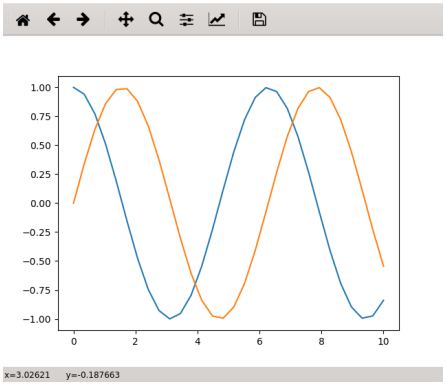
Wir wollen nun eine einfache Grafik erstellen, deren Eigenschaften wir im Weiteren schrittweise verändern werden. Zunächst erzeugen wir drei NumPy-Arrays, die die Daten für zwei Funktionen enthalten, die wir grafisch darstellen wollen.

```
In [2]: x = np.linspace(0, 10, 30)
   ...: y1 = np.cos(x)
   ...: y2 = np.sin(x)
```

Nun können wir leicht eine Grafik erzeugen, die die beiden Funktionen darstellt.

```
In [3]: plt.plot(x, y1)
   ...: plt.plot(x, y2)
Out[3]: [<matplotlib.lines.Line2D at 0x7f48b0a01320>]
```

Führt man diese beiden Kommandos aus, so öffnet sich ein Fenster, das neben der Grafik auch einige Icons enthält, die es erlauben, die Darstellung zu modifizieren.[4]



Die acht Icons haben von links nach rechts die folgenden Funktionen:

- Nachdem man mit Hilfe der anderen Icons Veränderungen an der Grafik vorgenommen hat, kann man mit diesem Icon wieder zur ursprünglichen Darstellung zurückkehren.
- Mit diesem Icon kommt man zur vorhergehenden Darstellung zurück.

[4] Die Abbildungen sind mit Matplotlib 2.0 unter Verwendung des Defaultstils erstellt worden.

- Ist man zu einer früheren Darstellung zurückgegangen, so kommt man mit diesem Icon zu neueren Darstellungen.
- Mit diesem Icon lässt sich der Bildausschnitt verschieben.
- Mit diesem Icon lässt sich ein kleinerer Bildausschnitt durch Aufziehen eines Rechtecks wählen. Man kann also bei Bedarf in die Abbildung hineinzoomen.
- Dieses Icon erlaubt es, die Seitenverhältnisse der Abbildung und die Größe der Ränder zu verändern.
- Dieses Icon ist nicht bei jedem Backend vorhanden, wird aber bei dem hier verwendeten Qt-Backend angezeigt. Es erlaubt die Wahl des x- und y-Achsenabschnitts, das Setzen von Beschriftungen sowie die Wahl von logarithmischen Achsen. Je nach Art der Grafik lässt sich hier zum Beispiel auch die Farbpalette einstellen. Diese Funktionalität ist aber auch durch entsprechende pyplot-Funktionen verfügbar, wie wir im Folgenden sehen werden.
- Das letzte Icon dient dazu, die Grafik in einer Datei zu speichern, sei es in einem Bitmap-Format wie zum Beispiel `png` oder in einem Vektorgrafik-Format wie `eps`, `pdf` oder `svg`.

Bevor wir uns aber mit der Beschriftung der Grafik beschäftigen, wollen wir uns zunächst der Darstellung der Daten zuwenden. Standardmäßig werden die Daten mit Hilfe einer durchgezogenen Kurve dargestellt, deren Farbe von einem Datensatz zum nächsten automatisch wechselt. Wir wollen nun die Darstellung dieser Kurven verändern. Dazu beschaffen wir uns zunächst die beiden Linien

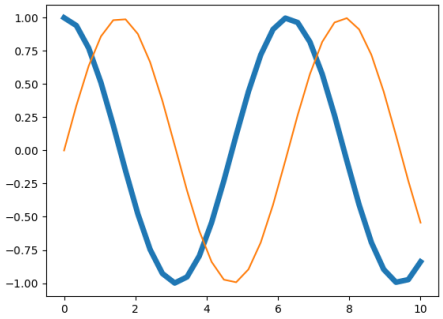
```
In [4]: line1, line2 = plt.gca().lines
```

`gca` steht hier für »get current axes«, das die aktuelle Untergrafik zurückgibt. In unserem Fall handelt es sich einfach um die aktuelle Grafik. Da aber eine Abbildung, wie wir später noch sehen werden, aus mehreren Untergrafiken bestehen kann, ist es im Prinzip wichtig, zwischen einer Untergrafik und der gesamten Abbildung zu unterscheiden. Mit Hilfe des `lines`-Attributes kann man eine Liste der aktuell vorhandenen Linien erhalten. Da wir wissen, dass es sich um zwei Linien handelt, können wir die Liste direkt entpacken.

Nachdem wir nun Zugriff auf die Linien haben, können wir deren Eigenschaften ändern. Dazu gibt es zwei Möglichkeiten, wie wir hier am Beispiel der Linienbreite zeigen wollen. Man kann die Linienbreite durch eine entsprechende Methode setzen, wobei man anschließend die Grafik mit einer `draw`-Anweisung in dem noch geöffneten Fenster neu erzeugen muss.

```
In [5]: line1.set_linewidth(5)
   ...: plt.draw()
```

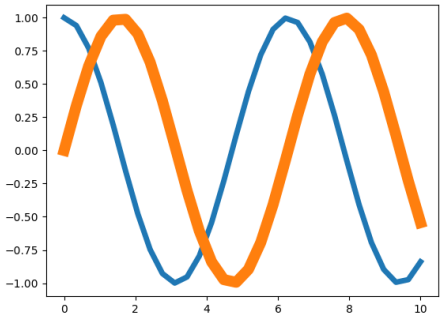
Das Ergebnis sieht dann folgendermaßen aus:



Alternativ kann man die `setp`-Methode verwenden, die es erlaubt, Eigenschaften zu setzen (»set properties«), in diesem Fall die Linienbreite der zweiten Linie.

```
In [6]: plt.setp(line2, linewidth=10)
```

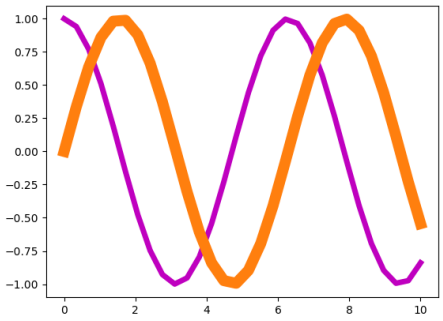
Die Grafik hat danach das folgende Aussehen:



Im nächsten Schritt wollen wir die Farben der beiden Linien verändern und verwenden dafür wieder die `setp`-Methode,

```
In [7]: plt.setp(line1, color='m')
```

und erhalten folgende Ausgabe:



In diesem Beispiel haben wir ausgenutzt, dass sich eine Reihe von Farben mit Hilfe eines Buchstabens auswählen lassen, nämlich

```
In [8]: mpl.colors.BASE_COLORS
Out[8]: {'b': (0, 0, 1),
         'c': (0, 0.75, 0.75),
         'g': (0, 0.5, 0),
         'k': (0, 0, 0),
         'm': (0.75, 0, 0.75),
         'r': (1, 0, 0),
         'w': (1, 1, 1),
         'y': (0.75, 0.75, 0)}
```

Es stehen auf diese Weise also die Farben Blau (`b`), Cyan (`c`), Grün (`g`), Schwarz (`k`), Magenta (`m`), Rot (`r`), Weiß (`w`) und Gelb (`y`) zur Verfügung. Die aufgelisteten Tupel geben die RGB-Darstellung der jeweiligen Farben an, das heißt den Anteil der Farben Rot, Grün und Blau. Dieses ist direkt an den Farben Rot und Blau nachvollziehbar. Bei Grün ist die Helligkeit reduziert, da diese Farbe bei voller Helligkeit im Allgemeinen schlecht zu sehen ist. Bei Weiß haben alle drei Kanäle ihren Maximalwert, während bei Schwarz keiner der Kanäle beiträgt. Bei den Komplementärfarben Cyan, Magenta und Gelb sind zwei der drei Kanäle beteiligt.

Eine große Zahl von Farben lässt sich durch Angabe eines Farbnamens erhalten, wobei man sich eine vollständige Liste der verfügbaren Namen anzeigen lassen kann.

```
In [9]: mpl.colors.cnames
Out[9]: {'aliceblue': '#F0F8FF', 'antiquewhite': '#FAEBD7', 'aqua': '#00FFFF',
         'aquamarine': '#7FFFD4', 'azure': '#F0FFFF', 'beige': '#F5F5DC',
         ...
         'violet': '#EE82EE', 'wheat': '#F5DEB3', 'white': '#FFFFFF',
         'whitesmoke': '#F5F5F5', 'yellow': '#FFFF00', 'yellowgreen': '#9ACD32'}
```

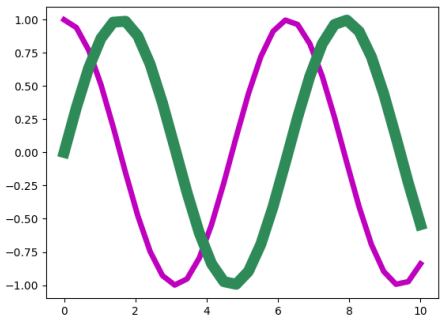
Jeder Farbe ist wieder ein RGB-Wert zugeordnet, der in diesem Fall hexadezimal kodiert ist. Der RGB-Wert besteht hier aus drei zweistelligen Hexadezimalzahlen, die demnach Werte zwischen 0 und 255 repräsentieren. Um auf die obige RGB-Darstellung zu kommen, muss man das Intervall von 0 bis 255 auf das Intervall von 0 bis 1 abbilden, also durch 255 teilen.

Die obige Liste der verfügbaren Farbnamen ist nicht vollständig wiedergegeben. Für die Farbauswahl praktischer ist die folgende, daraus erstellte Farbtabelle, die mit Hilfe des Skripts `named_colors.py` aus der matplotlib-Galerie erzeugt werden kann.

Als Beispiel ändern wir die Farbe der zweiten Linie auf `seagreen`

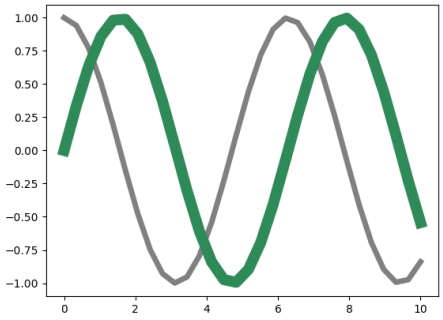
```
In [10]: plt.setp(line2, color='seagreen')
```

und erhalten damit die folgende Ausgabe



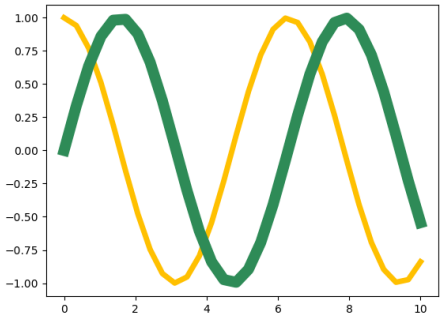
Falls die bisher genannten Farben nicht ausreichen sollten, kann man Farben auch stufenlos wählen, wenn man von der endlichen Auflösung der Gleitkommazahlen in Python absieht. So lassen sich Grautöne kontinuierlich von Schwarz bis Weiß mit Hilfe einer Zahl zwischen 0 und 1 festlegen. In dem folgenden Beispiel wird für die erste Linie ein Grauton vorgegeben.

```
In [11]: plt.setp(line1, color='0.5')
```

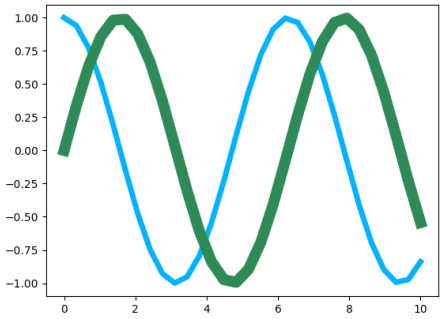


Wie bereits weiter oben beschrieben, lassen sich Farben in der RGB-Darstellung hexadezimal oder durch ein aus drei Zahlen bestehendes Tupel auswählen, wie die folgenden beiden Beispiele zeigen.

```
In [12]: plt.setp(line1, color='#FFC000')
```



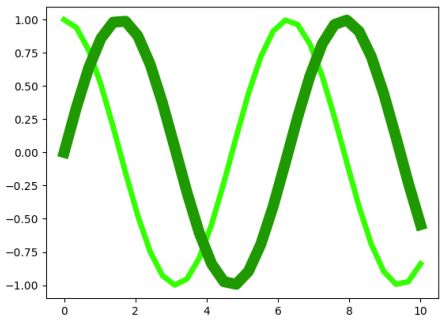

```
In [13]: plt.setp(line1, color=(0, 0.7, 1))
```



Alternativ zum RGB-System lassen sich Farben auch mit Hilfe des HSV-Systems darstellen, in dem die Farben ebenfalls durch ein Tupel aus drei Gleitkommazahlen zwischen 0 und 1 charakterisiert werden. Der erste Wert steht für den Farbton (»hue«) und gibt die Position im Farbkreis an. Dabei sind die Werte 0 und 1 der Farbe Rot zugeordnet. Grün liegt bei 1/3 und Blau bei 2/3. Der zweite Wert gibt die Farbsättigung (»saturation«) an. Verringert man diesen Wert, so geht man von der Farbe zu einem entsprechenden Grauton über. Der dritte Wert (»value«) schließlich beeinflusst die Helligkeit der Farbe.

Im folgenden Beispiel setzen wir den ersten Wert auf 0.3 und wählen so einen Grünton aus. Wie schon weiter oben erwähnt, besteht bei Grüntönen die Gefahr, dass sie zu hell erscheinen. Daher haben wir für die zweite Linie den dritten Wert des Tupels reduziert.

```
In [14]: plt.setp(line1, color=mpl.colors.hsv_to_rgb((0.3, 1, 1)))
   ...: plt.setp(line2, color=mpl.colors.hsv_to_rgb((0.3, 1, 0.6)))
```



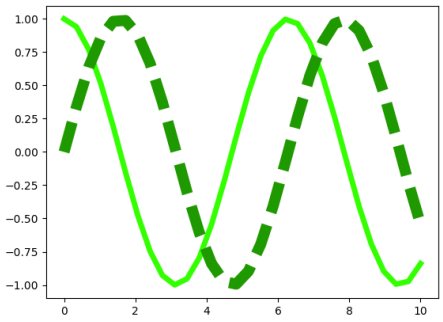
Um Unterschiede zwischen verschiedenen Linien hervorzuheben, kann man nicht nur verschiedene Farben einsetzen, sondern auch unterschiedliche Linienarten. Dies ist besonders dann sinnvoll, wenn man davon ausgehen muss, dass die Abbildung in schwarz/weiß gedruckt wird, so dass Farben auf Grautöne abgebildet würden.

Die Linienart wird in matplotlib durch einen String charakterisiert, der sich aus der folgenden Auflistung ergibt.

```
In [15]: mpl.lines.Line2D.lineStyles
Out[15]: {'': '_draw_nothing',
          ' ': '_draw_nothing',
          '-': '_draw_solid',
          '--': '_draw_dashed',
          '-.': '_draw_dash_dot',
          ':': '_draw_dotted',
          'None': '_draw_nothing'}
```

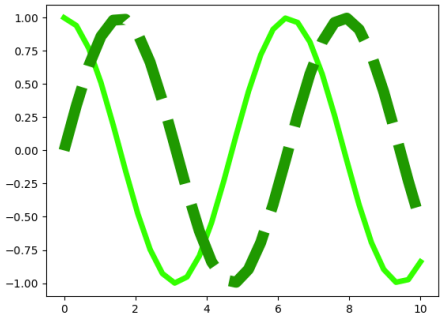
Möchte man eine Linie gestrichelt darstellen, kann man wie bei den Farben die `setp`-Methode verwenden und übergibt mit Hilfe des Schlüsselworts `linestyle` den entsprechenden Wert `'--'`.

```
In [16]: plt.setp(line2, linestyle='--')
```

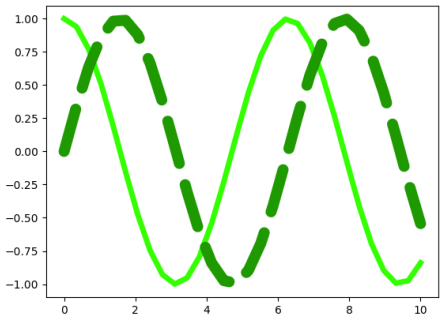


Bei Bedarf kann man die Strichlänge anpassen oder auch die Form der Strichenden festlegen.

```
In [17]: plt.setp(line2, dashes=(6, 2))
```

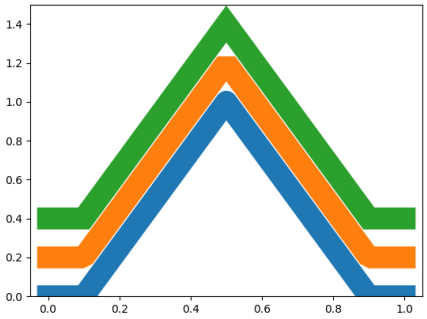



```
In [18]: plt.setp(line2, dashes=(4, 2), dash_capstyle='round')
```



Vor allem bei Linienverläufen, die Spitzen enthalten, kann es interessant sein, die Form vorzugeben, mit der Linien aneinander gefügt werden. Es gibt hierfür drei Möglichkeiten, die im Folgenden dargestellt sind.

```
In [19]: plt.cla()
   ...: xdata = np.array([0, 0.1, 0.5, 0.9, 1])
   ...: ydata = np.array([0, 0, 1, 0, 0])
   ...: for n in range(3):
   ...:     plt.plot(xdata, ydata+0.2*n)
   ...: line = plt.gca().lines
   ...: plt.setp(line, linewidth=20)
   ...: plt.ylim(0, 1.5)
   ...: for n, joinstyle in enumerate(('round', 'bevel', 'miter')):
   ...:     plt.setp(line[n], solid_joinstyle=joinstyle)
```

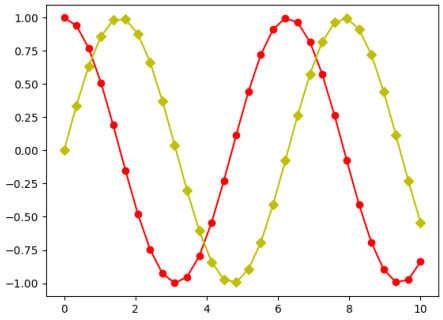


In dem obigen Codebeispiel haben wir eine Funktion verwendet, die bis jetzt noch nicht besprochen wurde, nämlich `cla()`. Der Name steht für »clear current axes« und löscht die aktuellen Unterabbildung, die in unserem Fall einfach der aktuellen Abbildung entspricht.

Bis jetzt haben wir die Datenpunkte lediglich durch Linien dargestellt. Dies ist jedoch häufig entweder nicht gewünscht oder nicht ausreichend. Insbesondere bei Messdaten sollen die Datenpunkte oft einzeln markiert werden.

In vielen Fällen kann man bei matplotlib auf eine sehr einfache Syntax zurückgreifen. Um dies zu illustrieren, erstellen wir unsere Beispielgrafik, mit der wir bisher überwiegend gearbeitet haben, neu. Im ersten `plot`-Aufruf verwenden wir als drittes Argument die Zeichenkette `'ro-'`. Das erste Zeichen, `r`, wird als Farbe interpretiert und bedeutet, wie wir bereits wissen, Rot. Das zweiten Zeichen, `o`, wird als Symbol interpretiert, in diesem Fall als Kreis. Der abschließende Bindestrich gibt an, dass nicht nur Symbole dargestellt werden sollen, sondern die Symbole außerdem durch eine durchgezogene Linie verbunden werden sollen. Für den zweiten Datensatz verlangen wir mit `'yD-'` statt roten Kreisen gelbe Rauten (»diamonds«).

```
In [20]: plt.cla()
   ...: plt.plot(x, y1, 'ro-')
   ...: plt.plot(x, y2, 'yD-')
Out[20]: [<matplotlib.lines.Line2D at 0x7f48b0110860>]
```



Um die Symbole modifizieren zu können, verschaffen wir uns Zugriff auf die einzelnen Linie, wie wir dies früher schon getan haben.

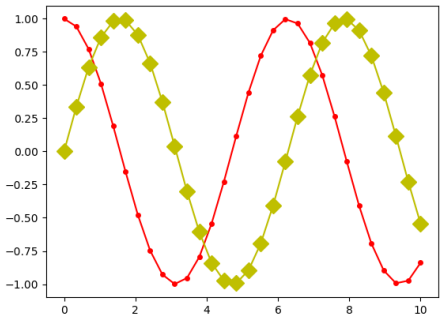
```
In [21]: line1, line2 = plt.gca().lines
```

Zunächst verändern wir die Größe der Symbole. Dazu beschaffen wir uns zunächst den aktuellen Wert, um eines der Symbole relativ zu diesem Wert vergrößern und das andere verkleinern zu können. Hierzu verwenden wir die Partnermethode zu `setp`. nämlich `getp`, das für »get property« steht, es also erlaubt, Eigenschaften in Erfahrung zu bringen.

```
In [22]: plt.getp(line1, 'markersize')
Out[22]: 6.0
```

Die Standardgröße der Symbole beträgt also 6, so dass wir davon ausgehend nun mit der bewährten `setp`-Methode neue Sybolgrößen setzen können.

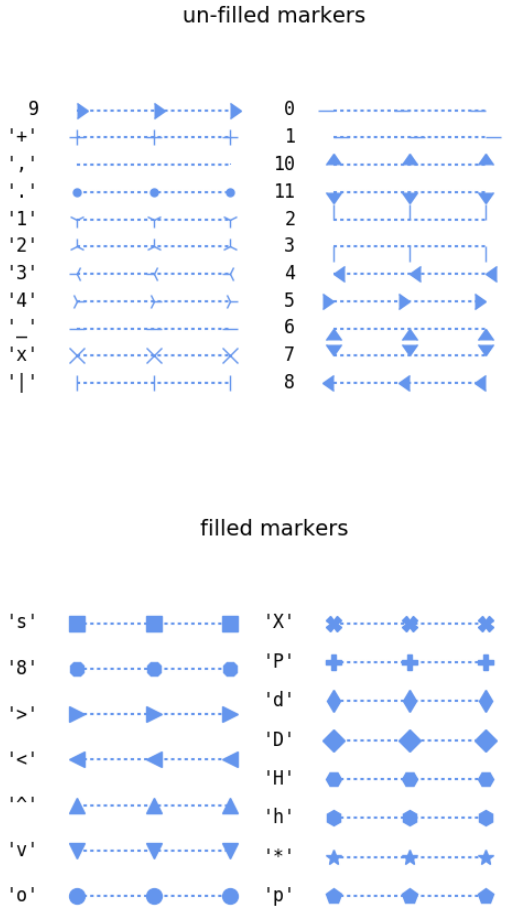
```
In [23]: plt.setp(line1, markersize=4)
   ...: plt.setp(line2, markersize=10)
```



Neben Kreisen und Rauten stellt matplotlib natürlich noch weitere Symbole zur Verfügung, die man sich als Dictionary ausgeben lassen kann.

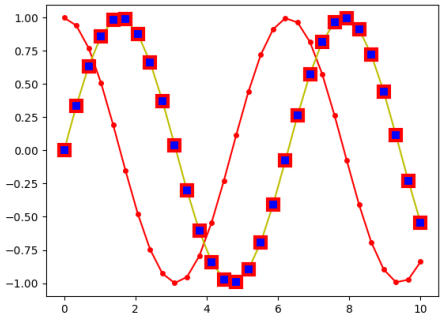
```
In [24]: mpl.lines.Line2D.markers
Out[24]: {'.': 'point', ',': 'pixel', 'o': 'circle', 'v': 'triangle_down',
          ...
          8: 'caretleftbase', 9: 'caretrightbase', 10: 'caretupbase',
          11: 'caretdownbase', 'None': 'nothing', None: 'nothing',
          ' ': 'nothing', '': 'nothing'}
```

Das Dictionary ist hier nur auszugsweise wiedergegeben, da die folgende grafische Darstellung nützlicher ist. Zu beachten ist, dass zwar die meisten Symbole mit einem Zeichen ausgewählt werden, es aber dennoch einige Symbole gibt, auf die mit einer Ziffer Bezug genommen wird. So bezeichnen `4` und `'4'` unterschiedliche Symbole.



Bei Bedarf lassen sich die Eigenschaften von Symbolen im Detail festlegen. Das folgende Beispiel erzeugt auf der Linie 2 quadratische blaue Symbole mit einem roten Rand von 3 Punkten Breite.

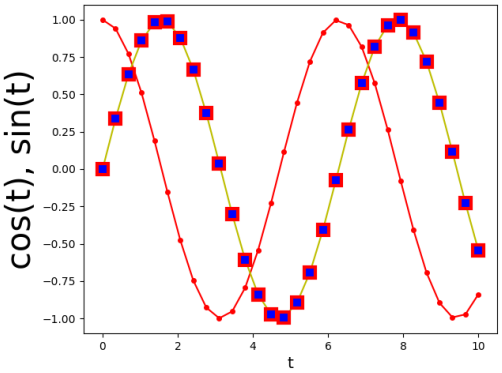
```
In [25]: plt.setp(line2, marker='s', markerfacecolor='b',
   ...:           markeredgewidth=3, markeredgecolor='r')
```



Zu einem ordentlichen Funktionsgraphen gehört selbstverständlich auch eine Achsenbeschriftung. Im einfachsten Fall verwendet man `xlabel` und `ylabel` und gibt den Text als Argument in Form einer Zeichenkette an. Häufig muss man allerdings die Schriftgröße anpassen. Dies kann mit Hilfe eines entsprechenden Bezeichners geschehen, wie es hier für die *x*-Achse erfolgt, oder aber durch Angabe einer Schriftgröße in Punkten, wie es hier für die *y*-Achse gezeigt ist.

```
In [26]: plt.xlabel('t', fontsize='x-large')
Out[26]: <matplotlib.text.Text at 0x7f48b0a30668>
```

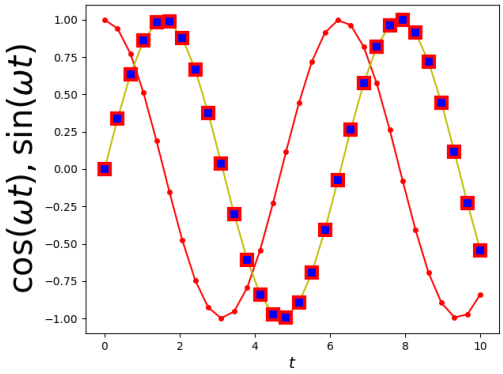
```
In [27]: plt.ylabel('cos(t), sin(t)', fontsize=30)
Out[27]: <matplotlib.text.Text at 0x7f48b015a780>
```



Einen besseren Mathematiksatz erhält man, wenn man die TeX-Syntax verwendet[5]. Dabei werden mathematische Teile in Dollarzeichen eingeschlossen, was unter anderem zur Konsequenz hat, dass mathematische Symbole kursiv dargestellt werden. TeX- und LaTeX-Kommandos beginnen mit `\` und so lassen sich beispielsweise griechische Buchstaben durch Voranstellen dieses Zeichens vor den Namen des Buchstabens erzeugen. `\omega` wird so zu einem ω.

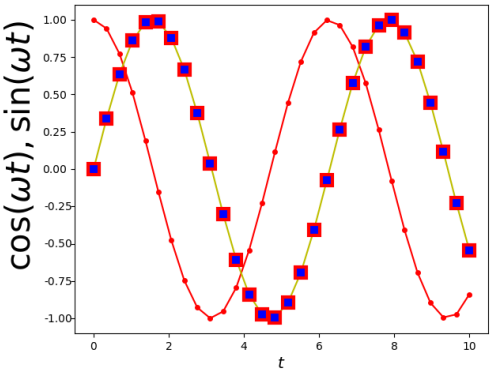
```
In [28]: plt.xlabel('$t$')
   ...: plt.ylabel(r'$\cos(\omega t), \sin(\omega t)$')
Out[28]: <matplotlib.text.Text at 0x7f48b015a780>
```

[5] Bei TeX und seiner Variante LaTeX handelt es sich um ein sehr mächtiges Textsatzsystem, das im wissenschaftlichen Umfeld stark genutzt wird. Für weitere Informationen siehe zum Beispiel www.dante.de.



Standardmäßig interpretiert matplotlib selbst die in TeX-Syntax übergebenen Zeichenketten. Dabei wird nur ein Teil der äußerst umfangreichen Möglichkeiten von TeX unterstützt. Man kann jedoch auch verlangen, dass der Text von der lokal vorhandenen TeX-Installation gesetzt wird.

```
In [29]: mpl.rc('text', usetex = True)
   ...: plt.ylabel(r'$\cos(\omega t), \sin(\omega t)$')
Out[29]: <matplotlib.text.Text at 0x7f48b015a780>
```

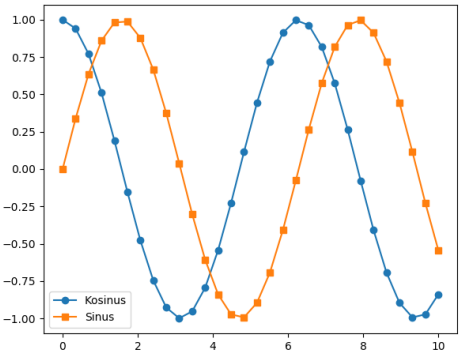


Gelegentlich ist es sinnvoll, eine Grafik mit einer Legende zu versehen, die die Bedeutung der einzelnen Kurven erläutert. Eine Möglichkeit, den Beschriftungstext vorzugeben, besteht darin, dies gleich bei der Erzeugung der Kurven mit Hilfe der Variable `label` zu tun.

```
In [30]: plt.cla()
   ...: plt.plot(x, y1, 'o-', label='Kosinus')
   ...: plt.plot(x, y2, 's-', label='Sinus')
Out[30]: [<matplotlib.lines.Line2D at 0x7f48b09f9828>]
```

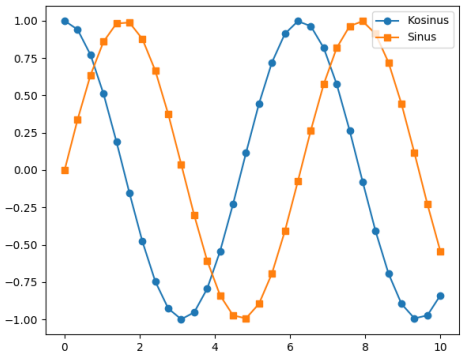
Die `legend`-Methode wird anschließend benutzt, um die Legende im Graphen zu setzen.

```
In [31]: plt.legend()
Out[31]: <matplotlib.legend.Legend at 0x7f48af928080>
```



Man kann aber beispielsweise auch verlangen, dass die Legende rechts oben platziert wird.

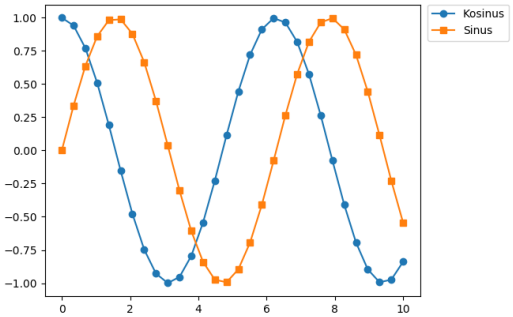
```
In [32]: plt.legend(loc='upper right')
Out[32]: <matplotlib.legend.Legend at 0x7f48af946ba8>
```



Steht ausreichend horizontaler Platz zur Verfügung, so kann es im vorliegenden Fall günstig sein, die Legende außerhalb der Grafik anzuordnen, wie das folgende Beispiel zeigt.

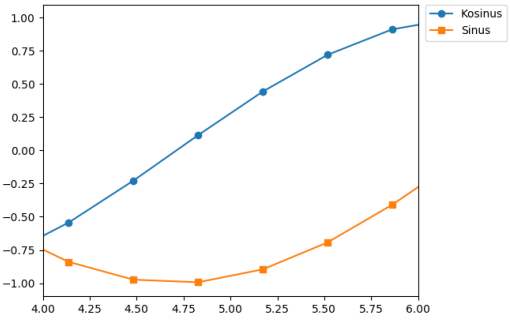
```
In [33]: plt.legend(bbox_to_anchor=(1.02, 1), loc='upper left', borderaxespad=0)
Out[33]: <matplotlib.legend.Legend at 0x7f48af8d4780>
```

Hier wird festgelegt, dass die Legende mit der oberen linken Ecke etwas außerhalb des rechten oberen Randes der Grafik platziert wird. Der genaue Punkt wird relativ zur so genannten »bounding box« angegeben, die hier immer die horizontale und vertikale Länge 1 besitzt, also unabhängig von den Problemkoordinaten ist, die hier in horizontaler Richtung von 0 bis 10 und in vertikaler Richtung von -1 bis 1 laufen. Der Punkt `(1.02, 1)` liegt somit wie behauptet leicht rechts von der oberen rechten Ecke der Grafik.



Die Achsen haben neben der bereits besprochenen Achsenbeschriftung noch weitere Eigenschaften, die sich in matplotlib einstellen lassen. So kann es sinnvoll sein, den Umfang der Achsen in Problemkoordinaten unabhängig von dem Bereich festzulegen, in dem sich die Daten befinden. Im folgenden Beispiel wird die x-Achse auf einen kleinen Ausschnitt der bisherigen Achse eingeschränkt.

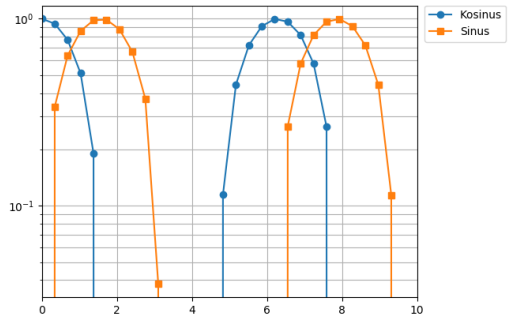
```
In [34]: plt.xlim(4, 6)
Out[34]: (4, 6)
```




```
In [35]: plt.xlim(0, 10)
Out[35]: (0, 10)
```

Nachdem wir wieder zum ursprünglichen Achsenumfang zurückgekehrt sind, wollen wir die y-Achse logarithmisch darstellen. Dies hat in unserem Fall den Nebeneffekt, dass die Bereiche, in denen negative y-Werte auftreten, nicht dargestellt werden. Außerdem wollen wir ein Koordinatengitter für beide Achsen anzeigen lassen.

```
In [36]: plt.yscale('log')
   ...: plt.grid(which='both')
```



Der Wechsel zwischen linearer und logarithmischer Achse kann auch direkt im Grafikfenster mit Hilfe der Tasten `k` für die x-Achse und `l` für die y-Achse erfolgen. Dies ist besonders dann praktisch, wenn man nur schnell überprüfen möchte, wie Daten in einer einfach- oder doppelt-logarithmischen Auftragung aussehen.

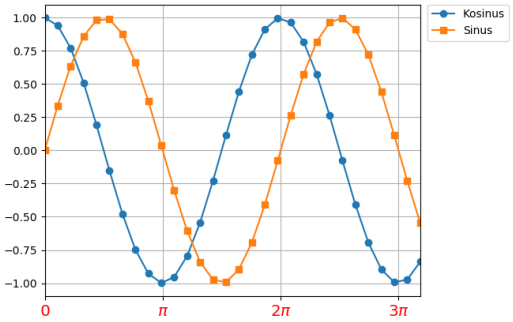
```
In [37]: plt.yscale('linear')
```

Wieder zu einer linearen Skala zurückgekehrt, wollen wir noch an einem einfachen Beispiel zeigen, wie man die Achseneinteilung den jeweiligen Bedürfnissen anpassen kann. Eine genauere Diskussion der verschiedenen Möglichkeiten, die matplotlib zu diesem Zweck zur Verfügung stellt, würde hier zu weit führen.

Nachdem wir in unserer Beispielgrafik trigonometrische Funktionen darstellen, wollen wir gerne die x-Achse in Vielfache von π einteilen. Dies kann dadurch geschehen, dass wir der `xticks`-Methode eine Liste von Punkten auf der x-Achse sowie ein Tupel mit den entsprechenden Beschriftungen übergeben. Außerdem können wir Schrifteigenschaften festlegen, zum Beispiel die Schriftgröße und die Schriftfarbe, die wir hier rot wählen.

```
In [38]: plt.xticks(np.pi*np.arange(0, 4), ('0', r'$\pi$', r'$2\pi$', r'$3\pi$'),
   ...:            size='x-large', color='r')
Out[38]: ([<matplotlib.axis.XTick at 0x7f48b0a011d0>,
  <matplotlib.axis.XTick at 0x7f48b0178908>,
  <matplotlib.axis.XTick at 0x7f48b016ada0>,
  <matplotlib.axis.XTick at 0x7f48af92f8d0>],
 <a list of 4 Text xticklabel objects>)
```

Das Ergebnis sieht dann folgendermaßen aus:



Hat man die optimale Form für die Grafik erreicht, so möchte man diese häufig auch abspeichern. Wie wir schon gesehen haben, lässt sich das mit dem entsprechenden Icon im Grafikfenster erreichen. Genauso gut kann man die Grafik aber auch mit Hilfe der `savefig`-Funktion in einer Datei speichern. Dabei ist zunächst das gewünschte Format festzulegen.

Für die Darstellung auf einem Bildschirm oder zum Beispiel für die Einbindung in eine Webseite eignet sich ein Bitmapformat, das die Abbildung in einer gerasterten Form abspeichert. Hierfür gibt es sehr viele verschiedene Formate. In matplotlib bietet sich die Verwendung des `png`-Formats[6] an.

```
In [39]: plt.savefig('example.png')
```

Der Nachteil von Bitmapformaten ist, dass die Pixelstruktur der Abbildung bei einer vergrößerten Darstellung mehr oder weniger stark sichtbar wird. Benötigt man eine höher aufgelöste Ausgabe, beispielsweise zum Druck, so wird man eher zu einem Vektorformat greifen. matplotlib bietet hier das Postscript-Format an, das sich im Encapsulated Postscript-Format für die Einbettung in andere Dokumente eignet. Ein heute weit verbreitetes Format ist PDF (»portable document format«). Dieses Format kann auch Bitmap-Anteile enthalten, die dann natürlich der beschriebenen Skalierungsproblematik unterliegen. SVG (»scalable vector graphics«) ist ein Vektorgrafikformat, das für die Verwendung von Vektorgrafiken im Internet entwickelt wurde und von modernen Webbrowsern zumindest zu großen Teilen dargestellt werden kann.

```
In [40]: plt.savefig('example.pdf')
```

Die `savefig`-Funktion benötigt als zwingendes Argument den Namen der Datei, in der die Abbildung gespeichert werden soll. Sie akzeptiert außerdem eine Reihe weiterer Argumente, mit denen man zum Beispiel die Auflösung einer Bitmapgrafik oder die Hintergrundfarbe der Abbildung beeinflussen kann. Bei Bedarf empfiehlt sich ein Blick in die matplotlib-Dokumentation.

Abschließend wollen wir aus dem breiten Spektrum der Möglichkeiten von matplotlib noch drei Problemstellungen ansprechen, die in einem wissenschaftlichen Umfeld häufig vorkommen. Als erstes wollen wir uns mit der Darstellung von zweidimensionalen Daten mit Hilfe eines Konturplots beschäftigen.

Zunächst löschen wir die noch vorhandene Abbildung mit Hilfe der `clf`-Funktion (»clear figure«) und fangen auf diese Weise neu an.

```
In [41]: plt.clf()
```

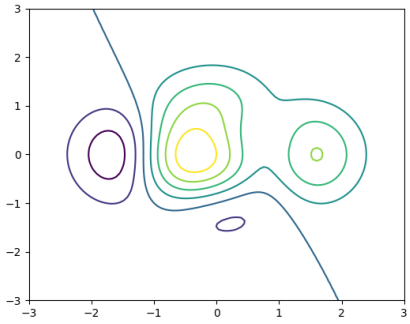
Um Daten zur Verfügung zu haben, die wir darstellen können, verwenden wir NumPy, mit dem wir zunächst ein regelmäßiges Koordinatengitter zu erzeugen, über dem dann eine Funktion von zwei Variablen ausgewertet wird.

```
In [42]: x, y = np.mgrid[-3:3:100j, -3:3:100j]
   ...: z = (1-x+x**5+y**3)*np.exp(-x**2-y**2)
```

Um eine Vorstellung vom Verhalten dieser Funktion anhand von Konturlinien zu erhalten, genügt es im einfachsten Fall, die x- und y-Koordinaten des Gitters und die zugehörigen Funktionswerte an die `contour`-Funktion zu übergeben.

[6] `png` steht für »portable network graphics« und speichert die Abbildung erlustfrei, aber durch Komprimierung platzsparend.

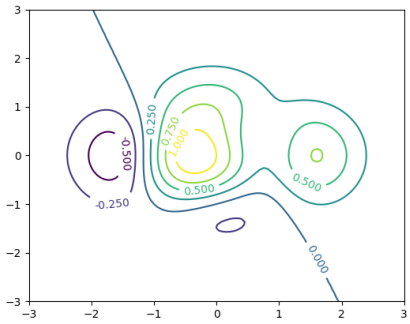
```
In [43]: contourset = plt.contour(x, y, z)
```



Wenn man jedoch nicht weiß, welche Farbe welchem Wert der Funktion zugeordnet ist, ist so ein Bild häufig nur eingeschränkt nützlich. Es ist daher sinnvoll, jede Konturlinie mit dem zugehörigen Funktionswert zu beschriften. Hierzu haben wir in der vorhergehenden Anweisung bereits die Konturlinien in der Variable `contourset` gespeichert, mit deren Hilfe wir nun die Beschriftung vornehmen können.

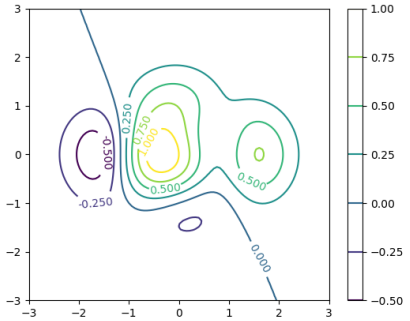
```
In [44]: plt.clabel(contourset, inline=1)
Out[44]: <a list of 8 text.Text objects>
```

Ist das Argument `inline` gleich `True` oder gleich 1, so wird die Kontur unter der Beschriftung entfernt. Dies ist die Standardeinstellung und müsste daher nicht unbedingt explizit angegeben werden.



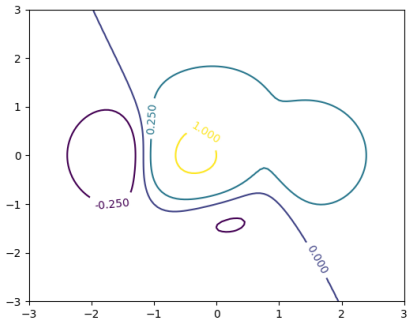
Eine andere Möglichkeit, eine Verbindung zwischen der Farbe der Konturlinien und dem entsprechenden Funktionswert herzustellen, besteht in der Verwendung eines Farbstreifens neben der eigentlichen Abbildung. Im Falle von Konturlinien sind hier nur einzelne Linien bei den entsprechenden Werten zu sehen. Füllt man die Flächen zwischen den Konturlinien, so ist dieser Farbstreifen durchgehend farbig dargestellt, wie wir gleich noch sehen werden.

```
In [45]: plt.colorbar(contourset)
Out[45]: <matplotlib.colorbar.Colorbar at 0x7f48aeda8a20>
```

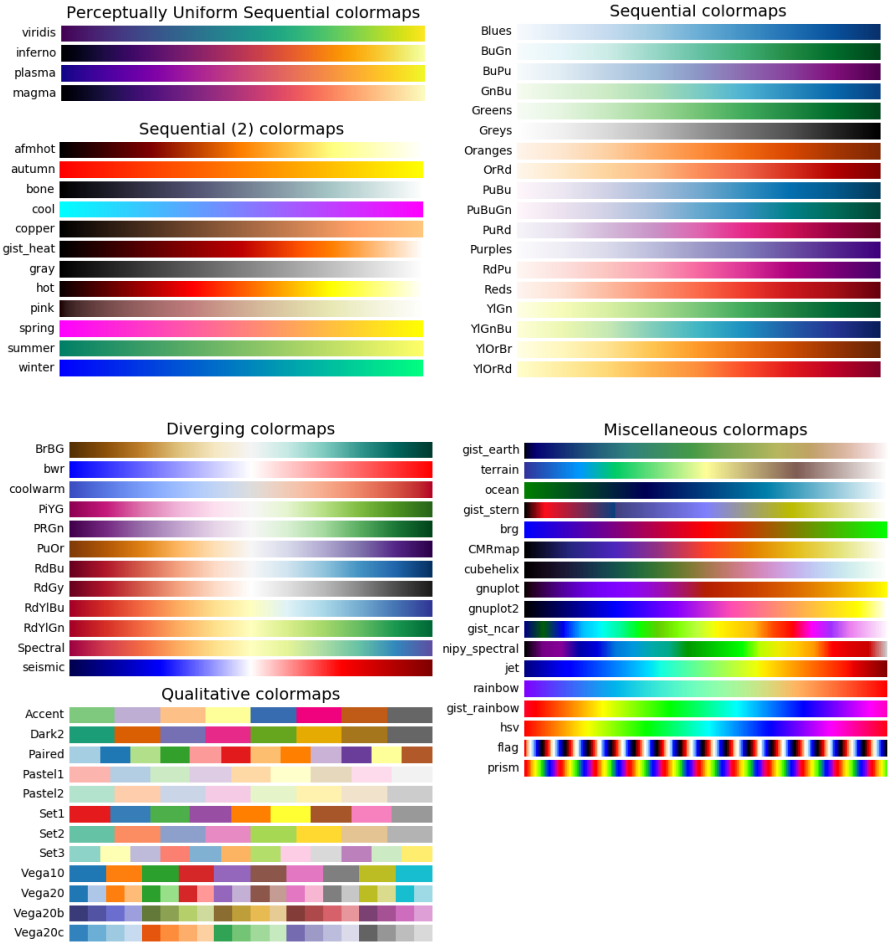


Bis jetzt hatten wir es matplotlib überlassen, die Werte für die einzelnen Konturlinien zu bestimmen. Es ist aber auch möglich, die entsprechenden Werte in einer Liste vorzugeben, wie das nächste Beispiel zeigt.

```
In [46]: plt.clf()
   ...: contourset = plt.contour(x, y, z, [-0.25, 0, 0.25, 1])
   ...: plt.clabel(contourset, inline=1)
Out[46]: <a list of 4 text.Text objects>
```



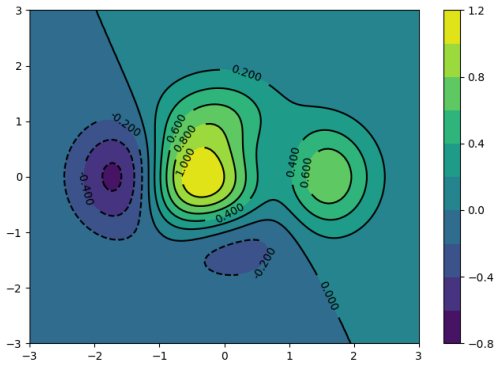
Der Zusammenhang zwischen Funktionswert und Farbe wird durch eine Farbpalette, eine so genannte »color map«, vermittelt. Die bisher verwendete Standardeinstellung läuft unter dem Namen `jet`. matplotlib stellt eine ganze Reihe verschiedener Farbpaletten zur Verfügung, von denen einige auch gut für Konturplots geeignet sind. Eine Zusammenstellung der Farbpaletten aus der matplotlib-Galerie ist nachfolgend abgebildet.



Im folgenden Beispiel wählen wir mit `viridis` eine der Farbpaletten, deren Helligkeit gleichmäßig variiert und damit auch in einer Schwarz-Weiß-Darstellung eine adäquate Darstellung der Farben liefert. Mit Hilfe der Funktion `contourf` füllen wir die Flächen zwischen den Konturlinien. Damit die Linien deutlicher sichtbar

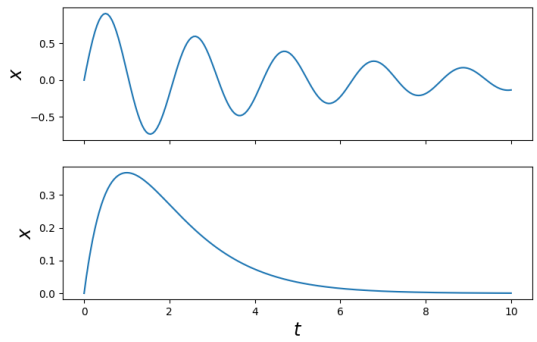
werden, werden sie in schwarz dargestellt.

```
In [47]: plt.clf()
   ...: levels = 10
   ...: contourset = plt.contourf(x, y, z, levels, cmap='viridis')
   ...: plt.colorbar(contourset)
   ...: contourlines = plt.contour(x, y, z, levels, colors=('k',))
   ...: plt.clabel(contourlines, inline=1)
Out[47]: <a list of 11 text.Text objects>
```



Vor allem bei der Erstellung von Grafiken für Publikationen steht man gelegentlich vor der Aufgabe, mehrere Abbildungen zu einer einzigen Abbildung zusammenzufassen. Man verwendet hierfür die `subplots`-Funktion, die in den ersten beiden Argumenten die Zahl der Zeilen und der Spalten enthält. Wir wollen zwei Grafiken übereinandersetzen und wählen daher 2 Zeilen und 1 Spalte. Neben der Gesamtgröße der Abbildung geben wir noch an, dass die *x*-Achse aus der unteren Abbildung auch für die obere Abbildung gelten soll. `subplots` gibt ein `figure`-Objekt zurück, das sich auf die ganze Abbildung bezieht sowie in unserem Fall zwei `axes`-Objekte, die sich jeweils auf die Unterabbildungen beziehen. Hier wird die Unterscheidung zwischen Abbildung und Unterabbildungen deutlich, die weiter oben bereits angedeutet wurde. Nachdem die Unterabbildungen angelegt wurden, können mit Hilfe des entsprechenden `axes`-Objekts Änderungen an den Unterabbildungen vorgenommen werden. Wir zeichnen hier konkret in jeder Unterabbildung einen Funktionsgraphen und nehmen eine Achsenbeschriftung vor.

```
In [48]: tvals = np.linspace(0, 10, 200)
   ...: x0vals = np.exp(-0.2*tvals)*np.sin(3*tvals)
   ...: x1vals = tvals*np.exp(-tvals)
   ...: fig, (ax0, ax1) = plt.subplots(2, 1, figsize=(8, 5), sharex=True)
   ...: ax0.plot(tvals, x0vals)
   ...: ax1.plot(tvals, x1vals)
   ...: ax1.set_xlabel('$t$', size='xx-large')
   ...: ax0.set_ylabel('$x$', size='xx-large')
   ...: ax1.set_ylabel('$x$', size='xx-large')
Out[48]: <matplotlib.text.Text at 0x7f48aeb96278>
```



Mit matplotlib sind auch dreidimensionale Darstellungen möglich, zumindest in einem gewissen Umfang, der für viele Zweck ausreicht. In dem folgenden Beispiel wird eine Funktion zweier Variablen dreidimensional darge-

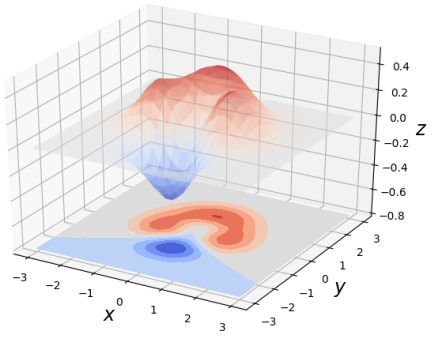
stellt. Zusätzlich wird in der *x*-*y*-Ebene eine Projektion der Funktionsdaten gezeigt. Dreidimensionale Darstellungen erfordern einen Import aus dem matplotlib Toolkit, der zunächst vorgenommen wird. Anschließend wird die darzustellende Funktion auf einem Gitter ausgewertet. Zudem wählen wir eine Farbpalette, die wir sowohl für die dreidimensionale Darstellung als auch für die Projektion verwenden.

Nachdem wir eine Abbildung unter der Variable `fig` erzeugt haben, können wir für eine darin enthaltene Unterabbildung `ax` eine dreidimensionale Projektionsdarstellung festlegen. Anschließend können wir mit Hilfe der zuvor berechneten Funktionsdaten eine dreidimensionale Darstellung der Funktion zeichnen lassen. Die Argumente `rstride` und `cstride` geben an, in welchen Abständen bezogen auf die Gitterweite Schnittlinien gezeichnet werden. Das letzte Argument, `alpha`, betrifft einen Aspekt der Farbdarstellung, den wir bis jetzt noch nicht besprochen hatten. Der Alphakanal gibt zusätzlich beispielsweise zu den Farbkanälen R, G und B die Transparenz der Farbe an. Im Beispiel wird die Oberfläche also teilweise transparent dargestellt.

Die Projektion in die *x*-*y*-Ebene stellen wir wie bereits besprochen mit Hilfe der Funktion `contourf` dar, wobei wir in diesem Beispiel auf die Darstellung von Konturlinien verzichten. Bei der Anwendung in einer dreidimensionalen Darstellung müssen wir noch die Ausrichtung der Projektsebene mit Hilfe der Normalenrichtung `zdir` und die Lage mit Hilfe von `offset` spezifizieren. Durch eine entsprechende Wahl von `zdir` wäre es auch möglich, Projektionen in die *x*-*z*- und die *y*-*z*-Ebene vorzunehmen.

Abschließend setzen wir in unserem Beispiel die Achsenbeschriftung und erweitern den Wertebereich der *z*-Achse, um die Projektion darstellen zu können. Zum Schluss wird die Abbildung, die hier im Gegensatz zu den meisten Beispielen dieses Kapitels objektorientiert erstellt wurde, dargestellt.

```
In [49]: from mpl_toolkits.mplot3d import Axes3D
   ...: x, y = np.mgrid[-3:3:30j, -3:3:30j]
   ...: z = (x**2+y**3)*np.exp(-x**2-y**2)
   ...: cmap = 'coolwarm'
   ...:
   ...: fig = plt.figure()
   ...: ax = fig.gca(projection='3d')
   ...: ax.plot_surface(x, y, z, rstride=1, cstride=1, cmap=cmap, alpha=0.5)
   ...: cset = ax.contourf(x, y, z, zdir='z', offset=-0.8, cmap=cmap)
   ...: ax.set_xlabel('$x$', size='xx-large')
   ...: ax.set_ylabel('$y$', size='xx-large')
   ...: ax.set_zlabel('$z$', size='xx-large')
   ...: ax.set_zlim(-0.8, 0.5)
   ...:
   ...: plt.draw()
```



Es sei noch angemerkt, dass sich die dreidimensionale Darstellung im grafischen Benutzerinterface nach Belieben in alle Richtungen drehen lässt, so dass sich bequem eine geeignete Blickrichtung finden lässt, die eine instruktive Sicht auf die dargestellten Daten ermöglicht.

In diesem Kapitel mussten wir uns auf einige für die Anwendung wichtige Problemstellungen konzentrieren und konnten daher keinen vollständigen Überblick über die Möglichkeiten von matplotlib geben. Um einen detaillierteren Einblick zu bekommen, bietet sich ein Blick in die bereits mehrfach erwähnte Beispielgalerie an. Dort ist auch der Code verfügbar, mit dem die in der Galerie gezeigten Beispiele erzeugt wurden. Weitere Informationen über die verfügbaren Funktionen und die jeweiligen Argumente liefert die umfangreiche Dokumentation, die ebenfalls auf der matplotlib-Webseite www.matplotlib.org bereitgestellt ist.

4.2 Erstellung von Grafiken mit PyX

In diesem Kapitel wollen wir als zweites Paket zur Erzeugung von Abbildungen `PyX` besprechen. Es handelt sich dabei um ein Paket, das zunächst von André Wobst und Jörg Lehmann während ihrer Doktorandenzeit an der Universität Augsburg entwickelt wurde. Zwischenzeitlich leistete Michael Schindler einige interessante Beiträge, und heute wird PyX von den beiden Erstgenannten weiterentwickelt.

PyX eignet sich im Gegensatz zu dem zuvor besprochenen matplotlib-Paket auch sehr gut zur Erstellung von schematischen Darstellungen. Wir wollen diesen Aspekt daher als erstes besprechen. Anschließend werden wir zeigen, wie in PyX auch grafische Darstellungen von Daten erzeugt werden können. Interessant ist, dass man für beide Anwendungen nur ein einziges Paket benötigt und die beiden Darstellungsarten auch kombinieren kann. Wie schon bei matplotlib würde es zu weit führen, alle Möglichkeiten von PyX zu diskutieren. Wir treffen daher im Folgenden eine Auswahl wichtiger anwendungsrelevanter Aspekte und verweisen ansonsten auf die Dokumentation unter pyx.sf.net sowie die Beispielseiten und die Galerie. Letztere ist derzeit weniger umfangreich als bei matplotlib, was jedoch nichts über den Funktionalitätsumfang aussagt.

Bis zur Version 0.12.1 war PyX nur unter Python 2 lauffähig und seit der Version 0.13 ist es ausschließlich unter Python 3 funktionsfähig. Die installierte Version von PyX kann man folgendermaßen herausfinden:

```
In [1]: import pyx
   ...: pyx.__version__
Out[1]: '0.13'
```

Die folgende Diskussion bezieht sich auf die Version 0.13 für Python 3. Zu Beginn eines Pythonskripts wird man zuerst PyX sowie bei Bedarf weitere Pakete importieren. Im Gegensatz zu matplotlib importieren wir alle Module, da die Auswirkung auf den Namensraum wesentlich überschaubarer ist als bei matplotlib. Natürlich kann man sich aber auch darauf beschränken, nur die jeweils benötigten Module zu importieren.

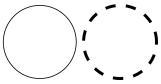
```
In [2]: from pyx import *
```

Ein wesentlicher Aspekt bei der Erstellung schematischer Abbildungen ist das Zeichnen von Pfaden, eine eventuelle Dekorierung von Pfaden sowie das Füllen von Flächen, die durch gegebene Pfade begrenzt werden. In PyX erfolgt das Zeichnen grundsätzlich auf einem »canvas«, also einer Leinwand. Wir werden später noch sehen, dass ein Canvas eine sehr flexible Struktur darstellt, die transformiert, also beispielsweise skaliert, gespiegelt oder geschert werden kann. Zudem kann ein Canvas in einen anderen Canvas eingefügt werden. Bei PyX wird man im Allgemeinen sehr früh einen Canvas bereitstellen, um darin zeichnen zu können. In unserem ersten Beispiel definieren wir uns einen Canvas. Diese Klasse ist im `canvas`-Modul definiert, so dass die Initialisierung mit der ersten Zeile aus dem folgenden Code erfolgt. Der Code in der zweiten Zeile verlangt, dass auf dem gerade definierten Canvas ein Kreis um den Ursprung mit Radius 1 gemalt wird. So lange nichts anderes angegeben wird, wird der Kreis mittels einer schwarzen durchgezogenen Linie mit der Defaultbreite dargestellt.

```
In [3]: c = canvas.canvas()
   ...: c.stroke(path.circle(0, 0, 1))
```

Abweichungen von den Defaulteinstellungen kann man in einer Attributliste angeben. Zu beachten ist, dass hier immer eine Liste anzugeben ist, selbst dann, wenn nur ein einziges Attribut neu definiert wird. Mit Hilfe der folgenden Codezeile wird auf dem bereits existierenden Canvas ein weiterer Kreis gemalt, der nun durch eine dickere, gestrichelte Linie dargesellt wird.

```
In [4]: c.stroke(path.circle(2.2, 0, 1),
   ...:          [style.linestyle.dashed, style.linewidth.THIck])
```



Neben durchgezogenen und gestrichelten Linien gibt es auch punktierte und strichpunktierte Linien, wobei sich die Strichlänge bei Bedarf einstellen lässt. Die Liniendicke wird umso größer, je mehr Buchstaben des Wortes

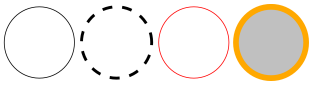
»thick« groß geschrieben werden. Von »THIN« bis »THICK« ändert sich die Linienbreite in Schritten von $\sqrt{2}$ und umfasst dabei einen Umfang von etwa einem Faktor 45.

Ein weiterer Parameter beim Malen des Pfades ist die Farbe, die man ganz ähnlich wie bei matplotlib auf verschiedene Weisen festlegen kann. Im `rgb`-System sind wenige Farben per Namen ansprechbar. Es handelt sich um rot (`red`), grün (`green`), blau (`blue`), weiß (`white`) und schwarz (`black`). Eine wesentlich größere Anzahl von Farbnamen ist im `cmyk`-System definiert, wobei `cmyk` für Cyan, Magenta, Gelb (`Yellow`) und Schwarz (`blacK`) steht. Einen roten Kreis kann man also folgendermaßen erhalten:

```
In [5]: c.stroke(path.circle(4.4, 0, 1), [color.rgb.red])
```

Man kann aber Pfade nicht nur zeichnen, sondern auch füllen. Hierfür gibt es zwei Möglichkeiten, die wir nun ansehen wollen. Man kann den Pfad durch Füllen dekorieren, indem man zu den Attributen `deco.filled()` hinzufügt. Die Füllung kann man wiederum mit einer Liste von Attributen genauer spezifizieren. In unserem Beispiel wollen wir als Farbe ein helles Grau festlegen. Lässt man in `color.grey()` das Argument von 0 bis 1 laufen, so erhält man Graustufen zwischen Schwarz und Weiß. Anhand der Linienfarbe demonstriert unser Beispiel auch die Möglichkeit, Farben mit Hilfe des `hsb`-Systems festzulegen, wobei der »value« in dieser Bezeichnung durch »brightness« ersetzt wurde, ein ebenfalls übliches Akronym für dieses zusätzlich auf dem Farbwert und der Sättigung basierenden Farbsystems.

```
In [6]: c.stroke(path.circle(6.6, 0, 1),
   ...:          [color.hsb(0.11, 1, 1), style.linewidth.THICK,
   ...:           deco.filled([color.grey(0.7)])])
```



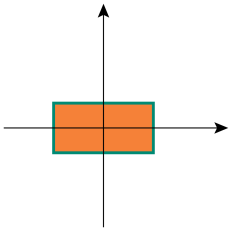
Ein alternativer Weg, einen gefüllten Kreis zu malen, der vollkommen äquivalent zu dem vorherigen Vorgehen ist, besteht darin, das Füllen des Kreises in den Vordergrund zu stellen. Dabei verwendet man statt der `stroke`-Methode die `fill`-Methode. Dabei hat man die Wahl, den Pfad selbst durch Angabe des `deco.stroked()`-Methode zu malen oder dies nicht zu tun. Das folgende Beispiel unterscheidet sich von dem vorhergehenden lediglich durch die Wahl der Farben, wobei wir hier noch die Verwendung des `rgb`-Systems demonstrieren, das wir bereits von matplotlib kennen.

```
In [7]: c.fill(path.circle(8.8, 0, 1), [color.rgb(1, 0.5, 0.5),
   ...:          deco.stroked([style.linewidth.THICK, color.rgb(0.5, 0.5, 1)])])
```

Selbstverständlich stehen als Pfade nicht nur Kreise zur Verfügung. Im folgenden Beispiel finden zwei weitere Pfade Verwendung, nämlich Linien (`path.line`) und Rechtecke (`path.rect`). Im ersten Fall sind die x- und y-Koordinate von Anfangs- und Endpunkt anzugeben, während im zweiten Fall die ersten beiden Argumente einen Eckpunkt angeben und die beiden folgenden Punkte die Breite und die Höhe des Rechtecks spezifizieren.

Zusätzlich zu diesen Pfaden demonstriert das Beispiel noch eine weitere Pfaddekoration, nämlich das Platzieren von Pfeilen am Anfang (`barrow`) und/oder Ende (`earrow`) eines Pfades. Dabei lässt sich die Größe des Pfeils festlegen. Hier verlangen wir mit `large` Pfeile, die etwas größer als die Defaultpfeile sind. Mit Hilfe von Großbuchstaben lässt sich die Pfeilgröße weiter erhöhen, ähnlich wie dies bei der Linienbreite der Fall war. Kleinere Pfeile erhält man mit Hilfe von `small`. Schließlich demonstriert das Beispiel die Verwendung von benannten Farben im `cmyk`-System, das oben schon kurz erläutert wurde.

```
In [8]: c = canvas.canvas()
   ...: c.fill(path.rect(-1, -0.5, 2, 1),
   ...:        [color.cmyk.Orange, deco.stroked([color.cmyk.PineGreen,
   ...:                                          style.linewidth.THick])])
   ...: c.stroke(path.line(-2, 0, 2.5, 0), [deco.earrow.large])
   ...: c.stroke(path.line(0, 2.5, 0, -2), [deco.barrow.large])
```



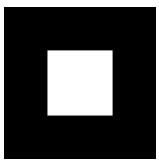
Neben der Verwendung vordefinierter Pfade erlaubt es PyX auch, Pfade aus mehreren Pfadsegmenten zu bilden. Im folgenden Beispiel wird ein Pfad aus vier geraden Linien zusammengesetzt. Zunächst wird mit `path.moveto()` ein Startpunkt gewählt, der hier im Koordinatenursprung liegt. Von diesem Punkt ausgehend wird dann eine Linie zu einem weiteren Punkt gezogen, der in absoluten Koordinaten angegeben wird und als nächster Ausgangspunkt dient. Auf diese Weise wird hier ein Pfad konstruiert, der ein Quadrat beschreibt.

Obwohl der Pfad schließlich an den Ausgangspunkt zurückkehrt, handelt es sich nicht um einen geschlossenen Pfad. Dies wird deutlich, wenn man die Linienbreite sehr groß wählt, wie es hier der Fall ist. Die Funktionsweise von `wscale` und weiteren Längenskalen wird etwas weiter unten genauer besprochen. Die letzte Zeile in dem folgenden Beispiel führt zwar dazu, dass der eingangs definierte Pfad `p` gezeichnet wird, aber die Linie ist offenbar nicht geschlossen.

```
In [9]: p = path.path(path.moveto(0, 0),
   ...:               path.lineto(2, 0),
   ...:               path.lineto(2, 2),
   ...:               path.lineto(0, 2),
   ...:               path.lineto(0, 0))
   ...: unit.set(wscale=40)
   ...: c = canvas.canvas()
   ...: c.stroke(p)
```

Das Schließen eines Pfades erreicht man durch Anhängen von `path.closepath()`, wobei es nicht einmal notwendig ist, zum Ausgangspunkt zurückzukehren. `closepath` fügt bei Bedarf selbstständig das fehlende Liniensegment zum Ausgangspunkt des Pfades ein. Die Quadratkontur wird jetzt vollständig dargestellt.

```
In [10]: p = path.path(path.moveto(0, 0),
   ...:               path.lineto(2, 0),
   ...:               path.lineto(2, 2),
   ...:               path.lineto(0, 2),
   ...:               path.closepath())
   ...: c = canvas.canvas()
   ...: c.stroke(p)
```



Für die folgenden Beispiele stellen wir die stark erhöhte Linienbreite zunächst einmal wieder auf die Defaultbreite zurück.

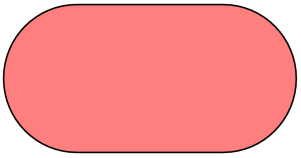
```
In [11]: unit.set(wscale=1)
```

In den beiden vorhergehenden Beispielen haben wir die Pfadsegmente mit Hilfe von absoluten Koordinaten festgelegt. Statt `lineto` kann man aber auch `rlineto` verwenden, in dem eine Linie relativ zum aktuellen Endpunkt des Pfades angegeben wird. Das folgende Beispiel demonstriert dies anhand der Simulation einer Zufallsbewegung, bei der in jedem Schritt um einen festen Abstand in eine zufällige Richtung weitergegangen wird.

```
In [12]: from numpy import random
   ...: from math import pi, cos, sin
   ...:
   ...: directions = 2*pi*random.random(1000)
   ...: pathelems = [path.rlineto(0.1*cos(dir), 0.1*sin(dir)) for dir in␣
↪directions]
   ...: p = path.path(path.moveto(0, 0), *pathelems)
   ...:
   ...: c = canvas.canvas()
   ...: c.stroke(p)
```

Neben Geraden kann man auch Kreissegmente verwenden, um Pfade zu konstruieren. Dies wird hier an einer stadionförmigen Kontur gezeigt.

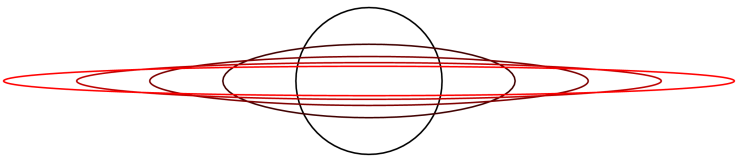
```
In [13]: p = path.path(path.moveto(-1, -1), path.lineto(1, -1),
   ...:                path.arc(1, 0, 1, 270, 90), path.lineto(-1, 1),
   ...:                path.arc(-1, 0, 1, 90, 270), path.closepath())
   ...: c = canvas.canvas()
   ...: c.stroke(p, [deco.filled([color.rgb(1, 0.5, 0.5)])])
```



Zu Beginn hatten wir bereits darauf hingewiesen, dass es die Möglichkeit gibt, einen Canvas zu transformieren. Dabei müssen die Objekte im Canvas transformiert werden, also letztendlich die Pfade. Es ist auch möglich, Pfade vor dem Zeichnen einer Transformation zu unterwerfen, wie wir anhand einiger Beispiel demonstrieren wollen.

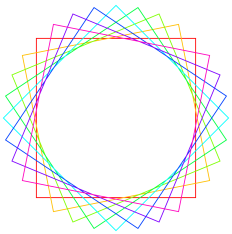
Aus dem vordefinierten Kreispfad lassen sich sehr einfach auch Ellipsen erzeugen, indem man die Skalierungstransformation anwendet und dabei unterschiedliche Skalierungsfaktoren in x- und y-Richtung verwendet. Das folgende Beispiel zeigt eine Reihe von Ellipsen mit unterschiedlicher Exzentrizität.

```
In [14]: p = path.circle(0, 0, 1)
   ...: ncircs = 5
   ...: c = canvas.canvas()
   ...: for n in range(ncircs):
   ...:     c.stroke(p, [trafo.scale(n+1, 1/(n+1)), color.hsb(1, 1, n/(ncircs-1))])
```



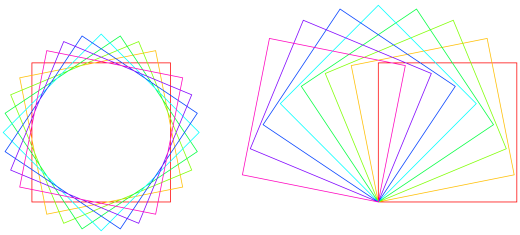
Will man die Hauptachsen der Ellipse nicht in Richtung der Koordinatenachsen legen, so kann man die Ellipse anschließend rotieren. Wir zeigen das Rotieren von Pfaden am Beispiel eines Quadrats, dessen Mittelpunkt im Ursprung liegt. Da die Rotation immer um den Ursprung herum erfolgt, wird das Quadrat um seinen Mittelpunkt gedreht.

```
In [15]: p = path.rect(-2, -2, 4, 4)
   ...: nrects = 8
   ...: c = canvas.canvas()
   ...: for n in range(nrects):
   ...:     c.stroke(p, [trafo.rotate(90*n/nrects), color.hsb(n/nrects, 1, 1)])
```



Möchte man stattdessen das Quadrat beispielsweise um seine linke untere Ecke drehen, so muss man diese Ecke zunächst in den Ursprung verschieben, um dann die Drehung durchzuführen. Anschließend erfolgt die Rückverschiebung an die ursprüngliche Position oder, wie in diesem Beispiel, an eine verschobene Position rechts neben der bereits existierenden Abbildung.

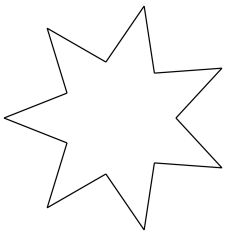
```
In [16]: for n in range(nrects):
   ...:     c.stroke(p, [trafo.translate(2, 2).rotated(
                         90*n/nrects).translated(8, -2),
                         color.hsb(n/nrects, 1, 1)])
```



Dieses Beispiel zeigt die Hintereinanderausführung von Transformationen, wobei die Transformationen von links nach rechts abgearbeitet werden. Außerdem ist zu beachten, dass die erste Verschiebung hier mit `translate` aufgerufen wird, während die zweite sowie eventuell noch weiter folgende Verschiebungen mit `translated` aufgerufen werden. Entsprechend ist in dem obigen Beispiel auch statt `rotate` die Methode `rotated` zu verwenden.

Neben Verschiebungen, Skalierungen und Drehungen stehen in PyX auch noch Scherungen (`slant`) und Spiegelungen (`mirror`) zur Verfügung. Das folgende Beispiel zeigt, wie man aus einem einzigen Liniensegment durch Drehung und Spiegelung einen sternförmigen Pfad erzeugen kann. Das Argument der `mirror`-Methode gibt dabei den Winkel an, unter der die Spiegelachse durch den Ursprung verläuft.

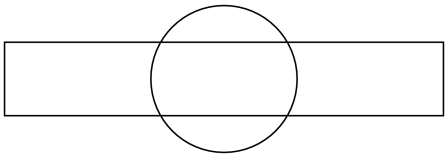
```
In [17]: nstar = 7
   ...: alpha = 360/nstar
   ...: p = path.line(1, 0, 2*cos(pi*alpha/360), 2*sin(pi*alpha/360))
   ...: c = canvas.canvas()
   ...: for n in range(nstar):
   ...:     c.stroke(p.transformed(trafo.rotate(alpha*n)))
   ...:     c.stroke(p.transformed(trafo.mirror(alpha/2).rotated(alpha*n)))
```



Gelegentlich ist es nützlich, Schnittpunkte von zwei Pfaden oder Pfadsegmente zwischen zwei vorgegebenen Schnittpunkten zur Verfügung zu haben. Zur Veranschaulichung zeichnen wir zunächst die beiden Pfade, die

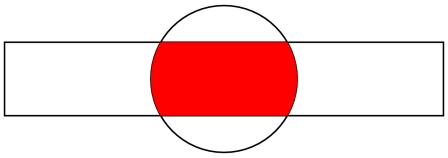
geschnitten werden sollen. Das sind hier ein Kreis und ein Rechteck.

```
In [18]: c = canvas.canvas()
   ...: p1 = path.circle(0, 0, 1)
   ...: p2 = path.rect(-3, -0.5, 6, 1)
   ...: c.stroke(p1)
   ...: c.stroke(p2)
```



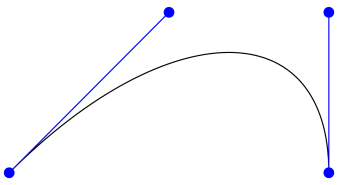
In der ersten Zeile des folgenden Codes wird der Pfad `p1`, also der Kreis, mit dem Pfad `p2`, dem Rechteck, geschnitten. Dabei werden zwei Tupel, hier `intersect_circle` und `intersect_rect` genannt, erzeugt, die entsprechend den vier Schnittpunkten jeweils vier Werte enthalten, die in einer Parametrisierung der Pfade die Schnittpunkte angeben. In der zweiten und dritten Zeile werden die beiden Pfade an den Schnittpunkten aufgetrennt, so dass zweimal vier Teilpfade entstehen. Diese werden anschließend so zusammengefügt, dass am Ende der Schnittbereich der beiden Pfade farbig gefüllt werden kann.

```
In [19]: intersect_circle, intersect_rect = p1.intersect(p2)
   ...: circle_subpaths = p1.split(intersect_circle)
   ...: rect_subpaths = p2.split(intersect_rect)
   ...: p = (circle_subpaths[0] << rect_subpaths[1]
   ...:       << circle_subpaths[2] << rect_subpaths[3])
   ...: c.fill(p, [color.rgb.red])
```



Bei komplizierteren Pfaden ist es unter Umständen nicht ganz einfach, die Tangenten- oder Normalenrichtung zu bestimmen. In solchen Fällen kann man sich von PyX helfen lassen. Zur Illustration konstruieren wir zunächst eine kubische Bézierkurve, die durch vier Punkte charakterisiert ist. Dabei handelt es sich um eine kubische Kurve, für die der erste und vierte Punkt den Anfangs- und den Endpunkt festlegen. Die Verbindungslinien vom ersten zum zweiten sowie vom dritten zum vierten Punkt bestimmen zudem die Kurvensteigung im Anfangs- und Endpunkt, wie im Beispiel durch die blauen Geraden dargestellt ist.

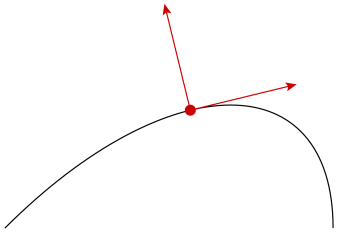
```
In [20]: x = (0, 3, 6, 6)
   ...: y = (0, 3, 3, 0)
   ...: p = path.curve(x[0], y[0], x[1], y[1], x[2], y[2], x[3], y[3])
   ...: c = canvas.canvas()
   ...: c.stroke(p)
   ...: for xc, yc in zip(x, y):
   ...:     c.fill(path.circle(xc, yc, 0.1), [color.rgb.blue])
   ...: c.stroke(path.line(x[0], y[0], x[1], y[1]), [color.rgb.blue])
   ...: c.stroke(path.line(x[2], y[2], x[3], y[3]), [color.rgb.blue])
```



Im folgenden Beispiel soll bei der Hälfte der Kurve der Tangenten- und der Normalenvektor eingezeichnet werden. Dazu wird in der dritten Zeile zunächst mit `p.arclen()` die Länge des Pfads `p` bestimmt und dann mittels `p.arclentoparam()` der Parameter berechnet, der der Hälfte der Pfadlänge entspricht. An diesem Punkt kann man dann mit `p.tangent()` ein Geradenstück mit vorgegebener Länge in Tangentialrichtung erzeugen, das hier mit einem Pfeil am Ende gezeichnet wird. Um den Normalenvektor zu zeichnen, wird der Tangentialvektor

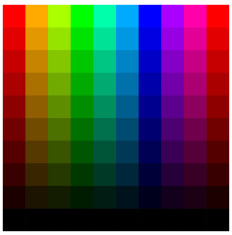
einfach um 90° im Gegenuhrzeigersinn gedreht. Dabei ist allerdings zu beachten, dass der Vektor zunächst in den Ursprung verschoben, dort gedreht, und anschließend wieder in den Ausgangspunkt zurückverschoben werden muss.

```
In [21]: c = canvas.canvas()
   ...: c.stroke(p)
   ...: paramhalf = p.arclentoparam(0.5*p.arclen())
   ...: x, y = p.at(paramhalf)
   ...: mycolor = color.rgb(0.8, 0, 0)
   ...: c.fill(path.circle(x, y, 0.1), [mycolor])
   ...: c.stroke(p.tangent(paramhalf, length=2), [deco.earrow, mycolor])
   ...: c.stroke(p.tangent(paramhalf, length=2), [deco.earrow, mycolor,
   ...:                 trafo.translate(-x, -y).rotated(90).translated(x, y)])
```



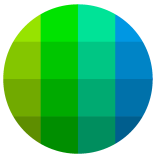
PyX bietet auch die Möglichkeit, mit Hilfe von Pfaden aus einem Canvas einen Teil herauszuschneiden. Zunächst zeigen wir den vollständigen Canvas, der eine matrixförmige Anordnung von gefärbten Quadraten im hsb-System enthält.

```
In [22]: c = canvas.canvas()
   ...: for nx in range(10):
   ...:     for ny in range(10):
   ...:         c.fill(path.rect(nx, ny, 1, 1), [color.hsb(nx/9, 1, ny/9)])
```



Nun legen wir beim Initialisieren des Canvas einen so genannten »clipping path« fest, in unserem Fall einen Kreis, der die Darstellung des Canvas auf das Innere dieses Pfads begrenzt. Damit wird innerhalb des Kreises der entsprechende Ausschnitt aus der zuvor dargestellten Farbmatrix gezeigt.

```
In [23]: c = canvas.canvas([canvas.clip(path.circle(4, 7, 2))])
   ...: for nx in range(10):
   ...:     for ny in range(10):
   ...:         c.fill(path.rect(nx, ny, 1, 1), [color.hsb(nx/9, 1, ny/9)])
```



Text ist ein wichtiger Bestandteil von grafischen Darstellungen, ganz gleich ob in Schemazeichnungen oder in Graphen. PyX überträgt die Aufgabe des Textsatzes an TeX oder LaTeX, woher auch das X im Namen des Pakets stammt. Damit bietet PyX die Möglichkeit, Grafiken mit komplexen Texten, bei Bedarf auch ganzen Paragraphen zu versehen. So ist es zum Beispiel möglich, Poster mit Hilfe von PyX zu gestalten. Im Folgenden werden wir einige grundlegende Aspekte von Text in PyX betrachten.

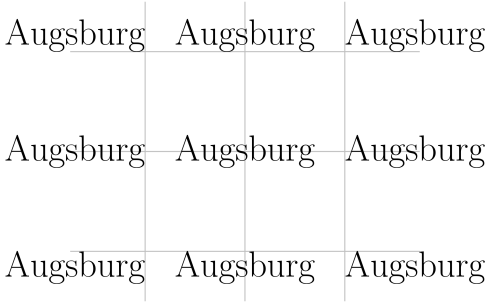
Im folgenden Beispiel wird zunächst ein Achsenkreuz am Ursprung gezeichnet, das hier lediglich zur Orientierung dienen soll. Von Bedeutung ist vor allem die letzte Zeile, in der der Text mit Hilfe der `text`-Methode gesetzt

wird. Die ersten beiden Argumente geben den Punkt an, an dem der Text gesetzt wird, hier also der Koordinatenursprung. Das dritte Argument enthält den Text, der hier zunächst in einer Variable gespeichert wurde. Wie wir weiter unten noch sehen werden, kann dieser Text beispielsweise auch Mathematikanteile entsprechend der TeX- oder LaTeX-Syntax enthalten. Abschließend folgt eine Liste von Attributen. In diesem Fall wird lediglich dafür gesorgt, das der Text etwas größer dargestellt wird. Die möglichen Größenattribute folgen dabei der Vorgabe von TeX wo in aufsteigender Größe die Einstellungen `tiny`, `scriptsize`, `footnotesize`, `small`, `normalsize`, `large`, `Large`, `LARGE`, `huge` und `Huge` definiert sind. Wie an der Ausgabe zu sehen ist, wird der Text mit dem linken Ende der Basislinie an dem im `text`-Aufruf angegebenen Punkt positioniert.

```
In [24]: c = canvas.canvas()
   ...: mytext = 'Augsburg'
   ...: mycolor = color.grey(0.7)
   ...: c.stroke(path.line(-1, 0, 1, 0), [mycolor])
   ...: c.stroke(path.line(0, -1, 0, 1), [mycolor])
   ...: c.text(0, 0, mytext, [text.size.huge])
```

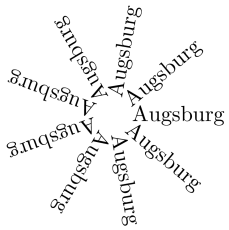
Die Attributliste von `text` kann neben der Größenabgabe vor allem auch Angaben zur Positionierung des Textes relativ zum angegebenen Referenzpunkt enthalten. Im nächsten Beispiel werden jeweils drei wichtige Varianten der horizontalen und vertikalen Positionierung dargestellt. `halign.right`, `halign.center` und `halign.left` sorgen dafür, dass der Referenzpunkt rechts vom Text, in dessen Mitte oder links vom Text liegt. Die vertikale Positionierung mit `valign.top`, `valign.middle` und `valign.bottom` führt dazu, dass der Referenzpunkt am oberen Ende, in der Mitte bzw. am unteren Ende des den Text umschließenden Rahmens liegt. Auf diese Weise sind sehr flexible Positionierungen möglich.

```
In [25]: c = canvas.canvas()
   ...: mytext = 'Augsburg'
   ...: mycolor = color.grey(0.7)
   ...: for nx in range(3):
   ...:     c.stroke(path.line(2*nx, 0, 2*nx, 6), [mycolor])
   ...: for ny in range(3):
   ...:     c.stroke(path.line(-1.5, 2*ny+1, 5.5, 2*ny+1), [mycolor])
   ...: for nx, xpos in enumerate((text.halign.right,
   ...:                            text.halign.center,
   ...:                            text.halign.left)):
   ...:     for ny, ypos in enumerate((text.valign.top,
   ...:                                text.valign.middle,
   ...:                                text.valign.bottom)):
   ...:         c.text(2*nx, 2*ny+1, mytext, [xpos, ypos, text.size.huge])
```



Natürlich können Texte auch transformiert werden, wie wir hier anhand der Drehung des Textes illustrieren. Dabei wird der Text zunächst aus dem Ursprung, der als Drehpunkt fungiert, etwas herausgerückt.

```
In [26]: c = canvas.canvas()
   ...: for n in range(9):
   ...:     c.text(0, 0, mytext, [text.valign.middle,
                                  trafo.translate(0.3, 0).rotated(40*n)])
```



Wie bereits erwähnt, unterstützt PyX sowohl den Textsatz mit TeX als auch mit LaTeX. Letzteres basiert auf TeX und stellt Funktionalität zur Verfügung, die den Textsatz erleichtert. In LaTeX kann man die TeX-Syntax verwenden, aber umgekehrt wird spezifische LaTeX-Syntax nicht von TeX verstanden. Daher sollte man sich zunächst überlegen, welche Syntax man verwenden möchte. Defaultmäßig ist TeX voreingestellt, das auch explizit mit Hilfe von `text.set(text.TexRunner)` verlangt werden kann. LaTeX wählt man mit Hilfe von `text.set(text.LatexRunner)` aus. Die folgenden beiden Beispiele sind äquivalent. Allerdings wird im ersten Beispiel die TeX-Syntax für einen Bruch benutzt, während im zweiten Beispiel die LaTeX-Syntax verwendet wird.

```
In [27]: text.set(text.TexRunner)
   ...: c = canvas.canvas()
   ...: c.text(0, 0, '$x = {1\over2}$')
```

$$x = \frac{1}{2}$$

```
In [28]: text.set(text.LatexRunner)
   ...: c = canvas.canvas()
   ...: c.text(0, 0, r'$x = \frac{1}{2}$')
```

$$x = \frac{1}{2}$$

Das letzte Textbeispiel zeigt, wie in LaTeX eine etwas komplexere mathematische Formel gesetzt werden kann. Außerdem ist in der letzten Zeile zu sehen, wie man die Textgröße ohne Verwendung der auf TeX zurückgehenden Schlüsselworte bei Bedarf stufenlos einstellen kann.

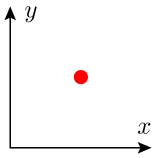
```
In [29]: c = canvas.canvas()
   ...: formula = r'$\displaystyle m\ddot{\vec r} = -\gamma\frac{Mm}{r^3}\vec r$'
   ...: c.text(0, 0, formula, [text.size(2)])
```

$$m\ddot{\vec{r}} = -\gamma\frac{Mm}{r^3}\vec{r}$$

Wir haben in verschiedenen Beispielen immer wieder Längen explizit festgelegt, zum Beispiel die Textgröße, die Linienbreite oder die Pfeilgröße. PyX stellt jedoch auch die Möglichkeit zur Verfügung, Längenskalen global zu definieren. Dabei wird Wert darauf gelegt, visuell unterschiedliche Längen auch unabhängig voneinander verändern zu können. So gibt es `uscale`, `vscale`, `wscale` und `xscale`, die wir jetzt ausgehend von einer Referenzabbildung, in der alle Skalen auf Eins gesetzt sind, verändern wollen, um ihre Auswirkung vorzustellen.

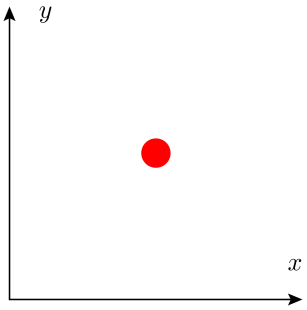
```
In [30]: def testfigure():
   ...:     c = canvas.canvas()
   ...:     c.stroke(path.path(path.moveto(2, 0), path.lineto(0, 0),
   ...:              path.lineto(0, 2)), [deco.barrow, deco.earrow])
   ...:     c.fill(path.circle(1, 1, 0.1), [color.rgb.red])
   ...:     c.text(2, 0.2, '$x$', [text.halign.right])
   ...:     c.text(0.2, 2, '$y$', [text.valign.top])
   ...:     return c
```

```
In [31]: unit.set(uscale=1, vscale=1, wscale=1, xscale=1)
   ...: testfigure()
```



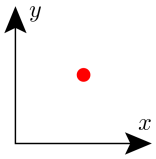
Zunächst verändern wir den Wert von `uscale`, wodurch Distanzen verändert werden. Dies betrifft in unserer Testabbildung die Länge der Achsen und die Größe der roten Kreisfläche. Unverändert bleiben die Linienbreite der Achsen, die Pfeilgröße sowie die Schriftgröße. Allerdings ist der Abstand der Achsenbeschriftung von der jeweiligen Achse nun größer geworden.

```
In [32]: unit.set(uscale=2, vscale=1, wscale=1, xscale=1)
    ...: testfigure()
```



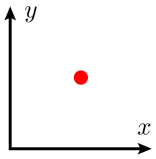
Im nächsten Beispiel wird `uscale` auf seinen ursprünglichen Wert zurückgesetzt und dafür `vscale` verdoppelt. Dadurch werden visuelle Elemente doppelt so groß dargestellt. In unserem Fall betrifft dies die Pfeile. Es würden aber zum Beispiel auch Symbole in einer Grafik oder Achsenticks vergrößert dargestellt werden.

```
In [33]: unit.set(uscale=1, vscale=2, wscale=1, xscale=1)
    ...: testfigure()
```



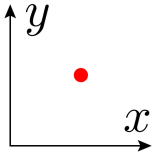
Der Parameter `wscale` beeinflusst alle Linienbreiten. Möchte man gleichzeitig auch die Pfeilgröße heraufsetzen, so könnte man zusätzlich noch `vscale` verändern.

```
In [34]: unit.set(uscale=1, vscale=1, wscale=2, xscale=1)
    ...: testfigure()
```



Als letztes verdoppeln wir den Wert von `xscale` und erreichen auf diese Weise, dass der gesamte Text in doppelter Größe ausgegeben wird. Damit spart man sich unter Umständen, Größenangaben in vielen `text`-Aufrufen zu ändern. Zudem sind die besprochenen Skalen nützlich, um über mehrere Abbildungen hinweg konsistente Längen zu verwenden.

```
In [35]: unit.set(uscale=1, vscale=1, wscale=1, xscale=2)
    ...: testfigure()
```



Für die folgenden Beispiele stellen wir alle Skalen wieder auf ihren Standardwert zurück.

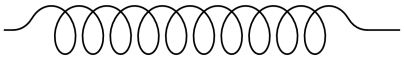
```
In [36]: unit.set(xscale=1)
```

PyX stellt so genannte Deformer zur Verfügung, die es erlauben, Pfade zu deformieren. Dies kann in bestimmten Fällen sehr nützlich sein. Mit Hilfe eines Deformers ist es zum Beispiel sehr leicht möglich, Kanten mit einem vorgegebenen Radius abzurunden. Zu diesem Zweck fügen wir in der Attributliste im folgenden Beispiel einen Aufruf der `smoothed`-Methode hinzu.

```
In [37]: box = path.rect(0, 0, 3, 2)
   ...: c = canvas.canvas()
   ...: c.stroke(box)
   ...: c.stroke(box, [deformer.smoothed(radius=0.5), trafo.translate(3.5, 0)])
   ...: c.stroke(box, [deformer.smoothed(radius=1), trafo.translate(7, 0)])
```

Besonders elegant ist die Erzeugung der Darstellung einer Feder aus einem Liniensegment durch die Verwendung des `cycloid`-Deformers, die im folgenden Beispiel dargestellt ist. Um einen optisch ansprechenden Eindruck zu erzielen, stellt man abhängig von der zur Verfügung stehenden Länge den Radius der Zykloide sowie die Zahl der halben Schleifen geeignet ein. Außerdem kann man mit `skipfirst` und `skiplast` dafür sorgen, dass ein Teil der Linie am Anfang und am Ende unverändert bleibt. Um den Übergang zwischen dem deformierten Teil und dem unveränderten Teil etwas zu glätten, kann wiederum der `smoothed`-Deformer zum Einstatz kommen.

```
In [38]: c = canvas.canvas()
   ...: c.stroke(path.line(0, 0, 5, 0), [deformer.cycloid(radius=0.3,
   ...:                                                   halfloops=21,
   ...:                                                   skipfirst=0.3*unit.t_cm,
   ...:                                                   skiplast=0.6*unit.t_cm),
   ...:                                  deformer.smoothed(radius=0.2)])
```



Eine Anwendung eines Deformers, bei der der ursprüngliche Pfad mit Hilfe einer »bounding box« erzeugt wird, zeigt das nächste Beispiel. Zunächst wird ein Text definiert, der erst später in einen Canvas eingefügt wird. Daher wird in der ersten Zeile statt `c.text` die Methode `text.text` verwendet. Dies erlaubt es uns, die »bounding box« dieses Textes zu bestimmen, anschließend mit `enlarged` zu vergrößern und uns mit `path` den zugehörigen Pfad zu beschaffen. Die Vergrößerung der »bounding box« wird hier mit `0.3*unit.t_cm` zu 0,3cm spezifiziert. In den Canvas wird dann zunächst der Pfad der vergrößerten »bounding box« in abgerundeter Form und mit roter Farbe gefüllt eingefügt. Anschließend kann dann der weiße Text zum Canvas hinzugefügt werden.

```
In [39]: mytext = text.text(0, 0, r'\textbf{\sffamily Hallo}', [color.grey(1)])
   ...: textbox = mytext.bbox().enlarged(0.3*unit.t_cm).path()
   ...: c = canvas.canvas()
   ...: c.stroke(textbox, [deco.filled([color.rgb.red]),
   ...:                    deformer.smoothed(radius=0.5)])
   ...: c.insert(mytext)
```

In vielen Fällen wird man die erzeugte Grafik auch in einer Datei speichern wollen. Dies kann in PyX genauso wie in matplotlib entweder in einem Vektorgrafik- oder einem Bitmapformat erfolgen. Für ersteres stehen aktuell die Ausgaben im PDF-Format sowie in Postscript und Encapsulated Postscript zur Verfügung. Daneben gibt es eine ganze Reihe von Bitmapformaten. Welche Formate verfügbar sind, hängt von den Fähigkeiten des installierten

ghostscript-Interpreters ab. Das folgende Beispiel zeigt das Abspeichern eines Canvas im PDF-, im EPS- und im PNG-Format. Verzichtet man in den ersten beiden Fällen auf die Angabe des Dateinamens, so wird der Name des erzeugenden Python-Skripts herangezogen, wobei die Endung py je nach Ausgabeformat durch pdf, ps oder eps ersetzt wird. Häufig ist dies ein sinnvolles Vorgehen, da man beim Kopieren von Skripten auf diese Weise vermeidet, bereits erzeugte Bilder des ursprünglichen Skripts zu überschreiben, wenn man vergisst, den Dateinamen anzupassen. Bei den Bitmapformaten wird das zu erzeugende Format aus der Endung des Dateinamens entnommen, sofern ein Name angegeben wurde. Andernfalls muss das Ausgabegerät spezifiziert werden. Unser Beispiel zeigt außerdem, wie die Auflösung der Bitmapgrafik beeinflusst werden kann.

```
In [40]: c.writePDFfile('hallo.pdf')
   ...: c.writeEPSfile('hallo.eps')
   ...: c.writeGSfile('hallo.png', resolution=300)
```

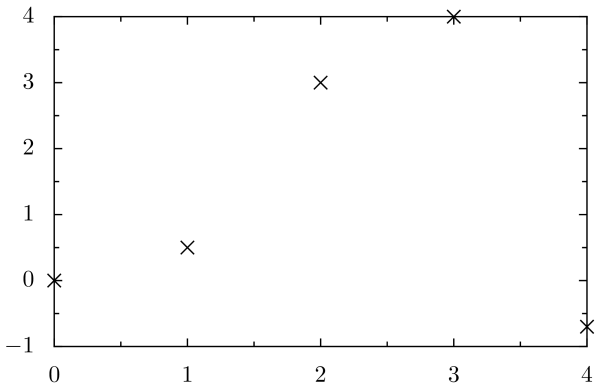
Weitere Informationen über mögliche Optionen beim Abspeichern von Grafiken kann man der Dokumentation entnehmen. Es sei hier nur erwähnt, dass man zum Beispiel beim PDF- und beim Postscript-Format die Papiergröße festlegen oder auch mehrseitige Dokumente erzeugen kann.

Bis jetzt haben wir nur Fähigkeiten von PyX besprochen, die von matplotlib nicht zur Verfügung gestellt werden. Mit PyX kann man jedoch auch genauso gut grafische Darstellungen von Daten erzeugen. Dies soll im Folgenden demonstriert werden.

Genauso wie bisher auch benötigt man für Graphen einen Canvas, allerdings mit erweiterten Fähigkeiten. Für eine gewöhnliche zweidimensionale Darstellung erzeugt man einen Canvas mit Hilfe von graph.graphxy. Wie wir noch sehen werden, lässt sich das Aussehen des Graphen durch geeignete Argumente sehr genau beeinflussen. In dem folgenden, sehr einfachen Beispiel legen wir lediglich die Breite des Graphen fest. Seine Höhe wird dann, da nichts anderes angegeben ist, mit Hilfe des goldenen Schnitts bestimmt. In unserem Beispiel wollen wir eine Reihe von Punkten darstellen, deren Lage durch eine Liste von Tupeln festgelegt wird.

Um der plot-Methode unseres Graphencanvas g die Datenquelle mitzuteilen, verwendet man in diesem Fall die graph.data.points-Methode, die als erstes Argument die Datenliste erhält. Die Argumente x und y geben die Spalte an. Die x-Werte sollen hier die ersten Werte der Tupel sein, die y-Werte die jeweils zweiten Werte. Man könnte aber auch die zweite Spalte gegen die erste Spalte darstellen oder aus längeren Tupeln die gewünschten Spalten nach Bedarf auswählen. Standardmäßig erfolgt nun eine Darstellung der Datenpunkte mit Hilfe von Kreuzen, die nicht durch eine Linie verbunden sind.

```
In [41]: g = graph.graphxy(width=8)
   ...: data = [(0, 0), (1, 0.5), (2, 3), (3, 4), (4, -0.7)]
   ...: g.plot(graph.data.points(data, x=1, y=2))
```



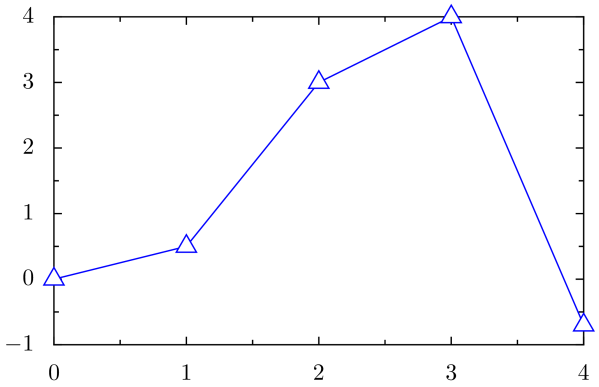
Im nächsten Beispiel wollen wir die Darstellung verbessern, indem wir die Datenpunkte durch blaue gerade Linien verbinden und als Symbole blau umrandete Dreiecke verwenden, die nicht von den Linien durchschnitten werden. Um dies zu realisieren, übergibt man der plot-Methode eine Liste von entsprechenden Attributen, so wie wir es von der stroke- oder der fill-Methode des Canvas her kennen.

Um die Übersichtlichkeit des Codes zu verbessern, haben wir im folgenden Beispiel zwei Variablen definiert, die die Eigenschaften der Linie und der Symbole enthalten. Diese Information könnte man alternativ direkt in der plot-Methode in der letzten Zeile unterbringen. Stattdessen eine Variable zu verwenden, hat neben der Übersichtlichkeit auch den Vorteil, dass sich die Vorgaben in verschiedenen plot-Aufrufen verwenden lassen und so

für eine einheitliche Darstellung sorgen sowie das Programmieren nach dem DRY-Prinzip[7] unterstützen.

Mit `graph.style.line` werden die Eigenschaften der zu zeichnenden Linien festgelegt. Diese werden im Argument als Liste angegeben, selbst wenn es sich, wie im folgenden Beispiel, nur um ein Attribut handelt, das hier die Farbe festlegt. Man könnte beispielsweise auch eine gestrichelte oder gepunktete Linie verlangen. Das Symbol wird mit Hilfe von `graph.style.symbol` festgelegt, wobei das Argument `symbol` die Form des Symbols, hier ein Dreieck, angibt. Zudem kann man mit `symbolattrs` eine Liste von Attributen übergeben, die das Aussehen des Symbols genauer festlegen. In unserem Fall wird verlangt, dass der Rand des Symbols blau ist und das Innere weiß gefüllt wird. Mit letzterem wird sichergestellt, dass die Linien die Symbole nicht schneiden.

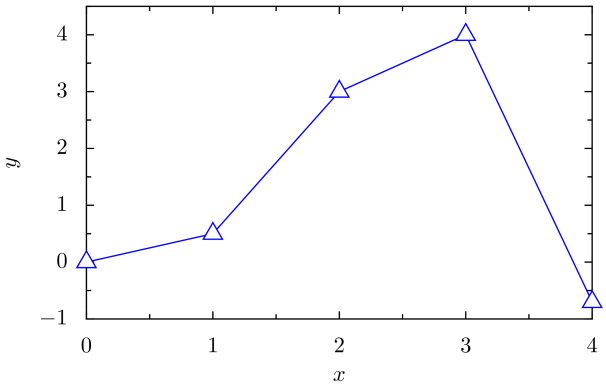
```
In [42]: g = graph.graphxy(width=8)
   ...: data = [(0, 0), (1, 0.5), (2, 3), (3, 4), (4, -0.7)]
   ...: myline = graph.style.line([color.rgb.blue])
   ...: mysymbol = graph.style.symbol(symbol=graph.style.symbol.triangle,
   ...:                               symbolattrs=[deco.filled([color.grey(1)]),
   ...:                                            deco.stroked([color.rgb.blue])])
   ...: g.plot(graph.data.points(data, x=1, y=2), [myline, mysymbol])
```



Im nächsten Beispiel nehmen wir Veränderungen an den Achsen vor. Hierzu definiert man im Graphencanvas mit Hilfe der Argumente `x` und `y` die Eigenschaften der zugehörigen Achsen. Mit `graph.axis.lin` erhält man eine linear eingeteilte Achse, deren Eigenschaften mit den zugehörigen Argumenten übergeben werden können. Dies ist zum Beispiel der Achsenumfang, der mit `min` und/oder `max` festgelegt werden kann, oder der Achsentitel, der mit `title` übergeben wird, wobei wie beim Text TeX- oder LaTeX-Syntax verwendet werden kann. In dem folgenden Beispiel wird unter anderem der Wertebereich der *y*-Achse vergrößert, damit das Symbol bei $x = 3$ im Innern des *y*-Wertebereichs liegt.

```
In [43]: g = graph.graphxy(width=8,
   ...:                    x=graph.axis.lin(title='$x$'),
   ...:                    y=graph.axis.lin(min=-1, max=4.5, title='$y$'))
   ...: data = [(0, 0), (1, 0.5), (2, 3), (3, 4), (4, -0.7)]
   ...: myline = graph.style.line([color.rgb.blue])
   ...: mysymbol = graph.style.symbol(symbol=graph.style.symbol.triangle,
   ...:                               symbolattrs=[deco.filled([color.grey(1)]),
   ...:                                            deco.stroked([color.rgb.blue])])
   ...: g.plot(graph.data.points(data, x=1, y=2), [myline, mysymbol])
```

[7] DRY steht für »don't repeat yourself«. Das DRY-Prinzip fordert dazu auf, Codewiederholungen nach Möglichkeit zu vermeiden.



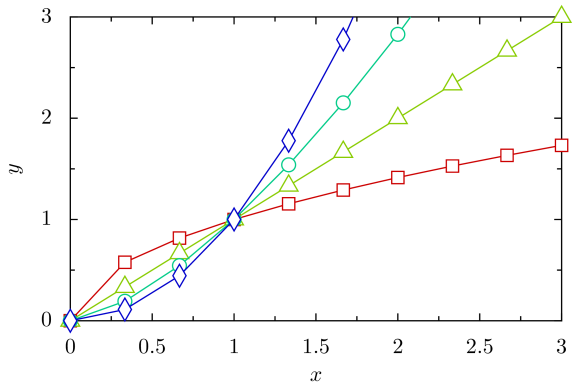
Bis jetzt haben wir als Datenquelle eine Liste von Tupeln verwendet. PyX kann alternativ auch Daten aus einer Datei einlesen oder aus vorgegebenen Funktionen berechnen. Dabei sind auch parametrisch definierte Funktionen möglich.

Wir wollen nun eine Darstellung von verschiedenen Potenzfunktionen erzeugen. Um eine Funktion darzustellen, wählt man die `graph.data.function`-Methode, in der man den funktionalen Zusammenhang zwischen x und y in einer Zeichenkette angibt. Normalerweise wählt PyX selbst die Zahl der Punkte, an denen die Funktion ausgewertet wird. Da wir die zugehörigen Punkte mit Symbolen darstellen wollen, legen wir jedoch die Zahl der Punkte mit Hilfe des Arguments `points` fest.

In unserem Beispiel werden mittels eines einzigen Aufrufs der `plot`-Methode mehrere Funktionen gezeichnet. Hierzu wird eine ganze Liste von `function`-Aufrufen als erstes Argument übergeben. Dabei ist es in PyX sehr elegant möglich festzulegen, wie die einzelnen Funktionsgraphen dargestellt werden sollen. Wie schon im vorigen Beispiel benutzen wir `graph.style.line`, um die Attribute der Linien festzulegen. Die entsprechende Liste enthält zwei Aufrufe von `attr.changelist`, die festlegen, wie sich bestimmte Attribute beim Wechsel von einer Linie zur nächsten ändern. Der erste Aufruf geht die in der vierten Zeile definierte Farbliste der Reihe nach durch. Der zweite Aufruf enthält eine Liste, die nur aus einem Element besteht und in diesem Fall dazu führt, dass alle Linien durchgezogen sind.

Bei der Symbolart verwenden wir eine vordefinierte Symbolliste, die der Reihe nach Quadrat, Dreieck, Kreis, Raute, Kreuz und Pluszeichen verwendet. In unserem Beispiel mit nur vier Funktionen kommen nur die ersten vier Symbole zum Einsatz. Damit die Farbe der Symbole zur Farbe der Linien passt, übergeben wir unsere Farbliste auch an die Liste der Symbolattribute. Zudem legen wir, wie schon im vorigen Beispiel, fest, dass die Symbole weiß zu füllen sind. Damit ergibt sich ingesamt das unter dem Code dargestellte Bild.

```
In [44]: g = graph.graphxy(width=8,
   ...:                     x=graph.axis.lin(min=0, max=3, title='$x$'),
   ...:                     y=graph.axis.lin(min=0, max=3, title='$y$'))
   ...: colors = [color.hsb(2*n/9, 1, 0.8) for n in range(0, 4)]
   ...: mylines = graph.style.line(lineattrs=[
             attr.changelist(colors), attr.changelist([style.linestyle.solid])
                                                 ])
   ...: mysymbols = graph.style.symbol(symbol=graph.style.symbol.changesquare,
   ...:                                symbolattrs=[
                 attr.changelist(colors), deco.filled([color.grey(1)])
                                                    ])
   ...: g.plot([graph.data.function('y(x)=x**{}'.format(exponent/2), points=10)
   ...:                                             for exponent in range(1, 5)],
   ...:        [mylines, mysymbols])
```

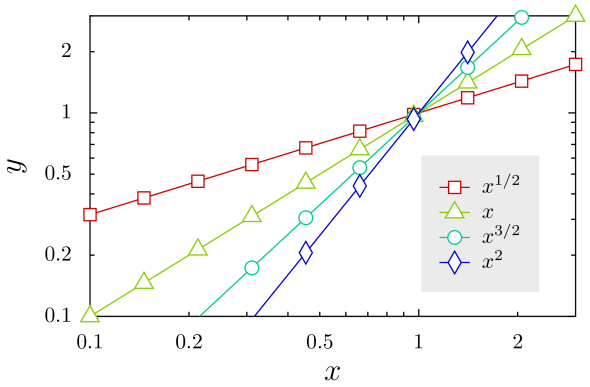


Im nächsten Schritt stellen wir die Abbildung des vorigen Beispiels auf eine doppelt-logarithmische Darstellung um und fügen zusätzlich eine Legende hinzu. Logarithmische Achsen erhält man einfach dadurch, dass man im Graphencanvas die *x*-Achse und/oder *y*-Achse mit Hilfe eines Aufrufs der `graph.axis.log`-Methode definiert. Beim Achsenumfang ist darauf achten, dass dieser eine logarithmische Darstellung erlauben muss. Negative Werte einschließlich der Null müssen also ausgeschlossen sein.

Um eine Legende zu erzeugen, definiert man das `key`-Argument des Graphencanvas entsprechend. Im Beispiel führt der Aufruf der `graph.key.key`-Methode dazu, dass die Legende unten rechts (`br` = »bottom right«) positioniert wird und die Angabe des Wertes für das Argument `dist` den Abstand zwischen den einzelnen Legendeneinträgen gegenüber dem Standardwert verringert, die Legende also etwas komprimierter darstellt wird. Schließlich verlangen wir mit Hilfe der an `keyattrs` übergebenen Liste, dass die Legende grau hinterlegt wird.

Der in der Legende dargestellte Text ist für jeden Datensatz im `title`-Argument zu übergeben. Im Beispiel haben wir hierfür eine Funktion definiert, die je nach ganz- oder halbzahligem Argument die Darstellung unter Verwendung der TeX-Syntax geeignet wählt.

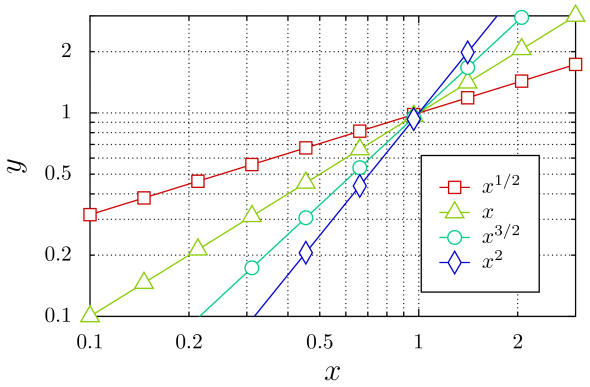
```
In [45]: def keytitle(dblexponent):
   ...:     if dblexponent == 2: return '$x$'
   ...:     if dblexponent % 2:
   ...:         return '$x^{{{}/2}}$'.format(dblexponent)
   ...:     else:
   ...:         return '$x^{}$'.format(dblexponent//2)
   ...:
   ...: g = graph.graphxy(width=8,
   ...:                   x=graph.axis.log(min=0.1, max=3, title='\Large $x$'),
   ...:                   y=graph.axis.log(min=0.1, max=3, title='\Large $y$'),
   ...:                   key=graph.key.key(pos="br", dist=0.1,
   ...:                                     keyattrs=[deco.filled([color.grey(0.9)])]))
   ...: g.plot([graph.data.function('y(x)=x**{}'.format(exponent/2),
   ...:                             points=10, title=keytitle(exponent))
   ...:                                       for exponent in range(1, 5)],
   ...:        [mylines, mysymbols])
```



Gelegentlich möchte man die Ablesung von Werten in einem Graphen durch Gitterlinien unterstützen. Dies lässt sich unabhängig voneinander für beide Achsen mit Hilfe des `painter`-Arguments festlegen. Dazu geben wir dem Painter `graph.axis.painter.regular`, der schon bisher für die Darstellung der Achsen zuständig war, ein

Attributargument mit. Wie sonst auch muss es sich dabei um eine Liste handeln, die in unserem Fall nur vorgibt, dass die Gitterlinien gepunktet sein sollen. Außerdem modifizieren wir unser Beispiel noch dahingehend, dass der Hintergrund der Legende nun weiß gehalten ist. Dafür wird der Rahmen mit einer schwarzen Linie gezeichnet. Diese Änderungen erhält man durch eine entsprechende Anpassung der Liste des `keyattrs`-Arguments.

```
In [46]: mygridattrs = [style.linestyle.dotted]
   ...: mypainter = graph.axis.painter.regular(gridattrs=mygridattrs)
   ...: g = graph.graphxy(width=8,
   ...:                   x=graph.axis.log(min=0.1, max=3, title='\Large $x$',
   ...:                                    painter=mypainter),
   ...:                   y=graph.axis.log(min=0.1, max=3, title='\Large $y$',
   ...:                                    painter=mypainter),
   ...:                   key=graph.key.key(pos="br", dist=0.1,
   ...:                                     keyattrs=[deco.filled([color.grey(1)]),
   ...:                                               deco.stroked()]))
   ...: g.plot([graph.data.function('y(x)=x**{}'.format(exponent/2),
   ...:                             points=10, title=keytitle(exponent))
   ...:                                               for exponent in range(1, 5)],
   ...:        [mylines, mysymbols])
```

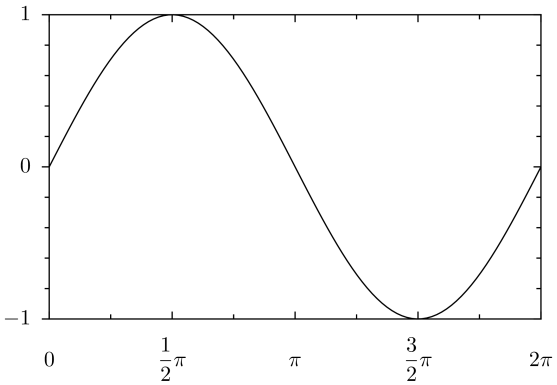


In matplotlib hatten wir gesehen, dass die Achseneinteilung recht flexibel gestaltet werden kann. Dies ist auch in PyX möglich. Wir hatten schon im Zusammenhang mit der Erzeugung von Gitterlinien gesehen, wie sich Achseneigenschaften beeinflussen lassen. Bei der y-Achse modifizieren wir wieder die Darstellung der Achse, genauer der zugehörigen Ticks, mit Hilfe der `graph.axis.painter.regular`-Methode. Um die Ticks nach außen zeigen zu lassen, setzen wir die `innerticklength` auf `None` während die `outerticklength` auf den Standardwert gesetzt wird, den bisher `innerticklength` hatte.

Ein weiterer Aspekt, den wir an der y-Achse gegenüber dem Standardverhalten verändern wollen, ist der Abstand zwischen den Ticks. Für die Einteilung der Achsen ist der »parter« zuständig. Wir geben dem bereits bisher im Verborgenen für uns tätigen `graph.axis.parter.linear` über das Argument `tickdists` eine Liste mit, die den Abstand zwischen Ticks und Subticks enthält. Durch die Wahl der Werte in unserem Beispiele erreichen wir, dass der Abstand zwischen zwei Ticks in fünf Intervalle unterteilt wird.

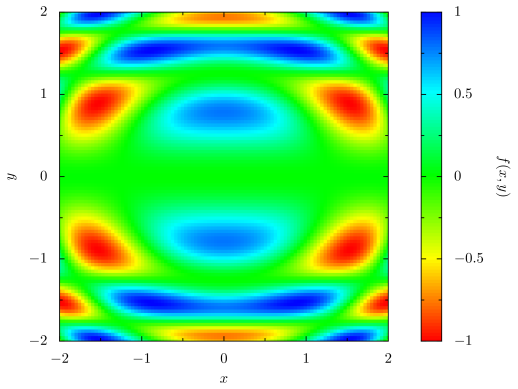
Schließlich wollen wir noch die Beschriftung der x-Achse modifizieren. Hierfür ist der »texter« zuständig. Wir verwenden hier `graph.axis.texter.rational`, der die Achsenbeschriftung mit Hilfe von Brüchen darstellt und stellen den Beschriftungen ein π hintenan. Damit die Beschriftung zum tatsächlichen Wert passt, müssen wir im Gegenzug die Werte durch π dividieren, was mit Hilfe des Arguments `divisor` eingestellt wird.

```
In [47]: mypainter = graph.axis.painter.regular(
   ...:                    innerticklength=None,
   ...:                    outerticklength=graph.axis.painter.ticklength.normal)
   ...: g = graph.graphxy(width=8,
   ...:                   x=graph.axis.linear(min=0, max=2*pi, divisor=pi,
   ...:                       texter=graph.axis.texter.rational(suffix=r"\pi")),
   ...:                   y=graph.axis.linear(
                              parter=graph.axis.parter.linear(tickdists=[1,0.2]),
   ...:                       painter=mypainter))
   ...: g.plot(graph.data.function('y(x)=sin(x)'))
```



Um eine Funktion zweier Variablen darzustellen, eignen sich neben Konturgrafiken, die wir im Zusammenhang mit matplotlib besprochen haben, auch zweidimensionale Farbdarstellungen. Der größte Teil des folgenden Beispielcodes dient dazu, eine Liste von Daten zu erzeugen, die nun aus Tupeln mit jeweils drei Einträgen besteht. Die ersten beiden Einträge geben den Ort in der Ebene an und der dritte Eintrag entspricht dem Funktionswert an dieser Stelle. Wie in unserem ersten Graphenbeispiel verwenden wir wieder `graph.data.points`. Für die Darstellung benötigen wir jedoch jetzt nicht mehr nur noch x- und y-Werte, sondern auch einen Farbwert, der sich aus dem Funktionswert ergibt. Das Argument `color` erhält daher den Wert 3, entsprechend der dritten Datenspalte. Außerdem müssen wir den Zeichenstil auf `graph.style.density` abändern. Damit wir keine Graustufendarstellung erhalten, geben wir dort noch den Farbgradienten an, der die Abbildungen von den Daten auf zugehörige Farben bewerkstelligt.

```
In [48]: import numpy as np
   ...: def f(x, y):
   ...:     return cos(x**2+y**2)*sin(2*y**2)
   ...:
   ...: xmin = -2
   ...: xmax = 2
   ...: ymin = -2
   ...: ymax = 2
   ...: npts = 100
   ...: data = [(x, y, f(x, y)) for x in np.linspace(xmin, xmax, npts)
   ...:                         for y in np.linspace(xmin, xmax, npts)]
   ...: g = graph.graphxy(height=8, width=8,
   ...:                   x=graph.axis.linear(title=r'$x$'),
   ...:                   y=graph.axis.linear(title=r'$y$'))
   ...: g.plot(graph.data.points(data, x=1, y=2, color=3, title='$f(x,y)$'),
   ...:        [graph.style.density(gradient=color.rgbgradient.Rainbow)])
```

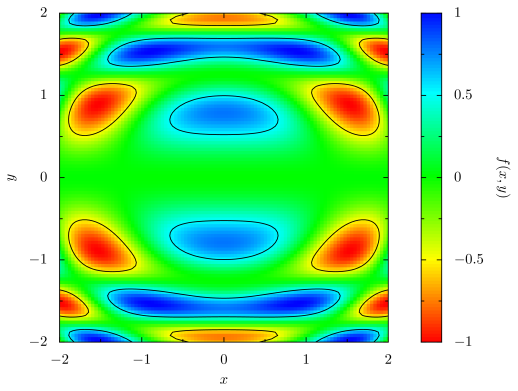


Konturlinien werden derzeit von PyX nicht unterstützt. Man kann sich allerdings bis zu einem gewissen Grad damit behelfen, dass man auf das Paket `scikit-image`[8] zurückgreift. Dieses Paket bietet eine Vielzahl interessanter Methoden zur Bildbearbeitung mit Python. Das `measure`-Paket bietet unter anderem die Möglichkeit, in einem zweidimensionalen Datensatz zu einem vorgegebenen Wert die Konturlinie oder gegebenenfalls mehrere Konturlinien zu bestimmen. Dabei ist allerdings zu beachten, dass sich die Funktion `find_contours` nicht um

[8] Scikits sind Erweiterungen von SciPy. Informationen zu `scikit-image` findet man unter scikit-image.org.

die in unserem Fall durchaus vorhandenen *x*- und *y*-Koordinaten kümmert, sondern mit Pixelpositionen arbeitet, wie es für ein gerastertes Bild adäquat ist. Daher müssen wir unsere in der Liste `data` gespeicherten Daten zunächst in ein NumPy-Array umbauen, das die richtigen Dimensionen besitzt und lediglich die Funktionswerte enthält. Anhand dieser Daten berechnet `find_contours` dann eine Liste von Konturdaten. Diese müssen wir vor der Verwendung noch in unsere Problemdaten zurückwandeln, was problemlos gelingt, da wir die Minimal- und Maximalwerte auf der *x*- und der *y*-Achse sowie die Zahl der Punkte kennen. Anschließend können wir die Konturen mit einem `plot`-Aufruf in unserem Farbbild einzeichnen lassen.

```
In [49]: from skimage.measure import find_contours
   ...:
   ...: data = [(x, y, f(x, y)) for x in np.linspace(xmin, xmax, npts)
   ...:                          for y in np.linspace(ymin, ymax, npts)]
   ...: g = graph.graphxy(height=8, width=8,
   ...:                   x=graph.axis.linear(title=r'$x$'),
   ...:                   y=graph.axis.linear(title=r'$y$'))
   ...: g.plot(graph.data.points(data, x=1, y=2, color=3, title='$f(x,y)$'),
   ...:        [graph.style.density(gradient=color.rgbgradient.Rainbow)])
   ...:
   ...: for level in (-0.5, 0.5):
   ...:     contours = find_contours(np.array([d[2] for d in data]).reshape(
                                                          npts, npts), level)
   ...:     for c in contours:
   ...:         c_rescaled = [(xmin+x*(xmax-xmin)/(npts-1),
   ...:                        ymin+y*(ymax-ymin)/(npts-1)) for x, y in c]
   ...:         g.plot(graph.data.points(c_rescaled, x=1, y=2),
   ...:                                  [graph.style.line()])
```

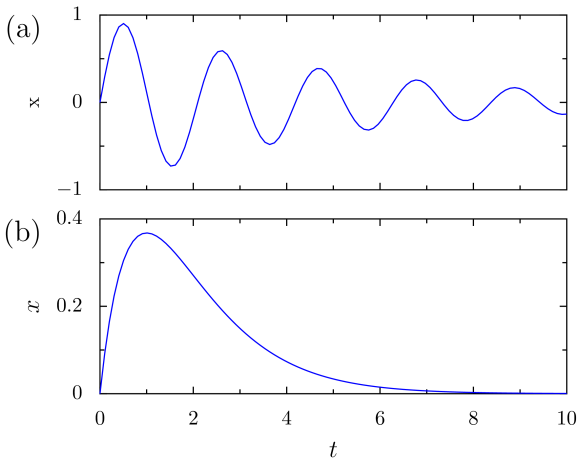


Wenn man zwei Graphen in einer Abbildung zusammenfassen will, so sollen häufig einander entsprechende Achsen gekoppelt werden, also den gleichen Achsenumfang und die gleiche Achseneinteilung besitzen. Häufig soll zudem bei einer Achse die Beschriftung entfallen. Im folgenden Beispiel verwenden wir zwei Graphencanvasse `g1` für den unteren Graphen und `g2` für den oberen Graphen, die in einen gemeinsamen Canvas `c` eingefügt werden. Dies kann, wie aus dem nachstehenden Code ersichtlich wird, bereits bei der Instantiierung der beiden Graphen geschehen. Dabei muss man allerdings darauf achten, dass die beiden Graphen relativ zueinander geeignet positioniert werden. Nachdem die untere linke Ecke des unteren Graphen im Ursprung liegt, können wir die vertikale Position `ypos` des oberen Graphen dadurch festlegen, dass wir die Höhe des unteren Graphen `g1.height` noch etwas vergrößern, hier um den Wert 0,5.

Da die beiden Graphen nun vertikal angeordnet sind, ist es sinnvoll, die *x*-Achse des oberen Graphen and die *x*-Achse des unteren Graphen zu koppeln. Dies geschieht, indem man für die *x*-Achse des oberen Graphen eine `graph.axis.linkedaxis` wählt, der man als Argument mit Hilfe von `g1.axes["x"]` die *x*-Achse des unteren Graphen übergibt.

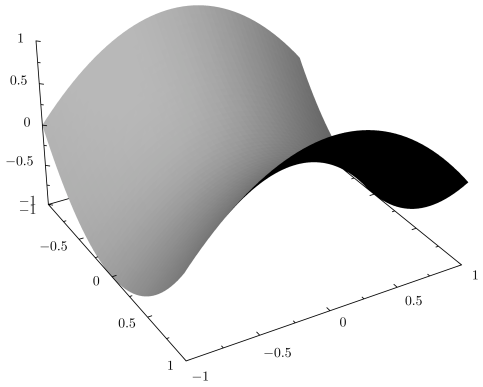
Schließlich möchte man häufig die einzelnen Graphen mit einer Bezeichnung versehen, um auf sie Bezug nehmen zu können. Zur Positionierung eignen sich die Attribute `xpos` und `ypos`, die die linke untere Ecke eines Graphen angeben, sowie `width` und `height`, die die Breite und Höhe des Graphen angeben. In diesem Zusammenhang sei noch darauf hingewiesen, dass wir in diesem Beispiel von der automatischen Bestimmung der Höhe aus der Breite mit Hilfe des goldenen Schnitts abgewichen sind, indem wir die Höhe explizit gesetzt haben.

```
In [50]: c = canvas.canvas()
   ...: g1 = c.insert(graph.graphxy(width=8, height=3,
   ...:                             x=graph.axis.lin(title='\large $t$'),
   ...:                             y=graph.axis.lin(title='\large $x$')))
   ...: g1.plot(graph.data.function("y(x)=x*exp(-x)", min=0, max=10),
   ...:         [graph.style.line(lineattrs=[color.rgb.blue])])
   ...: g1.text(g1.xpos-1, g1.height, '\Large (b)',
   ...:         [text.halign.right, text.valign.top])
   ...: g2 = c.insert(graph.graphxy(width=8, height=3,
   ...:                             ypos=g1.height+0.5,
   ...:                             x=graph.axis.linkedaxis(g1.axes["x"]),
   ...:                             y=graph.axis.lin(title='x')))
   ...: g2.plot(graph.data.function("y(x)=exp(-0.2*x)*sin(3*x)"),
   ...:                 [graph.style.line(lineattrs=[color.rgb.blue])])
   ...: g2.text(g2.xpos-1, g2.ypos+g2.height, '\Large (a)',
   ...:         [text.halign.right, text.valign.top])
```



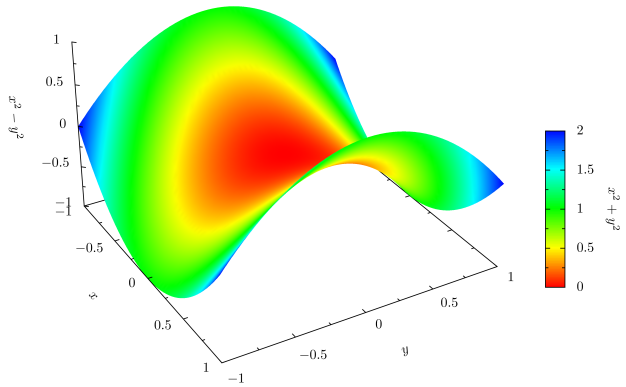
Abschließend wollen wir uns noch dreidimensionalen Darstellungen zuwenden und zu Beginn ein möglichst einfaches Beispiel betrachten. Zunächst werden Daten auf einem zweidimensionalen Gitter erzeugt und in einer Liste, die hier wieder den Namen `data` besitzt, mit Tupeln zu je drei Einträgen abgelegt. Im Unterschied zu den vorhergehenden Beispielen müssen wir den Canvas des Graphen nicht mit `graph.graphxy` sondern mit `graph.graphxyz` erzeugen, da wir ja drei Achsen x, y und z benötigen. Entsprechend müssen wir im Argument von `graph.data.points` auch die Spalte angeben, aus der die Daten für die z-Achse genommen werden sollen. In unserem Fall ist dies die Spalte 3. Schließlich muss man in der Attributliste der `plot`-Methode mit `graph.style.surface` angeben, dass die Daten in Form einer Fläche dargestellt werden sollen. So lange nichts anderes festgelegt wird, wird diese Fläche in Grautönen dargestellt.

```
In [51]: data = [(x, y, x**2-y**2)
   ...:          for x in np.linspace(-1, 1, 50) for y in np.linspace(-1, 1, 50)]
   ...: g = graph.graphxyz(size=4)
   ...: g.plot(graph.data.points(data, x=1, y=2, z=3), [graph.style.surface()])
```



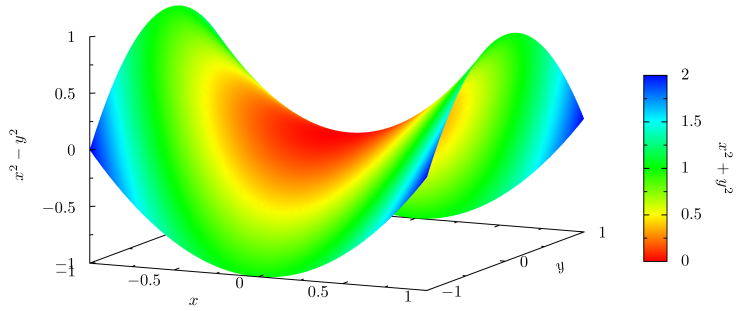
Statt einer grauen Flache kann man auch eine farbige Fläche erhalten, indem man eine geeignete Farbpalette in `graph.style.surface` mit Hilfe des Arguments `gradient` übergibt. Wird nichts Weiteres festgelegt, so wird die Farbe auf der Basis des lokalen Funktionswerts gewählt. Wir wollen jedoch in der Farbe noch eine weitere Information kodieren. Dazu erweitern wir die Tupel in unserer Liste `data` um eine weitere Spalte, die wir in `graph.data.points` der Farbachse `color` zuordnen. Außerdem haben wir Achsenbeschriftungen in der gleichen Weise, wie wir das weiter oben schon getan haben, vorgegeben. In `graph.style.surface` haben wir noch zwei Argumente auf `None` gesetzt. Mit `gridcolor` kann eine Farbe für ein Koordinatengitter vorgegeben werden, das auf die Fläche gezeichnet wird. Hier verzichten wir explizit auf ein solches Gitter, auch wenn dies bereits die Defaulteinstellung ist. Zudem verhindern wir mit der Vorgabe für `backcolor`, dass die Rückseite der Fläche schwarz dargestellt wird wie es im vorigen Beispiel der Fall war.

```
In [52]: data = [(x, y, x**2-y**2, x**2+y**2)
   ...:          for x in np.linspace(-1, 1, 50) for y in np.linspace(-1, 1, 50)]
   ...: g = graph.graphxyz(size=4,
   ...:                    x=graph.axis.lin(title='$x$'),
   ...:                    y=graph.axis.lin(title='$y$'),
   ...:                    z=graph.axis.lin(title='$x^2-y^2$'))
   ...: g.plot(graph.data.points(data, x=1, y=2, z=3, color=4, title='$x^2+y^2$'),
   ...:        [graph.style.surface(gradient=color.rgbgradient(
   ...:                                          color.gradient.Rainbow),
   ...:                             gridcolor=None,
   ...:                             backcolor=None)])
```



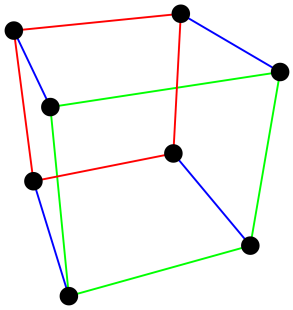
Das nächste Beispiel zeigt, dass man den Blick auf die Fläche den jeweiligen Bedürfnissen anpassen kann. Dazu gibt man im Argument `projector` des Graphencanvas den entsprechenden Projektor an. Dabei kann es sich um eine Parallelprojektion wie im folgenden Beispiel oder um eine Zentralprojektion wie im übernächsten Beispiel handeln. Bei der Parallelprojektion gibt man den Polarwinkel und den Azimutwinkel für den Normalenvektor auf der *x*-*y*-Ebene in Grad an. Bei der Zentralprojektion kommt noch eine Abstandsvariable hinzu.

```
In [53]: g = graph.graphxyz(size=4,
   ...:                    x=graph.axis.lin(title='$x$'),
   ...:                    y=graph.axis.lin(title='$y$'),
   ...:                    z=graph.axis.lin(title='$x^2-y^2$'),
   ...:                    projector=graph.graphxyz.parallel(-65, 10))
   ...: g.plot(graph.data.points(data, x=1, y=2, z=3, color=4, title='$x^2+y^2$'),
   ...:        [graph.style.surface(gradient=color.rgbgradient(
   ...:                                          color.gradient.Rainbow),
   ...:                             gridcolor=None,
   ...:                             backcolor=None)])
```



Projektoren kann man auch selbstständig verwenden, um Punkte im dreidimensionalen Raum auf eine Ebene zu projezieren. Der folgende Code berechnet für die Ecken eines Würfels die Projektion in die Ebene, und zeichnet die Achsen und Kanten.

```
In [54]: import itertools
   ...: projector = graph.graphxyz.central(10, -20, 30).point
   ...: a = 2
   ...: cube = list(itertools.product((-a, a), repeat=3))
   ...: c = canvas.canvas()
   ...: for edge in ((0, 1), (1, 3), (3, 2), (2, 0)):
   ...:     x1, y1 = projector(*cube[edge[0]])
   ...:     x2, y2 = projector(*cube[edge[1]])
   ...:     c.stroke(path.line(x1, y1, x2, y2),
   ...:              [style.linewidth.Thick, color.rgb.red])
   ...:     x1, y1 = projector(*cube[edge[0]+4])
   ...:     x2, y2 = projector(*cube[edge[1]+4])
   ...:     c.stroke(path.line(x1, y1, x2, y2),
   ...:              [style.linewidth.Thick, color.rgb.green])
   ...:     x1, y1 = projector(*cube[edge[0]])
   ...:     x2, y2 = projector(*cube[edge[0]+4])
   ...:     c.stroke(path.line(x1, y1, x2, y2),
   ...:              [style.linewidth.Thick, color.rgb.blue])
   ...: for vertex in cube:
   ...:     x, y = projector(*vertex)
   ...:     c.fill(path.circle(x, y, 0.2))
```



Dieses Beispiel zeigt noch einmal, wie vielseitig man Grafiken mit PyX erstellen kann, wobei wir hier nur die wichtigsten Aspekte ansprechen konnten. Weitere Informationen findet man in der Dokumentation von PyX auf der Webseite des Projekts.

KAPITEL 5

Versionskontrolle mit Git

5.1 Vorbemerkungen

Bei der Entwicklung von Programmen ist es sinnvoll, ein Versionskontrollsystem zu verwenden, das es erlaubt, alte Programmversionen systematisch aufzubewahren. Damit wird es beispielsweise möglich, auf definierte ältere Programmstände zurückzugehen. Es kann auch sinnvoll sein, die Versionsnummer in vom Programm erzeugten Daten abzuspeichern. Sollte sich später herausstellen, dass ein Programm fehlerhaft war, lässt sich auf diese Weise entscheiden, ob Daten von diesem Fehler betroffen sind oder nicht.

Das erste Versionskontrollsystem war das 1972 entwickelte SCCS. Später folgten Systeme wie RCS, CVS, Subversion, Git und Mercurial. Bei den aktuellen Versionskontrollsysteme lassen sich zwei Arten unterscheiden, solche die die Programmversionen zentral auf einem Server speichern und solche, bei denen die Programmversionen auf verschiedenen Rechnern verteilt vorliegen können. Die zweite Variante schließt den Fall mit ein, bei dem die Programmversionen ausschließlich lokal auf einem Rechner vorgehalten werden. Während bei einem zentralen Versionskontrollsystem eine Internetverbindung zum Server zwingend notwendig ist, lassen sich bei einem dezentralen Versionskontrollsystem Versionierungen auch ohne Internetanbindung vornehmen.

Ein Beispiel für ein modernes zentrales Versionskontrollsystem ist Subversion, während es sich bei Git und Mercurial um dezentrale Versionskontrollsysteme handelt. Obwohl Mercurial in Python geschrieben ist, wollen wir uns im Folgenden mit Git beschäftigen, das sich bei der Entwicklung freier Software großer Beliebtheit erfreut. Auch wenn es im Detail Unterschiede zwischen Mercurial und Git gibt, sind die beiden Versionskontrollsysteme einander sehr ähnlich.

Die Entwicklung von Git[1] wurde 2005 von Linus Torvalds begonnen, um ein geeignetes Versionskontrollsystem zur Entwicklung des Betriebssystemkerns von Linux zur Verfügung zu haben. Die Anforderungen ergaben sich vor allem daraus, dass Linux von einer sehr großen Zahl von Programmierern entwickelt wird, und somit die Übertragung von Code möglichst effizient, aber auch sicher vonstatten gehen muss. Im Hinblick auf den ersten Aspekt ist ein dezentrales System wesentlich besser geeignet als ein zentrales System. Detaillierte Informationen über Git findet man im Internet unter git-scm.com im Dokumentationsbereich.

5.2 Grundlegende Arbeitsschritte

Um für ein Verzeichnis sowie die darunterliegenden Verzeichnisse eine Versionierung zu ermöglichen, muss man zunächst die von Git benötigte Verzeichnisstruktur einrichten. Wir nehmen an, dass in unserem Benutzerverzeich-

[1] Zur Namensgebung sagte Linus Torvalds unter anderem „I'm an egoistical bastard, and I name all my projects after myself. First Linux, now git.“ Die Ironie dieses Satzes wird deutlich wenn man bedenkt, dass *git* im Englischen so viel wie Blödmann oder Depp bedeutet.

nis, hier `/home/gert`, ein Verzeichnis `wd` für »working directory« existiert, in dem wir unsere Programmentwicklung durchführen wollen. Dieses Verzeichnis kann im Prinzip jeden beliebigen geeigneten Namen haben. Wir gehen zunächst in dieses Verzeichnis und initialisieren es für die Benutzung mit Git:

```
$ cd ~/wd
$ git init
Initialisierte leeres Git-Repository in /home/gert/wd/.git/
```

Im Unterverzeichnis `.git` werden alle relevanten Daten des Archivs liegen. So lange dieses Verzeichnis nicht modifiziert wird, was man ohnehin nicht tun sollte, oder gar gelöscht wird, sind die dort abgelegten Daten und damit alle Versionen noch verfügbar auch wenn alle anderen Dateien im Arbeitsverzeichnis gelöscht wurden.

Zu diesem Zeitpunkt ist es sinnvoll, Git auch den vollständigen Namen des Benutzers und eine zugehörige E-Mail-Adresse mitzuteilen:

```
$ git config --global user.name "Gert-Ludwig Ingold"
$ git config --global user.email "gert.ingold@physik.uni-augsburg.de"
```

Diese Informationen legt Git im Hauptverzeichnis des Benutzers in der Datei `.gitconfig` ab und verwendet sie bei der Übernahme von Dateien in das Versionsarchiv. Lässt man die Option `--global` weg, so wird die Information im lokalen Git-Verzeichnis abgelegt und gilt auch nur dort.

Bei Bedarf lassen sich noch weitere Parameter einstellen, beispielsweise der Editor, den Git aufrufen soll, um dem Benutzer beim Abspeichern einer neuen Version die Möglichkeit zu geben, einen Kommentar abzuspeichern. Da es immer wieder vorkommt, dass defaultmäßig ein Editor verwendet wird, mit dessen Bedienung man nicht vertraut ist, empfiehlt es sich, die gewünschte Einstellung vorzunehmen, zum Beispiel

```
$ git config --global core.editor vim
```

wenn man den Editor `vim` benutzen möchte.

Um das Arbeiten mit Git zu illustrieren, legen wir anschließend eine erste Version eines Skripts `hello.py` mit folgendem Inhalt

```
print('Hello world')
```

in unserem Verzeichnis an. Nun, aber auch zu jeder anderen Zeit, kann man den Zustand des Arbeitsverzeichnisses abfragen:

```
$ git status
Auf Branch master

Initialer Commit

Unbeobachtete Dateien:
  (benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)

        hello.py

nichts zum Commit vorgemerkt, aber es gibt unbeobachtete Dateien (benutzen Sie
"git add" zum Beobachten)
```

Git gibt hier eine ganze Menge an Informationen einschließlich eines Vorschlags, was wir als Nächstes tun könnten. Doch gehen wir der Reihe nach vor. Wir befinden uns laut der ersten Zeile der Statusausgabe auf dem `master`-Zweig. Weiter unten werden wir sehen, dass wir unter Git weitere Zweige anlegen können, in denen wir beispielsweise bestimmte Aspekte eines Programms weiterentwickeln wollen. Solche Zweige können später auch wieder zusammengeführt werden. Ferner weist uns Git darauf hin, dass noch keine Dateien versioniert wurden. Dem unteren Teil der Ausgabe können wir entnehmen, dass die Versionierung in zwei Stufen vor sich geht und es demzufolge zwei verschiedene Arten von Dateien gibt.

Git hat sehr wohl bemerkt, dass es eine neue Datei `hello.py` gibt, beachtet diese jedoch zunächst nicht weiter. Es wird aber am Ende darauf hingewiesen, dass sich Dateien mit Hilfe von `git add` für einen *commit*, also

eine Versionierung, vormerken lassen. Diese Dateien werden dabei in eine so genannte *staging area* gebracht. Wir führen diesen Schritt nun aus und sehen uns den neuen Status an:

```
$ git add hello.py
$ git status
Auf Branch master

Initialer Commit

zum Commit vorgemerkte Änderungen:
  (benutzen Sie "git rm --cached <Datei>..." zum Entfernen aus der Staging-Area)

        neue Datei:     hello.py
```

Damit ist unsere Datei nun für einen *commit* vorgemerkt. Gleichzeitig gibt uns Git einen Hinweis, wie wir die Datei wieder aus der *staging area* entfernen können, falls wir doch keine Versionierung durchführen möchten. Bevor wir mit einem *commit* fortfahren, wollen wir zunächst erkunden, was es damit auf sich hat, wenn eine Datei in die *staging area* gebracht wird. Dazu sehen wir uns etwas im `.git`-Unterverzeichnis um:

```
$ ls .git
branches  config  description  HEAD  hooks  index  info  objects  refs
$ ls .git/objects
75  info  pack
$ ls .git/objects/75
d9766db981cf4e8c59be50ff01e574581d43fc
```

Im Unterverzeichnis `.git/objects/75` liegt nun eine Datei mit der etwas merkwürdigen Bezeichnung `d9766db981cf4e8c59be50ff01e574581d43fc`. Stellt man noch die `75` aus dem Verzeichnisnamen voran, so handelt es sich hierbei um den so genannten SHA1-Hashwert[2] des Objekts, wie wir folgendermaßen überprüfen können[3]:

```
from hashlib import sha1
def githash(data):
    s = sha1()
    s.update(("blob %u\0" % len(data)).encode('utf8'))
    s.update(data)
    return s.hexdigest()

content = "print('hello world')\n"
print(githash(content))
```

SHA1-Hashwerte bestehen aus 40 Hexadezimalzahlen und charakterisieren den Inhalt eines Objekts eindeutig. Immerhin gibt es etwa 10^{48} verschiedene Hashwerte. Git benutzt diesen Hashwert, um schnell Objekte identifizieren und auf Gleichheit testen zu können. Meistens genügen die ersten sechs oder sieben Hexadezimalzahlen, um ein Objekt eindeutig auszuwählen. Wir können uns den Inhalt des erzeugten Objekts mit Hilfe von Git folgendermaßen ansehen:

```
$ git cat-file -p 75d9766
print('hello world')
```

Gemäß der obigen Statusanzeige müssen wir in einem zweiten Schritt noch einen *commit* ausführen:

```
$ git commit -m "ein erstes Skript"
[master (Basis-Commit) f442b34] ein erstes Skript
 1 file changed, 1 insertion(+)
  create mode 100644 hello.py
```

Mit Hilfe des Arguments `-m` haben wir noch einen Kommentar angegeben. Ohne dieses Argument hätte Git einen Editor geöffnet, um die Eingabe eines Kommentars zu ermöglichen. Es empfiehlt sich im Hinblick auf die Übersichtlichkeit von späteren längeren Ausgaben, Kommentare auf nicht zu lange Einzeiler zu beschränken.

[2] Siehe zum Beispiel en.wikipedia.org/wiki/SHA-1.
[3] Der folgende Code basiert auf einem Vorschlag auf stackoverflow.com/questions/552659/assigning-git-sha1s-without-git.

Was hat sich durch den *commit* im Verzeichnis der Objekte getan? Wir stellen fest, dass unser altes Objekt noch vorhanden ist und zwei Objekte hinzugekommen sind:

```
$ ls -R .git/objects
.git/objects:
75  ed  f4  info  pack

.git/objects/75:
d9766db981cf4e8c59be50ff01e574581d43fc

.git/objects/ed:
868ae92a213b64de2ad627b27458537539bcdc

.git/objects/f4:
42b34f6400811648a3c94a8ddd5bfb417e1cf5

.git/objects/info:

.git/objects/pack:
```

Sehen wir uns die neuen Objekte an:

```
$ git cat-file -p f442b34
tree ed868ae92a213b64de2ad627b27458537539bcdc
author Gert-Ludwig Ingold <gert.ingold@physik.uni-augsburg.de> 1420469345 +0100
committer Gert-Ludwig Ingold <gert.ingold@physik.uni-augsburg.de> 1420469345 +0100

ein erstes Skript
$ git cat-file -p ed868ae
100644 blob 75d9766db981cf4e8c59be50ff01e574581d43fc    hello.py
```

Bei dem ersten Objekt handelt es sich um ein so genanntes *commit*-Objekt, das neben den Angaben zur Person und dem Kommentar einen Verweis auf ein *tree*-Objekt enthält. Das zweite neue Objekt ist genau dieses *tree*-Objekt. Es enthält Informationen über die Objekte, die zu dem betreffenden *commit* gehören. In unserem Fall ist dies das uns bereits bekannte *blob*-Objekt, das den Inhalt unseres Skripts `hello.py` enthält.

Nun ist es Zeit, unser Skript zu überarbeiten. Im Wort »hello« ersetzen wir das kleine `h` durch ein großes `H`. Git meldet dann den folgenden Status:

```
$ git status
Auf Branch master
Änderungen, die nicht zum Commit vorgemerkt sind:
  (benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)
  (benutzen Sie "git checkout -- <Datei>...", um die Änderungen im␣
↪Arbeitsverzeichnis
   zu verwerfen)

        geändert:       hello.py

keine Änderungen zum Commit vorgemerkt (benutzen Sie "git add" und/oder
                                        "git commit -a")
```

Git hat erkannt, dass wir unser Skript modifiziert haben, führt aber keinerlei Schritte im Hinblick auf eine Versionierung aus. Diese sind uns überlassen, wobei uns Git wieder Hilfestellung gibt. Nehmen wir an, dass wir die Änderungen wieder rückgängig machen wollen. Dies geht wie folgt:

```
$ git checkout -- hello.py
$ git status
Auf Branch master
nichts zu committen, Arbeitsverzeichnis unverändert
$ cat hello.py
print('hello world')
```

Tatsächlich liegt jetzt wieder die ursprüngliche Fassung des Skripts vor. Da wir die neue Fassung nicht zur *staging area* hinzugefügt haben, sind unsere Änderungen verloren gegangen. Sie können somit nicht wiederhergestellt werden, wie dies bei einer erfolgten Versionierung der Fall gewesen wäre. Man sollte daher mit dem beschriebenen Vorgehen besonders vorsichtig sein.

Wir wiederholen nun zur Wiederherstellung der geänderten Version die Umwandlung des `h` in einen Großbuchstaben. Anschließend könnten wir wieder die beiden Schritte `git add hello.py` und `git commit` ausführen. Alternativ lässt sich dies in unserem Fall in einem einzigen Schritt bewältigen:

```
$ git commit -a -m "fange mit Großbuchstabe an"
[master 79ff614] fange mit Großbuchstabe an
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Zu beachten ist dabei allerdings, dass auf diese Weise alle Dateien, von denen Git weiß, dem *commit* unterzogen werden auch wenn dies vielleicht nicht gewünscht ist. Es ist daher oft sinnvoll, zunächst explizit mit `git add` die Dateien für einen *commit* festzulegen. Damit lassen sich gezielt thematisch zusammenhängende Änderungen auswählen.

Während der Hashwert des ersten *commit*-Objekts mit `f442b34` begann, fängt der Hashwert des neuesten *commit*-Objekts mit `79ff614` an. Git bezieht sich auf Versionen mit Hilfe dieser Hashwerte und nicht mit zeitlich ansteigenden Versionsnummern. Letzteres ist für ein dezentral organisiertes Versionskontrollsystem nicht möglich, da im Allgemeinen nicht bekannt sein kann, ob andere Entwickler in der Zwischenzeit Änderungen am gleichen Projekt durchgeführt haben.

Einen Überblick über die verschiedenen vorhandenen Versionen kann man sich folgendermaßen verschaffen:

```
$ git log
commit 79ff6141783ca76a5424271d2cede769ff45fb28
Author: Gert-Ludwig Ingold <gert.ingold@physik.uni-augsburg.de>
Date:   Mon Jan 5 16:30:22 2015 +0100

    fange mit Großbuchstabe an

commit f442b34f6400811648a3c94a8ddd5bfb417e1cf5
Author: Gert-Ludwig Ingold <gert.ingold@physik.uni-augsburg.de>
Date:   Mon Jan 5 15:49:05 2015 +0100

    ein erstes Skript
```

Die Ausgabe kann mit Optionen sehr detailliert beeinflusst werden. Wir geben hier nur ein Beispiel:

```
$ git log --pretty=oneline
79ff6141783ca76a5424271d2cede769ff45fb28 fange mit Großbuchstabe an
f442b34f6400811648a3c94a8ddd5bfb417e1cf5 ein erstes Skript
```

Diese einzeilige Ausgabe funktioniert dann besonders gut, wenn man sich wie weiter oben bereits empfohlen bei der Beschreibung der Version auf eine einzige, möglichst informative Zeile beschränkt. Informationen über weitere Optionen von Git-Befehlen erhält man grundsätzlich mit `git help` und der anschließenden Angabe des gewünschten Befehls, in unserem Falle also `git help log`.

Details zu einer Version, im Folgenden die Version `79ff614`, erhält man folgendermaßen:

```
$ git show 79ff614
commit 79ff6141783ca76a5424271d2cede769ff45fb28
Author: Gert-Ludwig Ingold <gert.ingold@physik.uni-augsburg.de>
Date:   Mon Jan 5 16:30:22 2015 +0100

    fange mit Großbuchstabe an

diff --git a/hello.py b/hello.py
index 75d9766..f7d1785 100644
--- a/hello.py
+++ b/hello.py
```

```
@@ -1 +1 @@
-print('hello world')
+print('Hello world')
```

Dieser Ausgabe kann man entnehmen, dass das Objekt `75d9766...` in das Objekt `f7d1785...` umgewandelt wurde. Aus den letzten Zeilen kann man die Details der Änderung ersehen.

Wir hatten weiter oben darauf hingewiesen, dass man im Detail beeinflussen kann, welche Dateien beim nächsten *commit* berücksichtigt werden. Dazu werden die betreffenden Dateien mit einem `git add` in die *staging area* aufgenommen. In diesem Zusammenhang kann es passieren, dass man eine Datei versehentlich zu diesem Index hinzufügt. Im folgenden Beispiel sei dies eine Datei namens `spam.py`:

```
$ git add spam.py
$ git status
Auf Branch master
zum Commit vorgemerkte Änderungen:
  (benutzen Sie "git reset HEAD <Datei>..." zum Entfernen aus der Staging-Area)

        neue Datei:     spam.py
```

Diese Datei lässt sich nun wie angegeben wieder aus der *staging area* entfernen:

```
$ git reset HEAD spam.py
$ git status
Auf Branch master
Unbeobachtete Dateien:
  (benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)

        spam.py

nichts zum Commit vorgemerkt, aber es gibt unbeobachtete Dateien (benutzen Sie
"git add" zum Beobachten)
```

Im Arbeitsverzeichnis ist die Datei `spam.py` weiterhin vorhanden. Im `reset`-Befehl verweist `HEAD` hier auf die Arbeitsversion im aktuellen Zweig, deren Hashwert wir somit nicht explizit kennen müssen.

5.3 Verzweigen und Zusammenführen

Bei der Entwicklung von Software ist es häufig sinnvoll, gewisse Weiterentwicklungen vom Hauptentwicklungsstrang zumindest zeitweise abzukoppeln. Dies erreicht man durch Verzweigungen. Ein typischer Fall wäre ein öffentliches Release, das im Hauptzweig zum nächsten Release weiterentwickelt wird. Daneben kann es aber noch einen Zweig geben, in dem ausschließlich Fehler des Releases korrigiert und dann wieder veröffentlicht werden. In einem anderen Szenario behinhaltet der Hauptzweig, der in Git unter dem Namen *master* läuft, immer eine lauffähige Version, während zur Entwicklung gewisser Programmaspekte separate Zweige benutzt werden. Um ein auf diese Weise entwickeltes Feature in die Version des Hauptzweiges einfließen zu lassen, muss man Zweige auch wieder zusammenführen können. Das Verzweigen und Zusammenführen geht in Git sehr einfach, da lediglich Markierungen gesetzt werden. Daher gehört das Verzweigen und Zusammenführen bei der Arbeit mit Git zu den Standardverfahren, die regelmäßig zum Einsatz kommen.

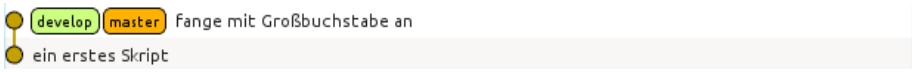
Zu Beginn gibt es nur einen Zweig, der, wie wir bereits wissen, den Namen `master` besitzt. Im vorigen Kapitel haben wir in diesem Zweig zwei Versionen erzeugt. Eine graphische Darstellung, die hier mit dem Git-Archive-Betrachter `gitg` erzeugt wurde, sieht dann folgendermaßen aus:

Die Information über die vorhandenen Zweige lässt sich auch direkt auf der Kommandzeile erhalten. In der folgenden Ausgabe ist zu erkennen, dass es nur einen Zweig, nämlich `master` gibt. Der Stern zeigt zudem an, dass wir uns gerade in diesem Zweig befinden.

```
$ git branch
* master
```

Die Situation wird interessanter, wenn wir einen weiteren Zweig anlegen, der von `master` abzweigt. Wir nennen ihn `develop`, da dort die Programmentwicklung erfolgen soll, während in `master` immer eine lauffähige Version enthalten sein soll. Damit ist es unproblematisch, wenn das Programm im `develop`-Zweig zeitweise nicht funktionsfähig ist.

```
$ git branch develop
$ git branch
  develop
* master
```



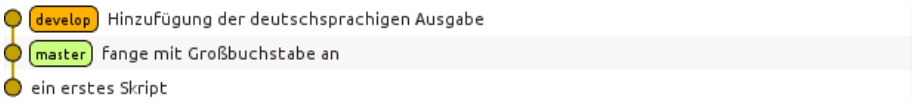
Der neue Zweig `develop` tritt zunächst nur als weitere Bezeichnung neben `master` in Erscheinung. Die Verzweigung wird erst später deutlich werden, wenn wir Dateien in den jeweiligen Zweigen verändern.

Um nun in `develop` arbeiten zu können, müssen wir in diesen Zweig wechseln:

```
$ git checkout develop
Zu Branch 'develop' gewechselt
$ git branch
* develop
  master
```

Der Stern zeigt an, dass der Zweigwechsel tatsächlich vollzogen wurde.

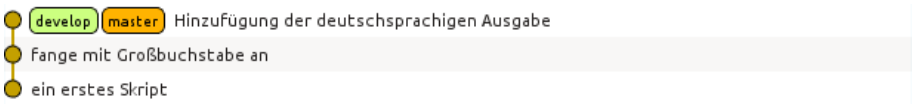
Bearbeitet man nun eine Datei im `develop`-Zweig und führt ein *commit* durch, so wird die Trennung der beiden Zweige deutlich.



Wir wechseln nun in den `master`-Zweig zurück und führen ein *merge*, also eine Vereinigung von zwei Zweigen durch. Git sucht in diesem Fall nach dem gemeinsamen Vorfahren der beiden Zweige und baut die im `develop`-Zweig durchgeführten Änderungen auch im `master`-Zweig ein:

```
$ git checkout master
Zu Branch 'master' gewechselt
$ git merge develop
Aktualisiere 79ff614..79f695b
Fast-forward
 hello.py | 1 +
  1 file changed, 1 insertion(+)
```

Da im `master`-Zweig in der Zwischenzeit keine Änderungen vorgenommen wurden, linearisiert Git die Vorgeschichte. Es sind aber nach wie vor beide Zweige vorhanden.



Möchte man festhalten, dass die Entwicklung im `develop`-Zweig durchgeführt wurde, so kann man dieses so genannte *fast forward* mit der Option `--no-ff` beim Zusammenführen der beiden Zweige verhindern. Um dies zu zeigen, wechseln wir zunächst in den `develop`-Zweig.

```
$ git checkout develop
Zu Branch 'develop' gewechselt
```

Dort führen wir die gewünschten Änderungen und einen anschließenden *commit* durch. Nach dem Wechsel in den `master`-Zweig benutzen wir nun beim Zusammenführen die Option `--no-ff`.

```
$ git commit -a -m 'dreifache Ausgabe'
[develop d2bfce0] dreifache Ausgabe
 1 file changed, 3 insertions(+), 2 deletions(-)
$ git checkout master
Zu Branch 'master' gewechselt
$ git merge --no-ff develop
Merge made by the 'recursive' strategy.
 hello.py | 5 +++--
 1 file changed, 3 insertions(+), 2 deletions(-)
```

Die folgende Abbildung zeigt, dass die Versionsgeschichte jetzt den Zweig darstellt, in dem die Änderung tatsächlich erfolgte.

Genauso wie man Änderungen aus dem `develop`-Zweig in den `master`-Zweig übernehmen kann, kann man auch Änderungen vom `master`-Zweig in den `develop`-Zweig übernehmen. Eine typische Situation besteht darin, dass im `master`-Zweig ein Fehler korrigiert wird, der auch in der Entwicklungsversion vorliegt. Zunächst nehmen wir an, dass im `develop`-Zweig weiter gearbeitet wird. Im `master`-Zweig wird der Fehler korrigiert, so dass jetzt in beiden Zweigen Änderungen vorliegen.

Um Änderungen aus dem `master`-Zweig in den `develop`-Zweig zu übernehmen, wechseln wir in Letzteren und führen dort ein *merge* des `master`-Zweigs durch:

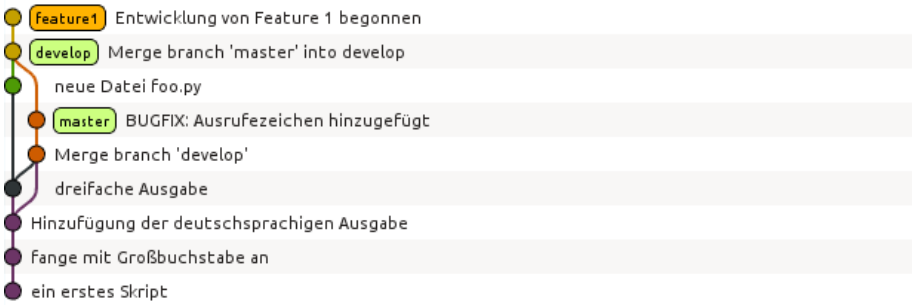
```
$ git checkout develop
Zu Branch 'develop' gewechselt
$ git merge master
Merge made by the 'recursive' strategy.
 hello.py | 4 ++--
 1 file changed, 2 insertions(+), 2 deletions(-)
```

Damit sieht unser Verzweigungsschema folgendermaßen aus:

Um ein neues Feature für ein Programm zu entwickeln, wird häufig ein Zweig vom `develop`-Zweig abgespalten und nach der Entwicklung mit Letzterem wieder zusammengeführt. Sollte die Entwicklung nicht erfolgreich gewesen sein, so verzichtet man auf die Zusammenführung oder löscht den überflüssig gewordenen Zweig. Bei

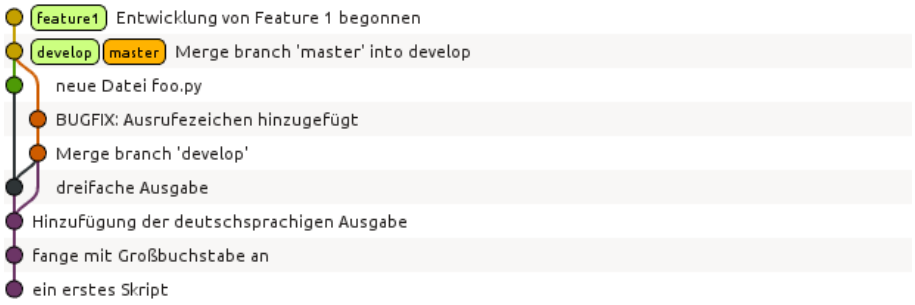
dieser Gelegenheit zeigen wir, wie man das Anlegen eines neuen Zweigs und das Wechseln in diesen Zweig mit einem Kommando erledigen kann:

```
$ git checkout -b feature1
Gewechselt zu einem neuen Branch 'feature1'
```



Unabhängig von der Entwicklung im `feature1`-Zweig kann man nun Änderungen zwischen dem `master`- und dem `develop`-Zweig austauschen:

```
$ git branch
  develop
* feature1
  master
$ git checkout master
Zu Branch 'master' gewechselt
$ git merge develop
Aktualisiere 70f9136..5b5d1e9
Fast-forward
 foo.py | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 foo.py
```



Bis jetzt gingen alle Zusammenführungen problemlos von statten. Es kann aber durchaus zu Konflikten kommen, die sich für Git nicht eindeutig auflösen lassen. Dann muss der Konflikt von Hand gelöst werden. Um dies zu illustrieren, führen wir im `develop`-Zweig eine Änderung ein, die beim Zusammenführen mit dem `feature1`-Zweig zu einem Konflikt führt.

Die folgende Ausgabe zeigt, wie Git einen Konflikt anzeigt. In der konfliktbehafteten Datei `hello.py` sind die kritischen Stellen gegenübergestellt. Zunächst wird die problematische Codestelle in der Arbeitsversion des

develop-Zweigs angezeigt. Getrennt von ======= folgt dann der Code aus dem feature1-Zweig, der im develop-Zweig aufgenommen werden soll.

```
$ git branch
* develop
  feature1
  master
$ git merge feature1
automatischer Merge von hello.py
KONFLIKT (Inhalt): Merge-Konflikt in hello.py
Automatischer Merge fehlgeschlagen; beheben Sie die Konflikte und committen Sie
dann das Ergebnis.
$ cat hello.py
<<<<<<< HEAD
for n in range(4):
    print('Hello world!')
    print('Hallo Welt!')
=======
def myfunc1(n):
    for _ in range(3):
        print('Hello world!')
        print('Hallo Welt!')
>>>>>>> feature1
```

In einer solchen Situation muss der Benutzer entscheiden, welche Version die gewünschte ist. Unter Umständen kann es erwünscht, Teile jeweils aus dem einen oder dem anderen Zweig zu entnehmen. Hat man eine zufriedenstellende Version erstellt, kann man einen *commit* durchführen.

```
$ git add hello.py
$ git commit -m'Konflikt behoben'
[develop ef71e70] Konflikt behoben
```

Um abschließend die drei Zweige zu zeigen, die in der Diskussion eine Rolle gespielt haben, führen wir noch je eine Änderung im master- und im feature1-Zweig durch und erhalten damit das folgende Bild:

Der Umstand, dass wir bereits in wenigen Schritten ein relativ komplexes Verzweigungsdiagramm erhalten haben, legt es inbesondere für größere Projekte nahe, sich eine Strategie für das Anlegen von Zweigen und die darin auszuführenden Aufgaben zu überlegen. Bei Projekten mit mehreren Entwicklern ist andererseits gerade die Möglichkeit, Zweige einzurichten, nützlich, um die anderen Entwickler nicht unnötig mit Code zu belasten, der nur lokal für einen Entwickler von Bedeutung ist.

5.4 Umgang mit entfernten Archiven

Bis jetzt haben wir uns nur mit der Arbeit mit einem lokalen Archiv beschäftigt. Wenn mehrere Entwickler zusammenarbeiten, muss es jedoch die Möglichkeit des Austauschs von Code geben. Unter einem zentralen Versionskontrollsystem wie Subversion dient hierzu das Archiv auf dem zentralen Server, über den ohnehin die gesamte Versionskontrolle läuft. Auch unter Git ist es sinnvoll, ein zentrales Archiv zu haben, das jedoch vor allem für den Datenaustausch und nicht so sehr für die Versionskontrolle herangezogen wird. Somit benötigt man nur für den Datenaustausch mit dem zentralen Archiv eine funktionierende Internetanbindung, während die Versionskontrolle auch ohne sie möglich ist.

Je nachdem welches Protokoll für den Datenzugriff zugelassen ist und welche Zugriffsrechte man besitzt, kann man auf das zentrale Archiv lesend oder eventuell auch schreibend zugreifen. In den folgenden Beispielen wollen wir einen Zugriff per `ssh`, also *secure shell*, annehmen, der uns, nach entsprechender Authentifizierung, sowohl Lese- als auch Schreibzugriff ermöglicht. Das zentrale Archiv soll auf dem Rechner `nonexistent` liegen, der, wie der Name schon andeutet, in Wirklichkeit nicht existiert. Der Name ist also entsprechend anzupassen. Der Zugriff erfolge über einen Benutzer namens `user`. Auch der Benutzername muss an die tatsächlichen Gegebenheiten angepasst werden.

Als erstes erzeugen wir uns lokal ein Git-Arbeitsverzeichnis, indem wir das zentrale Archiv klonen. Zur Illustration haben wir dort zunächst wieder nur eine Version eines einfachen Skripts abgelegt.

```
$ git clone ssh://user@nonexistent.physik.uni-augsburg.de/home/user/dummy.git dummy
Klone nach 'dummy'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Empfange Objekte: 100% (3/3), Fertig.
Prüfe Konnektivität... Fertig
$ cd dummy
$ git branch -va
* master                96ffbf6 hello world Skript
  remotes/origin/HEAD   -> origin/master
  remotes/origin/master 96ffbf6 hello world Skript
$ cat hello.py
print('hello world')
```

Nach dem Wechsel in das Arbeitsverzeichnis sehen wir, dass neben dem gewohnten `master`-Zweig noch zwei `remote`-Zweige existieren. Hierbei handelt es sich um Zweige, die auf das zentrale Archiv verweisen, das standardmäßig mit `origin` bezeichnet wird. Um die `remote`-Zweige angezeigt zu bekommen, muss die Option `-a` angegeben werden. Andernfalls beschränkt sich die Ausgabe auf die lokal vorhandenen Zweige. Informationen über entfernte Archive und den zugehörigen Zugriffsweg erhält man mit:

```
$ git remote -v
origin  ssh://user@nonexistent.physik.uni-augsburg.de/home/user/dummy.git (fetch)
origin  ssh://user@nonexistent.physik.uni-augsburg.de/home/user/dummy.git (push)
```

Nehmen wir an, dass auf dem zentralen Server eine Datei verändert wurde, so können wir diese von dort in unser Arbeitsverzeichnis holen:

```
$ git fetch origin
remote: Counting objects: 5, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Entpacke Objekte: 100% (3/3), Fertig.
Von ssh://user@nonexistent.physik.uni-augsburg.de/home/user/dummy.git
   96ffbf6..26f3c10  master     -> origin/master
```

Dabei wird nur der `origin`-Zweig aktualisiert, wie am Ausrufezeichen, das in der aktuellen Version von `hello.py` hinzugefügt wurde, zu sehen ist:

```
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
```

```
  remotes/origin/master
$ cat hello.py
print('hello world')
$ git checkout origin
Note: checking out 'origin'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD ist jetzt bei 26f3c10... mit Ausrufezeichen
$ cat hello.py
print('hello world!')
```

Wie uns Git informiert, können wir im `origin`-Zweig keine Änderungen vornehmen. Wir können uns dort aber umsehen und uns auf diese Weise davon überzeugen, dass das Skript dort das Aufrufezeichen enthält. Die Änderung können wir wie im vorigen Kapitel beschrieben in den `master`-Zweig unseres lokalen Archivs übernehmen:

```
$ git checkout master
Vorherige Position von HEAD war 26f3c10... mit Ausrufezeichen
Zu Branch 'master' gewechselt
Ihr Branch ist zu 'origin/master' um 1 Commit hinterher, und kann vorgespult␣
↪werden.
  (benutzen Sie "git pull", um Ihren lokalen Branch zu aktualisieren)
$ git merge origin
Aktualisiere 96ffbf6..26f3c10
Fast-forward
 hello.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
$ cat hello.py
print('hello world!')
```

Die Aktualisierung auf den Stand des zentralen Archivs haben wir hier in zwei Schritten durchgeführt. Es ist jedoch auch möglich, dies in einem Schritt zu erledigen. Wir nehmen an, dass ein anderer Entwickler das Skript mit einem weiteren Ausrufezeichen versehen hat, und führen dann eine so genannte *pull*-Operation aus:

```
$ git pull origin
remote: Counting objects: 5, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Entpacke Objekte: 100% (3/3), Fertig.
Von ssh://user@nonexistent.physik.uni-augsburg.de/home/user/dummy.git
   26f3c10..10f6489  master     -> origin/master
Aktualisiere 26f3c10..10f6489
Fast-forward
 hello.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
$ cat hello.py
print("hello world!!")
```

Schreibzugriff vorausgesetzt kann man umgekehrt auch neue Dateiversionen im zentralen Archiv ablegen. Hierzu dient die *push*-Operation. Hierzu ändern wir den Ausgabetext unseres Skripts und legen das neue Skript in unser lokales Archiv. Anschließend kann die Übertragung in das zentrale Archiv erfolgen:

```
$ cat hello.py
print "Hallo Welt!!"
$ git commit hello.py -m"deutsche Variante"
[master d2b98d1] deutsche Variante
```

```
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push origin
Zähle Objekte: 3, Fertig.
Schreibe Objekte: 100% (3/3), 290 bytes | 0 bytes/s, Fertig.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://user@nonexistent.physik.uni-augsburg.de/home/user/dummy.git
   10f6489..d2b98d1  master -> master
```

Problematisch wird die Situation, wenn zwischen einer *pull*-Operation und einer *push*-Operation ein anderer Entwickler das zentrale Archiv verändert hat:

```
$ git commit hello.py -m"Ausgabe deutsch und englisch"
[master 8e5577d] Ausgabe deutsch und englisch
 1 file changed, 1 insertion(+)
$ git push origin
To ssh://user@nonexistent.physik.uni-augsburg.de/home/user/dummy
 ! [rejected]        master -> master (fetch first)
error: Fehler beim Versenden einiger Referenzen nach
               'ssh://user@nonexistent.physik.uni-augsburg.de/home/user/dummy.git'
Hinweis: Aktualisierungen wurden zurückgewiesen, weil das Remote-Repository Commits
Hinweis: enthält, die lokal nicht vorhanden sind. Das wird üblicherweise durch␣
↪einen
Hinweis: "push" von Commits auf dieselbe Referenz von einem anderen Repository aus
Hinweis: verursacht. Vielleicht müssen Sie die externen Änderungen zusammenzuführen
Hinweis: (z.B. 'git pull ...') bevor Sie erneut "push" ausführen.
Hinweis: Siehe auch die Sektion 'Note about fast-forwards' in 'git push --help'
Hinweis: für weitere Details.
```

Wir folgen dem Hinweis und holen uns zunächst die veränderte Version:

```
$ git pull origin
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Entpacke Objekte: 100% (6/6), Fertig.
Von ssh://user@nonexistent.physik.uni-augsburg.de/home/user/dummy.git
   d2b98d1..a2e308b  master     -> origin/master
automatischer Merge von hello.py
KONFLIKT (Inhalt): Merge-Konflikt in hello.py
Automatischer Merge fehlgeschlagen; beheben Sie die Konflikte und committen Sie␣
↪dann
das Ergebnis.
```

Dabei kommt es zu einem Konflikt, da das gleiche Skript in verschiedener Weise verändert wurde. Zunächst muss nun dieser Konflikt beseitigt werden, damit anschließend die gewünschte Fassung der Datei versioniert werden kann:

```
$ git add hello.py
$ git commit
[master c189281] Merge branch 'master' of
   ssh://user@nonexistent.physik.uni-augsburg.de/home/user/dummy.git
```

Anschließend kann diese Version erfolgreich im zentralen Archiv abgelegt werden:

```
$ git push origin
Zähle Objekte: 4, Fertig.
Delta compression using up to 4 threads.
Komprimiere Objekte: 100% (2/2), Fertig.
Schreibe Objekte: 100% (4/4), 515 bytes | 0 bytes/s, Fertig.
Total 4 (delta 1), reused 0 (delta 0)
To ssh://user@nonexistent.physik.uni-augsburg.de/home/user/dummy.git
   a2e308b..c189281  master -> master
```

An diesem Beispiel wird deutlich, dass es durchaus problematisch sein kann, wenn viele Entwickler Schreibzugriff auf ein zentrales Archiv haben. Daher wird häufig einem breiteren Personenkreis lediglich Lesezugriff gewährt. Nur ein einzelner Entwickler oder eine kleine Gruppe hat Schreibzugriff auf das zentrale Archiv und kann so neuen Code dort ablegen. Dies geschieht mit einer *pull*-Operation von einem Archiv des Entwicklers, der den Code zur Verfügung stellt. Hierzu ist wiederum eine Leseberechtigung nötig. Möchte ein Entwickler bei diesem Verfahren Code für das zentrale Archive zur Verfügung stellen, so stellt er eine *pull*-Anfrage (*pull request*). Eventuell nach einer Diskussion und Prüfung entscheidet der Verantwortliche für das zentrale Archiv über die Aufnahme in das zentrale Archiv und führt die *pull*-Operation durch (oder auch nicht). Eine häufig verwendete Infrastruktur, die in dieser Weise insbesondere auch für die Entwicklung freier Software benutzt wird, ist GitHub.

KAPITEL 6

Testen von Programmen

6.1 Wozu braucht man Tests?

Ein offensichtliches Ziel beim Programmieren besteht darin, letztlich ein funktionierendes Programm zu haben. Funktionierend heißt hierbei, dass das Programm die gewünschte Funktionalität korrekt bereitstellt. Im Bereich des numerischen Rechnens heißt dies insbesondere, dass die erhaltenen Ergebnisse korrekt sind. Versucht man, mit numerischen Methoden noch ungelöste naturwissenschaftliche Fragestellungen zu bearbeiten, so lässt sich normalerweise die Korrektheit nicht direkt überprüfen. Andernfalls wäre das gesuchte Ergebnis ja bereits bekannt. Immerhin hat man häufig die Möglichkeit, das Ergebnis auf seine Plausibilität hin zu überprüfen, aber auch hier sind Grenzen gesetzt. Es kann ja durchaus vorkommen, dass eine Problemstellung zu einem völlig unerwarteten Ergebnis führt, dessen Hintergründe nicht ohne Weiteres verständlich sind.

Um die Korrektheit der Ergebnisse möglichst weitgehend abzusichern, sollte man daher alle sich bietenden Testmöglichkeiten wahrnehmen. Nicht selten geschieht dies in der Praxis in einer sehr informellen Weise. Tests werden zwar durchgeführt, aber nicht dokumentiert und auch nicht wiederholt, nachdem der Code geändert wurde. Als Abhilfe ist es sinnvoll, einen Testrahmen aufzubauen, der es zum einen erlaubt, Tests zu definieren und damit zu dokumentieren, und zum anderen diese Tests in einfacher Weise auszuführen.

Beim Formulieren von Tests sollte man sich Gedanken darüber machen, was alles schief gehen könnte, um möglichst viele Problemfälle detektieren zu können. In diesem Prozess können sich schon Hinweise auf Möglichkeiten zur Verbesserung eines Programms ergeben. Im Rahmen des so genannten *test-driven developments* geht man sogar so weit, zunächst die Tests zu formulieren und dann das zugehörige Programm zu schreiben. Allerdings sind gerade im naturwissenschaftlichen Bereich die Anforderungen zu Beginn nicht immer so klar zu definieren, dass dieses Verfahren regelmäßig zur Anwendung kommen kann.

Tests können aber sehr wohl auch während des Entwicklungsprozesses geschrieben werden. Entdeckt man einen Fehler, der nicht von einem der Tests angezeigt wurde, so sollte man es sich zur Regel machen, einen Test zu schreiben, der diesen Fehler feststellen kann. Auf diese Weise kann man verhindern, dass sich dieser Fehler nochmals unbemerkt in das Programm einschleicht.

Um von dem Fehlschlagen eines Tests möglichst direkt auf die Fehlerursache schließen zu können, empfiehlt es sich, den Code in überschaubare Funktionen mit einer klaren Aufgabe zu zerlegen, die jeweils für sich getestet werden können. Das Schreiben von Tests kann dabei nicht nur die Korrektheit des Codes überprüfen helfen, sondern auch dazu beitragen, die logische Gliederung des Codes zu verbessern. Das Testen einzelner Codeeinheiten nennt man *Unit testing*, auf das wir uns in diesem Kapitel konzentrieren werden. Zusätzlich wird man aber auch das Zusammenwirken der einzelnen Teile eines Programms im Rahmen von Integrationstests überprüfen.

Beim Schreiben von Tests sollte man darauf achten, dass die einzelnen Test möglichst unabhängig voneinander sind, also jeweils spezifische Aspekte des Codes überprüfen. Dabei lohnt es sich, auf Randfälle zu achten, al-

so Situationen, die nicht dem allgemeinen Fall entsprechen und denen beim Programmieren eventuell nicht die notwendige Aufmerksamkeit zu Teil wird. Als Beispiel könnte man die Auswertung einer Funktion mit Hilfe einer Rekursionsformel nennen. Dabei wäre auch auf Argumente zu achten, bei denen die Rekursionsformel nicht verwendet wird, sondern direkt deren Anfangswert zurückzugeben ist.

Außerdem sollte man es sich zum Ziel setzen, den Code möglichst vollständig durch Tests abzudecken.[1] Werden Teile des Codes durch keinen Test ausgeführt, so könnten sich dort Fehler verstecken. Andererseits ist es nicht nötig, Bibliotheken, die bereits von Haus aus eigene umfangreiche Testsuites besitzen, zu testen. Man wird also zum Beispiel darauf verzichten, Funktionen der Python Standard Library zu testen.

Aus den verschiedenen Möglichkeiten, in Python einen Testrahmen aufzubauen, wollen wir zwei herausgreifen. Die erste basiert auf dem `doctest`-Modul, das es erlaubt, einfache Tests in den Dokumentationsstrings unterzubringen. Diese Tests erfüllen somit neben ihrer eigentlichen Aufgabe auch noch die Funktion, die Möglichkeiten der Verwendung beispielsweise einer Funktion oder einer Klasse zu dokumentieren. Die zweite Möglichkeit, die wir besprechen wollen, basiert auf dem `unittest`-Modul, das auch komplexere Testszenarien ermöglicht.

6.2 Das `doctest`-Modul

In Python ist die Dokumentation von Code nicht nur mit Kommentaren, die mit # eingeleitet werden, möglich, sondern auch mit Hilfe von Dokumentationsstrings. So können zum Beispiel Funktionen dokumentiert werden, indem nach der Kopfzeile ein typischerweise mehrzeiliger Dokumentationstext eingefügt wird. Dieses Vorgehen wird in Python unter anderem dadurch belohnt, dass dieser Text mit Hilfe der eingebauten `help`-Methode verfügbar gemacht wird. Ein weiterer Bonus besteht darin, dass im Dokumentationsstring Tests untergebracht werden können, die zugleich die Verwendung des dokumentierten Objekts illustrieren.

Während der Dokumentationsaspekt alleine durch die Anwesenheit des entsprechenden Textteils im Dokumentationsstring erfüllt wird, benötigen wir für den Test das `doctest`-Modul. Wir beginnen mit dem folgenden einfachen Beispiel.

```
def welcome(name):
    """
    be nice and greet somebody
    name: name of the person

    """
    return 'Hallo {}!'.format(name)
```

Mit `help(welcome)` wird dann bekanntermaßen der Dokumentationsstring ausgegeben, also

```
In [1]: help(welcome)

Help on function welcome in module __main__:

welcome(name)
    be nice and greet somebody
    name: name of the person
```

Wir erweitern nun den Dokumentationsstring um ein Anwendungsbeispiel, das einerseits dem Benutzer die Verwendung der Funktion illustriert und andererseits zu Testzwecken dienen kann.

```
def welcome(name):
    """
    be nice and greet somebody
    name: name of the person

    >>> welcome('Guido')
    'Hallo Guido!'

```

[1] Zur Überprüfung der Codeabdeckung durch Tests kann `coverage.py` dienen, dessen Dokumentation unter http://coverage.readthedocs.io zu finden ist.

```
9       """
10      return 'Hallo {}!'.format(name)
11
12  if __name__ == "__main__":
13      import doctest
14      doctest.testmod()
```

Der im Beispiel verwendete Name ist eine Referenz an den Schöpfer von Python, Guido van Rossum. Das Anwendungsbeispiel in den Zeilen 6 und 7 verwendet die Formatierung der Python-Shell nicht nur, weil sich der Code auf diese Weise direkt nachvollziehen lässt, sondern weil das `doctest`-Modul dieses Format erwartet. Gegebenenfalls sind auch mit `...` eingeleitete Fortsetzungszeilen erlaubt. Folgt nach der Ausgabe noch anderer Text, so muss dieser durch eine Leerzeile abgetrennt sein.

Der Code in den letzten drei Zeilen unseres Beispiels führt dazu, dass die Ausführung des Skripts den in der Dokumentation enthaltenen Code testet:

```
$ python example.py
$
```

Der Umstand, dass hier keine Ausgabe erzeugt wird, ist ein gutes Zeichen, denn er bedeutet, dass es bei der Durchführung der Tests keine Fehler gab. Das Auftreten eines Fehlers hätte dagegen zu einer entsprechenden Ausgabe geführt. Vielleicht will man aber wissen, ob und, wenn ja, welche Tests durchgeführt wurden. Hierzu verwendet man die Kommandozeilenoption `-v` für *verbose*, die hier nach dem Namen des Skripts stehen muss:

```
gert@teide:[...]/manuskript: python example.py -v
Trying:
    welcome('Guido')
Expecting:
    'Hallo Guido!'
ok
1 items had no tests:
    __main__
1 items passed all tests:
   1 tests in __main__.welcome
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
```

Der Ausgabe entnimmt man, dass ein Test erfolgreich durchgeführt wurde und zu dem erwarteten Ergebnis geführt habt. Will man diese ausführliche Ausgabe unabhängig von einer Kommandozeilenoption erzwingen, kann man beim Aufruf von `testmod` die Variable `verbose` auf `True` setzen.

Alternativ zu der bisher beschriebenen Vorgehensweise könnte man die letzten drei Zeilen unseres Beispielcodes weglassen und das `doctest`-Modul beim Aufruf des Skripts laden. Will man eine ausführliche Ausgabe erhalten, so hätte der Aufruf die folgende Form:

```
$ python -m doctest -v example.py
```

Den Fehlerfall illustriert ein Beispiel, in dem eine englischsprachige Ausgabe erwartet wird

```
def welcome(name):
    """
    be nice and greet somebody
    name: name of the person

    >>> welcome('Guido')
    'Hello Guido!'

    """
    return 'Hallo {}!'.format(name)
```

und das zu folgendem Resultat führt:

```
$ python -m doctest example.py
**********************************************************************
File "example.py", line 6, in example.welcome
Failed example:
    welcome('Guido')
Expected:
    'Hello Guido!'
Got:
    'Hallo Guido!'
**********************************************************************
1 items had failures:
   1 of   1 in example.welcome
***Test Failed*** 1 failures.
```

Bei Fehlern werden die Details auch ohne die Option `-v` ausgegeben.

Im Rahmen des *test-driven developments* könnte man als eine Art Wunschliste noch weitere Tests einbauen. Zum Beispiel soll auch ohne Angabe eines Namens eine sinnvolle Ausgabe erfolgen, und es soll auch eine Ausgabe in anderen Sprachen möglich sein.

```
def welcome(name):
    """
    be nice and greet somebody
    name: name of the person

    >>> welcome()
    'Hello!'

    >>> welcome(lang='de')
    'Hallo!'

    >>> welcome('Guido')
    'Hello Guido!'

    """
    return 'Hallo {}!'.format(name)
```

Die im Dokumentationsstring formulierten Anforderungen führen natürlich zunächst zu Fehlern:

```
$ python -m doctest example.py
**********************************************************************
File "example.py", line 6, in example.welcome
Failed example:
    welcome()
Exception raised:
    Traceback (most recent call last):
      File "/opt/anaconda3/lib/python3.6/doctest.py", line 1330, in __run
        compileflags, 1), test.globs)
      File "<doctest example.welcome[0]>", line 1, in <module>
        welcome()
    TypeError: welcome() missing 1 required positional argument: 'name'
**********************************************************************
File "example.py", line 9, in example.welcome
Failed example:
    welcome(lang='de')
Exception raised:
    Traceback (most recent call last):
      File "/opt/anaconda3/lib/python3.6/doctest.py", line 1330, in __run
        compileflags, 1), test.globs)
      File "<doctest example.welcome[1]>", line 1, in <module>
        welcome(lang='de')
    TypeError: welcome() got an unexpected keyword argument 'lang'
**********************************************************************
```

```
File "example.py", line 12, in example.welcome
Failed example:
    welcome('Guido')
Expected:
    'Hello Guido!'
Got:
    'Hallo Guido!'
**********************************************************************
1 items had failures:
   3 of   3 in example.welcome
***Test Failed*** 3 failures.
```

Der Code muss nun so lange angepasst werden, bis alle Tests korrekt durchlaufen, wie dies für das folgende Skript der Fall ist.

```
def welcome(name='', lang='en'):
    """
    be nice and greet somebody
    name: name of the person, may be empty
    lang: two character language code

    >>> welcome()
    'Hello!'

    >>> welcome(lang='de')
    'Hallo!'

    >>> welcome('Guido')
    'Hello Guido!'

    """
    greetings = {'de': 'Hallo',
                 'en': 'Hello',
                 'fr': 'Bonjour'}
    try:
        greeting = greetings[lang]
    except KeyError:
        errmsg = 'unknown language: {}'.format(lang)
        raise ValueError(errmsg)
    if name:
        greeting = ' '.join([greeting, name])
    return greeting+'!'
```

Da dieser Code zu einer `ValueError`-Ausnahme führt, wenn eine nicht implementierte Sprache angefordert wird, stellt sich die Frage, wie dieses Verhalten getestet werden kann. Das Problem besteht hier darin, dass die Ausgabe recht komplex sein kann. Der Aufruf `welcome('Guido', lang='nl')` führt zu:

```
Traceback (most recent call last):
  File "example.py", line 21, in welcome
    greeting = greetings[lang]
KeyError: 'nl'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "example.py", line 29, in <module>
    welcome('Guido', lang='nl')
  File "example.py", line 24, in welcome
    raise ValueError(errmsg)
ValueError: unknown language: nl
```

Für den Test im Dokumentationsstring müssen allerdings nur die erste Zeile, die die Ausnahme ankündigt, sowie

die letzte Zeile, die die Ausnahme spezifiziert, angegeben werden, wie dies die Zeilen 16-18 im folgenden Code zeigen.

```
1  def welcome(name='', lang='en'):
2      """
3      be nice and greet somebody
4      name: name of the person, may be empty
5      lang: two character language code
6
7      >>> welcome()
8      'Hello!'
9
10     >>> welcome(lang='de')
11     'Hallo!'
12
13     >>> welcome('Guido')
14     'Hello Guido!'
15
16     >>> welcome('Guido', 'nl')
17     Traceback (most recent call last):
18     ValueError: unknown language: nl
19
20     """
21     greetings = {'de': 'Hallo',
22                  'en': 'Hello',
23                  'fr': 'Bonjour'}
24     try:
25         greeting = greetings[lang]
26     except KeyError:
27         errmsg = 'unknown language: {}'.format(lang)
28         raise ValueError(errmsg)
29     if name:
30         greeting = ' '.join([greeting, name])
31     return greeting+'!'
```

In diesem Zusammenhang ist auch eine der Direktiven nützlich, die das `doctest`-Modul bereitstellt. Gibt man die Direktive `+ELLIPSIS` an, so kann `...` beliebigen Text in der betreffenden Zeile ersetzen. Wenn uns also die Fehlermeldung nicht genauer interessiert, können wir folgenden Test verwenden:

```
"""
>>> welcome('Guido', 'nl')  # doctest: +ELLIPSIS
Traceback (most recent call last):
ValueError: ...

"""
```

Tests, die nicht oder vorläufig nicht durchgeführt werden sollen, kann man mit der `+SKIP`-Direktive wie folgt markieren:

```
"""
>>> welcome('Guido', 'nl')  # doctest: +SKIP
'Goedendag Guido!'

"""
```

Weitere Direktiven, wie das gelegentlich nützliche `+NORMALIZE_WHITESPACE`, sind in der Dokumentation des `doctest`-Moduls zu finden.

Interessant ist, dass diese Art der Tests nicht nur in Dokumentationsstrings verwendet werden kann, sondern in beliebigen Texten. So lässt sich der Code in dem Text

```
Eine einfache Verzweigung in Python:
>>> x = 1
```

```
>>> if x < 0:
...     print('x ist negativ')
... else:
...     print('x ist nicht negativ')
x ist nicht negativ

Am Ende des Tests muss sich eine
Leerzeile befinden.
```

leicht testen:

```
$ python -m doctest -v example.txt
Trying:
    x = 1
Expecting nothing
ok
Trying:
    if x < 0:
        print('x ist negativ')
    else:
        print('x ist nicht negativ')
Expecting:
    x ist nicht negativ
ok
1 items passed all tests:
   2 tests in example.txt
2 tests in 1 items.
2 passed and 0 failed.
Test passed.
```

Doctests sind für einfachere Testsituationen sehr nützlich, da sie leicht zu schreiben sind und gleichzeitig die Dokumentation von Code unterstützen. Allerdings sind sie für komplexere Testszenarien, insbesondere im numerischen Bereich, weniger gut geeignet. Dann greift man eher auf *unit tests* zurück, die im folgenden Abschnitt beschrieben werden.

6.3 Das `unittest`-Modul

Beim Erstellen von Tests stellt sich zum einen die Frage nach der technischen Umsetzung, zum anderen aber auch danach, was ein Test sinnvollerweise überprüft. Da *unit tests* potentiell komplexer sein können als *doctests* rückt die zweite Frage hier etwas stärker in den Vordergrund. Wir wollen beide Aspekte, den technischen und den konzeptionellen, am Beispiel eines Programms zur Berechnung von Zeilen eines pascalschen Dreiecks diskutieren. Das Skript `pascal.py`

Quellcode 6.1: Code zur Berechnung von Zeilen eines pascalschen Dreiecks.

```
1  def pascal_line(n):
2      x = 1
3      yield x
4      for k in range(n):
5          x = x*(n-k)//(k+1)
6          yield x
7
8  if __name__ == '__main__':
9      for n in range(7):
10         line = ' '.join(map(lambda x: '{:2}'.format(x), pascal_line(n)))
11         print(str(n)+line.center(25))
```

erzeugt mit Hilfe der Zeilen 8-11 die Ausgabe

```
0             1
1           1   1
2         1   2   1
3       1   3   3   1
4     1   4   6   4   1
5   1   5  10  10   5   1
6 1   6  15  20  15   6   1
```

wobei jede Zeile durch einen Aufruf der Funktion `pascal_line` bestimmt wird. Getestet werden soll nur diese in den ersten sechs Zeilen definierte Funktion.

Ein offensichtlicher Weg, die Funktion zu testen, besteht darin, ausgewählte Zeilen des pascalschen Dreiecks zu berechnen und mit dem bekannten Ergebnis zu vergleichen. Hierzu erstellt man ein Testskript, das wir `test_pascal.py` nennen wollen:

```
1  from unittest import main, TestCase
2  from pascal import pascal_line
3
4  class TestExplicit(TestCase):
5      def test_n0(self):
6          self.assertEqual(list(pascal_line(0)), [1])
7
8      def test_n1(self):
9          self.assertEqual(list(pascal_line(1)), [1, 1])
10
11     def test_n5(self):
12         self.assertEqual(list(pascal_line(5)), [1, 5, 10, 10, 5, 1])
13
14 if __name__ == '__main__':
15     main()
```

Da dieses Testskript zunächst unabhängig von dem zu testenden Skript ist, muss die zu testende Funktion in Zeile 2 importiert werden. Die verschiedenen Testfälle sind als Methoden einer von `unittest.TestCase` abgeleiteten Klasse implementiert. Dabei ist wichtig, dass der Name der Methoden mit `test` beginnen, um sie von eventuell vorhandenen anderen Methoden zu unterscheiden. Wie wir später noch sehen werden, können mehrere Testklassen, wie hier `TestExplicit`, implementiert werden, um auf diese Weise eine Gliederung der Testfälle zu erreichen. Der eigentliche Test erfolgt in diesem Fall mit einer Variante der `assert`-Anweisung, die das `unittest`-Modul zur Verfügung stellt. Dabei wird auf Gleichheit der beiden Argumente getestet. Wir werden später noch sehen, dass auch andere Test möglich sind.

Die Ausführung der Tests wird durch die letzten beiden Zeilen des Testskripts veranlasst. Man erhält als Resultat:

```
$ python test_pascal.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Offenbar sind alle drei Tests erfolgreich durchgeführt worden. Dies wird unter anderem auch durch die drei Punkte in der zweiten Zeile angezeigt.

Um einen Fehlerfall zu illustrieren, bauen wir nun einen Fehler ein, und zwar der Einfachheit halber in das Testskript. Üblicherweise wird sich der Fehler zwar im zu testenden Skript befinden, aber das spielt hier keine Rolle. Das Testskript mit der fehlerhaften Zeile 12

```
1  from unittest import main, TestCase
2  from pascal import pascal_line
3
4  class TestExplicit(TestCase):
5      def test_n0(self):
6          self.assertEqual(list(pascal_line(0)), [1])
```

```
7
8       def test_n1(self):
9           self.assertEqual(list(pascal_line(1)), [1, 1])
10
11      def test_n5(self):
12          self.assertEqual(list(pascal_line(5)), [1, 4, 6, 4, 1])
13
14  if __name__ == '__main__':
15      main()
```

liefert nun die Ausgabe:

```
$ python test_pascal.py
..F
======================================================================
FAIL: test_n5 (__main__.TestExplicit)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_pascal.py", line 12, in test_n5
    self.assertEqual(list(pascal_line(5)), [1, 4, 6, 4, 1])
AssertionError: Lists differ: [1, 5, 10, 10, 5, 1] != [1, 4, 6, 4, 1]

First differing element 1:
5
4

First list contains 1 additional elements.
First extra element 5:
1

- [1, 5, 10, 10, 5, 1]
+ [1, 4, 6, 4, 1]

----------------------------------------------------------------------
Ran 3 tests in 0.003s

FAILED (failures=1)
```

Einer der drei Tests schlägt erwartungsgemäß fehl, wobei genau beschrieben wird, wo der Fehler aufgetreten ist und wie er sich manifestiert hat. In der zweiten Zeile deutet das `F` auf einen fehlgeschlagenen Test hin. Wenn erwartet wird, dass ein Test fehlschlägt, kann man ihn mit einem `@expectedFailure`-Dekorator versehen. Dann würde die Ausgabe folgendermaßen aussehen:

```
$ python test_pascal.py
..x
----------------------------------------------------------------------
Ran 3 tests in 0.003s

OK (expected failures=1)
```

Wenn wir die Testmethode `test_n5` wieder korrigieren, würden wir stattdessen

```
gert@teide:[...]/manuskript: python test_pascal.py
..u
----------------------------------------------------------------------
Ran 3 tests in 0.000s

FAILED (unexpected successes=1)
```

erhalten.

Während das Testen auf die beschriebene Weise noch praktikabel ist, ändert sich das für große Argumente. Das Testen für größere Argumente sollte man vor allem dann in Betracht ziehen, wenn man solche Argumente in der

Praxis verwenden möchte, da es dort eventuell zu unerwarteten Problemen kommen kann.

Als Alternative zur Verwendung des expliziten Resultats bietet es sich an auszunutzen, dass die Summe aller Einträge einer Zeile im pascalschen Dreieck gleich 2^n ist, während die alternierende Summe verschwindet. Diese beiden Tests haben die Eigenschaft, dass sie unabhängig von dem verwendeten Algorithmus sind und somit etwaige Fehler, zum Beispiel durch eine fehlerhafte Verwendung der Integerdivision, aufdecken. Der zusätzliche Code in unserem Testskript könnte folgendermaßen aussehen:

```
class TestSums(TestCase):
    def test_sum(self):
        for n in (10, 100, 1000, 10000):
            self.assertEqual(sum(pascal_line(n)), 2**n)

    def test_alternate_sum(self):
        for n in (10, 100, 1000, 10000):
            self.assertEqual(sum(alternate(pascal_line(n))), 0)

def alternate(g):
    sign = 1
    for elem in g:
        yield sign*elem
        sign = -sign
```

Dabei haben wir einen Generator definiert, der wechselnde Vorzeichen erzeugt. Auf diese Weise lässt sich der eigentliche Testcode kompakt und übersichtlich halten.

Eine weitere Möglichkeit für einen guten Test besteht darin, das Konstruktionsverfahren einer Zeile aus der vorhergehenden Zeile im pascalschen Dreieck zu implementieren. Dies leistet der folgende zusätzliche Code:

```
from itertools import chain

class TestAdjacent(TestCase):
    def test_generate_next_line(self):
        for n in (10, 100, 1000, 10000):
            expected = [a+b for a, b
                        in zip(chain(zero(), pascal_line(n)),
                               chain(pascal_line(n), zero()))]
            result = list(pascal_line(n+1))
            self.assertEqual(result, expected)

def zero():
    yield 0
```

Hier wird die `chain`-Funktion aus dem `itertools`-Modul verwendet, um die Ausgabe zweier Generatoren aneinanderzufügen.

Bei den *doctests* hatten wir gesehen, dass es sinnvoll sein kann zu überprüfen, ob eine Ausnahme ausgelöst wird. In unserem Beispiel sollte dies geschehen, wenn das Argument der Funktion `pascal_line` eine negative ganze Zahl ist, da dann der verwendete Algorithmus versagt. Die notwendige Ergänzung ist in dem folgenden Codestück gezeigt.

```
1  def pascal_line(n):
2      if n < 0:
3          raise ValueError('n may not be negative')
4      x = 1
5      yield x
6      for k in range(n):
7          x = x*(n-k)//(k+1)
8          yield x
```

Der zugehörige Test könnte folgendermaßen aussehen:

```
class TestParameters(TestCase):
    def test_negative_int(self):
        with self.assertRaises(ValueError):
            next(pascal_line(-1))
```

Die Verwendung von `assertRaises` muss nicht zwingend in einem `with`-Kontext erfolgen, macht den Code aber sehr übersichtlich. Da die Ausnahme erst dann ausgelöst wird, wenn ein Wert von dem Generator angefordert wurde, ist in der letzten Zeile die Verwendung von `next` erforderlich.

Bisher hatten wir es weder bei *doctests* noch bei *unit tests* mit Gleitkommazahlen zu tun, die jedoch beim numerischen Arbeiten häufig vorkommen und eine besondere Schwierigkeit beim Testen mit sich bringen. Um dies zu illustrieren, lassen wir in unserer Funktion `pascal_line` auch Gleitkommazahlen als Argument zu. So lassen sich zum Beispiel mit `pascal_line(1/3)` die Taylorkoeffizienten von

$$\sqrt[3]{1+x} = 1 + \frac{1}{3}x - \frac{1}{9}x^2 + \frac{5}{81}x^3 + \dots$$

bestimmen. Ist das Argument keine nichtnegative ganze Zahl, so wird der Generator potentiell unendlich viele Werte erzeugen. Die angepasste Version unserer Funktion sieht folgendermaßen aus:

Quellcode 6.2: Erweiterung der Funktion aus Quellcode 6.1 für das pascalsche Dreieck auf Gleitkommaargumente.

```
def pascal_line(n):
    x = 1
    yield x
    k = 0
    while n-k != 0:
        x = x*(n-k)/(k+1)
        k = k+1
        yield x
```

Die Koeffizienten der obigen Taylorreihe erhalten wir dann mit

```
p = pascal_line(1/3)
for n in range(4):
    print(n, next(p))
```

zu

```
0 1
1 0.3333333333333333
2 -0.11111111111111112
3 0.0617283950617284
```

Wir erweitern unsere Tests entsprechend:

```
class TestParameters(TestCase):
    @skip('only for integer version')
    def test_negative_int(self):
        with self.assertRaises(ValueError):
            next(pascal_line(-1))

class TestFractional(TestCase):
    def test_one_third(self):
        p = pascal_line(1/3)
        result = [next(p) for _ in range(4)]
        expected = [1, 1/3, -1/9, 5/81]
        self.assertEqual(result, expected)
```

Der erste Block zeigt beispielhaft, wie man eine Testfunktion mit Hilfe des `@skip`-Dekorators markieren kann, so dass diese nicht ausgeführt wird. Dazu muss allerdings zunächst `skip` aus dem `unittest`-Modul importiert werden. Auch die Testfunktionen `test_sum`, `test_alternate_sum` und `test_generate_next_line`

sollten für die Gleitkommaversion auf diese Weise deaktiviert werden, da sie nicht mehr korrekt funktionieren, zum Beispiel weil ein Überlauf auftritt. Als Testergebnis erhält man dann:

```
s...Fsss
======================================================================
FAIL: test_one_third (__main__.TestFractional)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_pascal.py", line 47, in test_one_third
    self.assertEqual(result, expected)
AssertionError: Lists differ: [1, 0.3333333333333333, -0.1111111111111112, 0.
↪0617283950617284] != [1, 0.3333333333333333, -0.1111111111111111, 0.
↪06172839506172839]

First differing element 2:
-0.1111111111111112
-0.1111111111111111

- [1, 0.3333333333333333, -0.1111111111111112, 0.0617283950617284]
?                                             -                  ^

+ [1, 0.3333333333333333, -0.1111111111111111, 0.06172839506172839]
?                                                                ^^


----------------------------------------------------------------------
Ran 8 tests in 0.004s

FAILED (failures=1, skipped=4)
```

Neben den vier nicht ausgeführten Tests, die wir mit dem `@skip`-Dekorator versehen hatten, wird hier noch ein fehlgeschlagener Test aufgeführt, bei dem es sich um unseren neuen Test der Gleitkommaversion handelt. Der Vergleich des erhaltenen und des erwarteten Resultats zeigt, dass die Ursache in Rundungsfehlern liegt.

Es gibt verschiedene Möglichkeiten, mit solchen Rundungsfehlern umzugehen. Das `unittest`-Modul bietet die Methode `assertAlmostEqual` an, die allerdings den Nachteil hat, nicht auf Listen anwendbar zu sein. Außerdem lässt sich dort nur die Zahl der Dezimalstellen angeben, die bei der Rundung zu berücksichtigen sind. Standardmäßig sind dies 7 Stellen. Eine mögliche Lösung wäre also:

```
class TestFractional(TestCase):
    def test_one_third(self):
        p = pascal_line(1/3)
        result = [next(p) for _ in range(4)]
        expected = [1, 1/3, -1/9, 5/81]
        for r, e in zip(result, expected):
            self.assertAlmostEqual(r, e)
```

Seit Python 3.5 gibt es auch die Möglichkeit, die Funktion `isclose` aus dem `math`-Modul zu verwenden, die es erlaubt, den absoluten und relativen Fehler mit `abs_tol` bzw. `rel_tol` bequem zu spezifizieren. Standardmäßig ist der absolute Fehler auf Null und der relative Fehler auf 10^{-9} gesetzt. Der Test könnte dann folgendermaßen aussehen:

```
class TestFractional(TestCase):
    def test_one_third(self):
        p = pascal_line(1/3)
        result = [next(p) for _ in range(4)]
        expected = [1, 1/3, -1/9, 5/81]
        for r, e in zip(result, expected):
            self.assertTrue(math.isclose(r, e, rel_tol=1e-10))
```

Auch in diesem Fall muss man alle Elemente explizit durchgehen, was den Testcode unnötig kompliziert macht. Abhilfe kann hier NumPy mit seinem `testing`-Modul schaffen, auf das wir im nächsten Abschnitt eingehen werden.

Zuvor wollen wir aber noch kurz eine Testsituation ansprechen, bei der der eigentliche Test eine Vorbereitung sowie Nacharbeit erfordert. Dies ist zum Beispiel beim Umgang mit Datenbanken der Fall, wo Tests nicht an Originaldaten durchgeführt werden. Stattdessen müssen zunächst Datentabellen für den Test angelegt und am Ende wieder entfernt werden.

In dem folgenden Beispiel soll eine Funktion zum Einlesen von Gleitkommazahlen getestet werden. Dazu müssen wir zunächst eine temporäre Datei erzeugen, die dann im Test eingelesen werden kann. Am Ende soll die temporäre Datei gelöscht werden.

```
import os
from unittest import TestCase
from tempfile import NamedTemporaryFile

def convert_to_float(datalist):
    return list(map(float, datalist.strip("\n").split(";")))

def read_floats(filename):
    with open(filename, "r") as file:
         data = list(map(convert_to_float, file.readlines()))
    return data

class testReadData(TestCase):
    def setUp(self):
        """speichere Testdaten in temporärer Datei

        """
        file = NamedTemporaryFile("w", delete=False)
        self.filename = file.name
        self.data = [[1.23, 4.56], [7.89, 0.12]]
        for line in self.data:
            file.write(";".join(map(str, line)))
            file.write("\n")
        file.close()

    def test_read_floats(self):
        """teste korrektes Einlesen der Gleitkommazahlen

        """
        self.assertEqual(self.data,
                         read_floats(self.filename))

    def tearDown(self):
        """lösche temporäre Datei

        """
        os.remove(self.filename)
```

Zunächst werden die beiden zum Einlesen verwendeten Funktionen definiert, wobei aus dem Test heraus die Funktion `read_floats` aufgerufen wird. In der Testklasse gibt es neben der Methode `test_read_floats`, die die Korrektheit des Einlesens überprüft, noch zwei weitere Methoden. Die Methode `setUp` bereitet den Test vor. In unserem Beispiel wird dort die temporäre Datei erzeugt, von der im Laufe des Tests Daten gelesen werden. Die Methode `tearDown` wird nach dem Test ausgeführt und dient hier dazu, die temporäre Datei wieder zu entfernen.

Auch ohne dass wir alle Möglichkeiten des `unittest`-Moduls besprochen haben, dürfte klar geworden sein, dass diese deutlich über die Möglichkeiten des `doctest`-Moduls hinausgehen. Eine Übersicht über weitere Anwendungsmöglichkeiten des `unittest`-Moduls findet man in der zugehörigen Dokumentation, wo inbesondere auch eine vollständige Liste der verfügbaren `assert`-Anweisungen angegeben ist.

6.4 Testen mit NumPy

Das Programmieren von Tests ist gerade beim numerischen Arbeiten sehr wichtig. Bei der Verwendung von NumPy-Arrays ergibt sich allerdings das Problem, dass man normalerweise nicht für jedes Arrayelement einzeln die Gültigkeit einer Testbedingung überprüfen möchte. Wir wollen daher kurz diskutieren, welche Möglichkeiten man in einem solchen Fall besitzt.

Die im folgenden Beispiel definierte Matrix hat nur positive Eigenwerte:

```
In [1]: import numpy as np

In [2]: import numpy.linalg as LA

In [3]: a = np.array([[5, 0.5, 0.1], [0.5, 4, -0.1], [0.1, -0.1, 3]])

In [4]: a
Out[4]:
array([[ 5. ,  0.5,  0.1],
       [ 0.5,  4. , -0.1],
       [ 0.1, -0.1,  3. ]])

In [5]: LA.eigvalsh(a)
Out[5]: array([ 2.97774394,  3.81381575,  5.20844031])

In [6]: np.all(LA.eigvalsh(a) > 0)
Out[6]: True
```

Dies lässt sich in Ausgabe 5 direkt verifizieren. Für einen automatisierten Test ist es günstig, die Positivitätsbedingung für jedes Element auszuwerten und zu überprüfen, ob sie für alle Elemente erfüllt ist. Dies geschieht in Eingabe 6 mit Hilfe der `all`-Funktion, die man in einem Test in der `assert`-Anweisung verwenden würde.

Im letzten Abschnitt hatten wir darauf hingewiesen, dass man bei Tests von Gleitkommazahlen die Möglichkeit von Rundungsfehlern bedenken muss. Dies gilt natürlich genauso, wenn man ganze Arrays von Gleitkommazahlen erzeugt und testen will. In diesem Fall ist es sinnvoll, auf die Unterstützung zurückzugreifen, die NumPy durch sein `testing`-Modul[2] gibt.

Als Beispiel betrachten wir unseren auf Gleitkommaargumente verallgemeinerten Code für das pascalsche Dreieck (Quellcode 6.2). Da wir dort gleich mehrere Werte vergleichen müssen, können wir wie folgt vorgehen:

```
class TestFractional(TestCase):
    def test_one_third(self):
        p = pascal_line(1/3)
        result = [next(p) for _ in range(4)]
        expected = [1, 1/3, -1/9, 5/81]
        np.testing.assert_allclose(result, expected, rtol=1e-10)
```

Hierbei haben wir wie üblich NumPy als `np` importiert. Die Funktion `assert_allclose` erlaubt es ähnlich wie `math.isclose`, bequem den absoluten und relativen Fehler zu spezifizieren, wobei die entsprechenden Variablen hier `atol` bzw. `rtol` lauten. Dabei wird der Unterschied zwischen dem tatsächlichen und dem erwarteten Ergebnis mit der Summe aus `atol` und dem mit `rtol` multiplizierten erwarteten Ergebnis verglichen. Defaultmäßig ist `atol` auf Null gesetzt, so dass nur der relative Fehler von Bedeutung ist, der defaultmäßig den Wert 10^{-7} hat. Gegenüber unseren früheren Tests der verallgemeinerten Funktion `pascal_line` hat der obige Test den Vorteil, dass nicht explizit über die Liste iteriert werden muss und der Testcode somit einfacher und übersichtlicher ist.

[2] Eine detaillierte Liste der verschiedenen Funktionen findet man in der Dokumentation zum Test Support.

KAPITEL 7

Laufzeituntersuchungen

7.1 Allgemeine Vorbemerkungen

Python wird immer wieder vorgeworfen, dass es im Vergleich zu einer Reihe anderer Programmiersprachen langsam sei. Häufig stellt dies kein echtes Problem dar, aber bei Bedarf gibt es zur Optimierung von Pythonskripten eine Reihe von Möglichkeiten. Die konsequente Verwendung von NumPy kann bei dazu geeigneten Anwendungen einen erheblichen Geschwindigkeitsvorteil bringen. Unter Umständen kann es auch sinnvoll sein, besonders zeitkritische Programmteile in C zu implementieren. In diesem Fall bietet sich die Verwendung des bereits in einem früheren Kapitel erwähnten Cython an. Mit dessen Hilfe ist es auch sehr einfach möglich, die Rechenzeit durch das Festlegen des Datentyps von Variablen zu reduzieren. Alternativ bietet sich auch die »just in time«-Kompilierung zum Beispiel mit PyPy[2] oder Numba[3] an, die die Programmausführung beschleunigen kann.

Im Einzelfall sollte man zunächst überlegen, ob das Laufzeitproblem wirklich schwerwiegend ist oder ob man für die Optimierung letztlich mehr Zeit investieren muss als man gewinnt. Es lohnt sich dabei, auf den Altmeister Donald E. Knuth zu hören, der schon vor mehr als 40 Jahren schrieb:

> There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintentance are considered. We *should* forget about small efficiencies, say about 97 % of the time: premature optimization is the root of all evil.
>
> Yet we should not pass up our opportunities in that critical 3 %. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only *after* that code has been identified.[1]

Bevor man also überhaupt mit der Optimierung eines Programms beginnt, sollte man zunächst immer erst feststellen, wo das Programm die meiste Zeit verbringt. Es lohnt sich nicht, Zeit in die Optimierung von Programmteilen zu investieren, deren Laufzeit im Vergleich zur gesamten Laufzeit unerheblich ist. Nach jeder Optimierung wird man erneut den laufzeitkritischsten Programmteil identifizieren, um so in eventuell mehreren Schritten zu einer hoffentlich akzeptablen Laufzeit zu kommen.

Vor der Optimierung eines Programms sollte man immer bedenken, dass sich dabei Fehler einschleichen können. Es nützt alles nicht, wenn man das Programm geändert hat, so dass es viel schneller läuft, dann aber nicht mehr das tut was es eigentlich tun soll. Daher sollte man mindestens eine funktionstüchtigte Version des Programms

[2] Weitere Informationen zu diesem Projekt findet man unter www.pypy.org.
[3] Weitere Informationen zu diesem Projekt findet man unter numba.pydata.org.
[1] D. E. Knuth, Computing Surveys **6**, 261 (1974). Das angegebene Zitat befindet sich auf Seite 268.

aufbewahren, z.B. eine Kopie, die eine Endung `.bak` erhält. Wesentlich besser ist es natürlich, ein Versionskontrollsystem zu verwenden, beispielsweise Git, das wir im Kapitel *Versionskontrolle mit Git* beschrieben haben. Außerdem ist es sinnvoll, Tests zu programmieren, die es erlauben, die neue Programmversion auf Korrektheit zu überprüfen. Techniken hierfür werden im Kapitel *Testen von Programmen* besprochen.

Bevor wir einige Möglichkeiten diskutieren, die Laufzeit von Python-Skripten zu bestimmen, wollen wir im nächsten Abschnitt zunächst auf einige Schwierigkeiten bei der Laufzeitmessung hinweisen.

7.2 Fallstricke bei der Laufzeitmessung

Python stellt mit dem Modul `time` eine Möglichkeit zur Verfügung, die aktuelle Zeit und damit letztlich auch Zeitdifferenzen zu bestimmen.

```
In [1]: import time

In [2]: time.ctime()
Out[2]: 'Thu Dec 22 14:39:30 2016'
```

Auch wenn die aktuelle Zeit hier in einem gut lesbaren Format ausgegeben wird, eignet sich dieses Ergebnis nur schlecht zur Bildung von Zeitdifferenzen. Besser ist es, die Zahl der Sekunden seit Beginn der „Zeitrechnung“ zu bestimmen. Dabei beginnt die Zeitrechnung auf Unix-Systemen am 1.1.1970 um 00:00:00 UTC.

```
In [3]: time.time()
Out[3]: 1482413973.190686
```

Damit lässt sich nun die Zeit bestimmen, die ein bestimmter Python-Code benötigt, wie folgendes Beispiel zeigt.

```
1 import time
2
3 summe = 0
4 start = time.time()
5 for n in range(1000000):
6     summe = summe+1
7 ende = time.time()
8 print('{:5.3f}s'.format(ende-start))
```

Hier wird die Zeitdauer gemessen, die die Schleife in den Zeilen 5 und 6 benötigt. Allerdings ist diese Zeit keineswegs immer genau gleich lang. Das um eine Schleife erweiterte Skript

```
1 import time
2
3 for _ in range(10):
4     summe = 0
5     start = time.time()
6     for n in range(1000000):
7         summe = summe+1
8     ende = time.time()
9     print('{:5.3f}s'.format(ende-start), end='  ')
```

liefert zum Beispiel die folgende Ausgabe

```
0.150s  0.108s  0.104s  0.103s  0.107s  0.106s  0.104s  0.103s  0.103s  0.103s
```

wobei das Ergebnis beim nächsten Lauf oder erst recht auf einem anderen Computer deutlich anders aussehen kann. Es kann also sinnvoll sein, über mehrere Durchläufe zu mitteln, wie es das `timeit`-Modul tut, das wir im nächsten Abschnitt besprechen werden.

Bei der Ermittlung von Laufzeiten ist weiter zu bedenken, dass der Prozessor auch von anderen Aufgaben in Anspruch genommen wird, so dass wir gerade zwar die während des Laufs verstrichene Zeit bestimmt haben, nicht aber die Zeit, die der Prozessor hierfür tatsächlich aufgewendet hat. Dies illustrieren wir im folgenden

Beispiel, in dem wir das Skript zeitweilig pausieren lassen. Damit wird in Zeile 9 simuliert, dass andere Prozesse für eine Unterbrechung der Ausführung unseres Skripts sorgen. Außerdem benutzen wir in den Zeilen 5 und 11 `time.process_time()`, um die vom Prozessor aufgewandte Zeit für den Prozess zu bestimmen, in dem unser Skript abgearbeitet wird.

```
1  import time
2
3  summe = 0
4  start = time.time()
5  start_proc = time.process_time()
6  for n in range(10):
7      for m in range(100000):
8          summe = summe+1
9      time.sleep(1)
10 ende = time.time()
11 ende_proc = time.process_time()
12 print('Gesamtzeit: {:5.3f}s'.format(ende-start))
13 print('Systemzeit: {:5.3f}s'.format(ende_proc-start_proc))
```

Die Ausgabe

```
Gesamtzeit: 10.248s
Systemzeit: 0.238s
```

zeigt, dass die Gesamtdauer des Skripts erwartungsgemäß um etwa 10 Sekunden länger ist als die in Anspruch genommene Prozessorzeit.

Vorsicht ist auch geboten, wenn man den zu testenden Codeteil der Übersichtlichkeit halber in eine Funktion auslagert, da dann die Zeit für den Funktionsaufruf relevant werden kann. Dies ist besonders der Fall, wenn die eigentliche Auswertung der Funktion nur sehr wenig Zeit erfordert. So liefert der folgende Code

```
1  import time
2
3  summe = 0
4  start_proc = time.process_time()
5  for n in range(10000000):
6      summe = summe+1
7  ende_proc = time.process_time()
8  print('Systemzeit: {:5.3f}s'.format(ende_proc-start_proc))
```

eine Laufzeit von 1,122 Sekunden, während der äquivalente Code

```
1  import time
2
3  def increment_by_one(x):
4      return x+1
5
6  summe = 0
7  start_proc = time.process_time()
8  for n in range(10000000):
9      increment_by_one(summe)
10 ende_proc = time.process_time()
11 print('Systemzeit: {:5.3f}s'.format(ende_proc-start_proc))
```

mit 1,529 Sekunden gemessen wurde und somit um fast 40 Prozent langsamer läuft.

Unabhängig von den genannten Problemen bedeutet jede Laufzeitmessung immer einen Eingriff in die Ausführung des Skripts, so dass die gemessene Laufzeit unter Umständen deutlich gegenüber der normalen Laufzeit des entsprechenden Codes erhöht sein kann.

Die in den Beispielen verwendete Methode der Laufzeitbestimmung hat Nachteile. Unter anderem erfordert sie eine explizite Modifizierung des Codes, was häufig unerwünscht ist. Im Folgenden besprechen wir einige ausgewählte Alternativen, die entsprechend den jeweiligen Erfordernissen eingesetzt werden können.

7.3 Das Modul `timeit`

Um die Laufzeit von Einzeilern oder kleineren Codeteilen zu testen, kann man das Python-Modul `timeit` heranziehen. Dies ist zum Beispiel dann nützlich, wenn man sich ein Bild davon machen möchte, welche Codevariante die schnellere sein wird. Im Allgemeinen wird dabei über mehrere oder sogar viele Wiederholungen gemittelt, um zu einem möglichst zuverlässigen Ergebnis zu kommen. Die wohl einfachste Möglichkeit, `timeit` einzusetzen, besteht in der Benutzung der IPython-Shell.

Einen Laufzeitvergleich zwischen zwei Arten eine Zahl zu quadrieren, kann man in IPython folgendermaßen vornehmen:

```
In [1]: n = 5

In [2]: %timeit n*n
10000000 loops, best of 3: 166 ns per loop

In [3]: %timeit n**2
1000000 loops, best of 3: 252 ns per loop
```

Das Prozentzeichen wird `timeit` vorangestellt, um es als so genannten »magischen Befehl« zu kennzeichnen, also einen Befehl der IPython-Shell und nicht ein Python-Kommando. Da `timeit` in diesem Fall nicht als Python-Kommando interpretiert werden kann, könnte man sogar auf das Prozentzeichen verzichten. Es zeigt sich, dass die Quadrierung durch Multiplikation mit 166 Nanosekunden schneller ausgeführt wird als die Quadrierung durch Potenzierung, die 252 Nanosekunden benötigt. Natürlich hängt die Laufzeit vom verwendeten Prozessor ab und ist auch nicht unbedingt auf die letzte Stelle genau reproduzierbar.

Wie in der Ausgabe dieses Beispiels zu sehen ist, wird der Befehl, dessen Laufzeit bestimmt werden soll, mehrfach ausgeführt. Dabei wird die Zahl der Wiederholungen automatisch so bestimmt, dass sich eine vernünftige Gesamtlaufzeit ergibt.

Um die Laufzeit von mehrzeiligem Code zu untersuchen, wendet man den magischen `timeit`-Befehl auf eine ganze Zelle an, indem man ein zweites Prozentzeichen voranstellt. Dies ist in folgendem Beispiel gezeigt.

```
In [4]: %%timeit
   ...: summe = 0
   ...: for n in range(1000):
   ...:     summe = summe+n
   ...:
10000 loops, best of 3: 104 us per loop
In [5]: %timeit sum(range(1000))
10000 loops, best of 3: 22.2 us per loop
```

Im ersten Fall verwenden wir `%%timeit` mit zwei Prozentzeichen, damit sich dieser Befehl auf die nächsten drei Zeilen und nicht nur die nächste Zeile bezieht. Im zweiten Fall genügt dagegen wiederum `%timeit`. In diesem Beispiel liegt die Ausführungszeit im Mikrosekundenbereich, wobei die explizite Schleife fast fünfmal mehr Zeit benötigt.

Auch wenn man mit der IPython-Shell sehr bequem die Laufzeit von Codestücken untersuchen kann, mag es gelegentlich notwendig sein, das `timeit`-Modul direkt in einem Python-Skript einzusetzen. Daher wollen wir uns nun die Anwendung dieses Moduls ansehen.

Das folgende Beispiel untersucht den Laufzeitunterschied bei der Berechnung des Sinus mit Hilfe des `math`-Moduls und mit NumPy in Abhängigkeit von der Anzahl der Funktionsargumente.

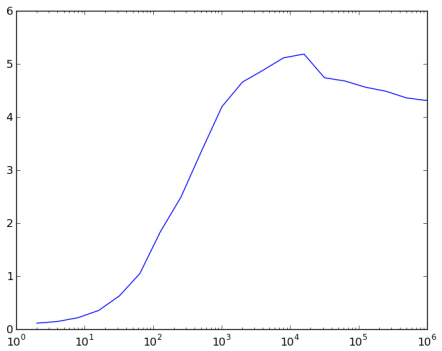
```
1 import numpy as np
2 import math
3 import timeit
4 import matplotlib.pyplot as plt
5
6 def f_numpy(nmax):
7     x = np.linspace(0, np.pi, nmax)
8     result = np.sin(x)
```

```
9
10 def f_math(nmax):
11     dx = math.pi/(nmax-1)
12     result = [math.sin(n*dx) for n in range(nmax)]
13
14 x = []
15 y = []
16 for n in np.logspace(0.31, 6, 20):
17     nint = int(n)
18     t_numpy = timeit.timeit("f_numpy({})".format(nint),
19                             "from __main__ import f_numpy",
20                             number=20)
21     t_math = timeit.timeit("f_math({})".format(nint),
22                            "from __main__ import f_math",
23                            number=20)
24     x.append(nint)
25     y.append(t_math/t_numpy)
26 plt.plot(x, y)
27 plt.xscale("log")
28 plt.show()
```

Zunächst definieren wir in den Zeilen 6 bis 12 zwei Funktionen, die jeweils den Sinus für eine vorgegebene Anzahl von Argumenten berechnen, einmal mit Hilfe von NumPy und einmal mit Hilfe des `math`-Moduls. In den Zeilen 16 bis 25 wird für verschiedene Argumentanzahlen die Laufzeit für die beiden Varianten bestimmt. Sehen wir uns einen der Aufrufe zur Laufzeitbestimmung genauer an, konkret den Code in den Zeilen 18 bis 20. Nachdem wir in Zeile 3 das `timeit`-Modul geladen hatten, können wir in Zeile 18 die `timeit`-Funktion aus diesem Modul aufrufen. Das erste Argument enthält den auszuführenden Code, in unserem Fall also einfach den Funktionsaufruf von `f_numpy`.

Nachdem der Code als String zu übergeben ist, können wir problemlos in der gezeigten Weise ein Argument oder auch mehrere übergeben. Da die von `timeit` aufgerufenen Funktion keinen Zugriff auf den Namensraum des umgebenden Skripts besitzt, würde es nicht funktionieren, das Argument einfach als `nint` in dem String unterzubringen. Tatsächlich ist nicht einmal die Funktion `f_numpy` bekannt. Der `timeit`-Funktion wird daher in Zeile 19 explizit mitgeteilt, dass zunächst aus unserem Hauptskript, auf das mit `__main__` Bezug genommen wird, `f_numpy` zu importieren ist. In Zeile 20 verlangen wir schließlich noch, dass zwanzig Funktionsläufe durchgeführt werden sollen, um eine gemittelte Laufzeit berechnen zu können. Eine automatische Bestimmung einer sinnvollen Zahl von Wiederholungen nimmt `timeit` hier im Gegensatz zur Verwendung in IPython nicht vor.

Wie die folgende Abbildung zeigt, bietet NumPy für sehr kleine Argumentanzahlen keinen Geschwindigkeitsvorteil, ganz im Gegenteil. Dies hängt damit zusammen, dass im Zusammenhang mit der Verwendung von Arrays einiges an Zusatzarbeit anfällt. Bei mehr als etwa 100 Argumenten erlaubt NumPy in unserem Fall jedoch eine schnellere Berechnung des Sinus. Der Geschwindigkeitsvorteil kann auf der hier verwendeten Hardware immerhin einen Faktor 4 bis 5 betragen.



7.4 Das Modul `cProfile`

Das `timeit`-Modul, das wir gerade beschrieben haben, ist sehr gut geeignet, um die Laufzeit eines bestimmten Codesegments zu untersuchen. Bei der Optimierung eines Programms interessiert man sich jedoch vor allem dafür, welche Teile des Programms wieviel Zeit benötigen. Dann können die rechenintensiven Codeteile identifiziert und gezielt optimiert werden.

Häufig ist dies jedoch nicht nötig, und es genügt festzustellen, wieviel Zeit in den einzelnen Funktionen oder Methoden verbracht wurde. Dies funktioniert dann besonders gut, wenn man den Code sinnvoll modularisiert, was ja auch im Hinblick auf das Testen von Vorteil ist, wie wir im Kapitel *Testen von Programmen* betont hatten. Im Folgenden werden wir beschreiben, wie man mit Hilfe des Moduls `cProfile` feststellen kann, wieviel Zeit in welchen Funktionen während des Programmlaufs verbracht wird.

Als Beispiel ziehen wir das folgende Skript mit Namen `pi.py` zur Berechnung der Kreiszahl π heran, wobei eine Berechnung auf 100.000 Stellen durchgeführt wird. Das Skript basiert auf dem Gauss-Legendre oder Brent-Salamin-Algorithmus und nutzt aus, dass Python beliebig lange Integers zulässt.

```
from math import sqrt

def division(numerator, denominator, stellen):
    resultat = str(numerator//denominator)+"."
    for n in range(stellen):
        numerator = (numerator % denominator)*10
        resultat = "%s%s" % (resultat, numerator//denominator)
    return resultat

def wurzel_startwert(quadrat):
    """bestimme näherungsweise die Wurzel aus einem langen Integer

       Es wird die Wurzel auf der Basis der ersten 12 oder 13 Stellen
       mit Hilfe des entsprechenden Floats gezogen.
    """
    str_quadrat = str(quadrat)
    nrdigits = len(str_quadrat)
    keepdigits = 12
    if nrdigits % 2:
        keepdigits = keepdigits+1
    lead_sqrt_estimate = sqrt(float(str_quadrat[:keepdigits]))
    return int(lead_sqrt_estimate)*10**((nrdigits-keepdigits)//2)+1

def wurzel(quadrat):
    x = wurzel_startwert(quadrat)
    xold = 0
    while x != xold:
        xold = x
        x = (x*x+quadrat)//(2*x)
    return x

def agm_iteration(a, b):
    return (a+b)//2, wurzel(a*b)

def ausgabe(x, zeilenlaenge=80):
    str_x = "\u03c0="+str(x)+"\u2026"
    while len(str_x) > 0:
        print(str_x[:zeilenlaenge])
        str_x = str_x[zeilenlaenge:]

stellen = 100000
skalenfaktor = 10**(stellen+6)
a = skalenfaktor
b = wurzel(skalenfaktor**2//2)
c_sum = 0
```

```
46 faktor_two = 2
47 while a != b:
48     a, b = agm_iteration(a, b)
49     faktor_two = faktor_two*2
50     c_sum = c_sum+faktor_two*(a*a-b*b)
51 numerator = 4*a**2
52 denominator = skalenfaktor**2-c_sum
53 ergebnis = division(numerator, denominator, stellen)
54 ausgabe(ergebnis)
```

Die gesamte Ausgabe ist zu lang, um sie hier vollständig wiederzugeben, so dass wir uns auf die ersten beiden Zeilen beschränken.

```
π=3.14159265358979323846264338327950288419716939937510582097494459230781640628620
899862803482534211706798214808651328230664709384460955058223172535940812848111 7
```

Von den verschiedenen Varianten, `cProfile` zu benutzen, wählen wir hier eine, bei der wir das zu untersuchende Programm nicht modifizieren müssen. Dazu rufen wir das Modul mit geeigneten Argumenten auf:

```
$ python -m cProfile -o pi.prof pi.py
```

Hierbei wird das Programm `pi.py` unter der Kontrolle des `cProfile`-Moduls ausgeführt. Die Option `-o` legt fest, dass die Ergebnisse in der Datei `pi.prof` gespeichert werden sollen. Dabei handelt es sich um eine Binärdatei, die mit Hilfe des `pstats`-Moduls analysiert werden kann. Dazu geht man folgendermaßen vor:

```
In [1]: import pstats

In [2]: p = pstats.Stats('pi.prof')

In [3]: p.sort_stats('time').print_stats(8)
Fri Dec 23 15:36:56 2016    pi.prof

         2882 function calls in 68.377 seconds

   Ordered by: internal time
   List reduced from 76 to 8 due to restriction <8>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       18   41.008    2.278   49.819    2.768 pi.py:27(wurzel)
        1   17.776   17.776   17.776   17.776 pi.py:4(division)
       18    8.812    0.490    8.812    0.490 pi.py:12(wurzel_startwert)
        1    0.424    0.424   68.377   68.377 pi.py:1(<module>)
       17    0.320    0.019   47.346    2.785 pi.py:36(agm_iteration)
        1    0.024    0.024    0.037    0.037 pi.py:40(ausgabe)
     1251    0.011    0.000    0.011    0.000 {built-in method builtins.print}
     1270    0.002    0.000    0.002    0.000 {built-in method builtins.len}

Out[3]: <pstats.Stats at 0x7f1a26ed4ac8>
```

Nachdem in Eingabe 1 das `pstats`-Modul geladen wurde, wird in Eingabe 2 die zuvor erzeugte binäre Datei `pi.prof` eingelesen. Man erhält so eine `pstats.Stats`-Instanz, die nun analysiert werden kann. In den meisten Fällen wird man die Daten nach der benötigten Zeit sortieren und auch nur die obersten Datensätze ausgeben wollen, da die Gesamtliste unter Umständen recht lang sein kann. In Eingabe 3 sortieren wir mit der `sort_stats`-Methode nach der Zeit, die in der jeweiligen Funktion verbracht wurde. Anschließend wird mit der `print_stats`-Methode dafür gesorgt, dass nur die ersten acht Zeilen ausgegeben werden.

Das Schlüsselwort `time` in der `sort_stats`-Methode verlangt eine Sortierung nach der Zeit, die in der jeweiligen Funktion verbracht wurde. Wird von einer Funktion aus eine andere Funktion aufgerufen, so wird die Uhr für die aufrufende Funktion angehalten. Dies ist zum Beispiel in der Funktion `wurzel` der Fall, die in Zeile 25 die Funktion `wurzel_startwert` aufruft. Für die Funktion `wurzel` wurde gemäß der obigen Ausgabe eine Zeit `tottime` von 41,008 Sekunden gemessen. Diese enthält nicht die 8,812 Sekunden, die von `wurzel_startwert` benötigt werden. Die von `wurzel` benötigte Gesamtzeit lässt sich in der Spalte

`cumtime` (*cumulative time*, also aufsummierte Zeit) zu 49,819 Sekunden ablesen. Dies entspricht bis auf einen Rundungsfehler der Summe der Zeiten, die in `wurzel` und `wurzel_startwert` verbracht wurden. Ist man an den aufsummierten Zeiten interessiert, so kann man das Schlüsselwort `cumtime` in der `sort_stats`-Methode verwenden.

```
In [4]: p.sort_stats('cumtime').print_stats(8)
Fri Dec 23 15:36:56 2016    pi.prof

         2882 function calls in 68.377 seconds

   Ordered by: cumulative time
   List reduced from 76 to 8 due to restriction <8>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   68.377   68.377 {built-in method builtins.exec}
        1    0.424    0.424   68.377   68.377 pi.py:1(<module>)
       18   41.008    2.278   49.819    2.768 pi.py:27(wurzel)
       17    0.320    0.019   47.346    2.785 pi.py:36(agm_iteration)
        1   17.776   17.776   17.776   17.776 pi.py:4(division)
       18    8.812    0.490    8.812    0.490 pi.py:12(wurzel_startwert)
        1    0.024    0.024    0.037    0.037 pi.py:40(ausgabe)
     1251    0.011    0.000    0.011    0.000 {built-in method builtins.print}

Out[4]: <pstats.Stats at 0x7f1a26ed4ac8>
```

Die Ausgabe zeigt auch, dass es nicht immer auf die Zeit ankommt, die pro Aufruf einer Funktion benötigt wird. Diese Information findet sich in der Spalte `percall`. So benötigt `division` in unserem Beispiel 17,776 Sekunden je Aufruf, während `wurzel` nur 2,278 Sekunden je Aufruf benötigt. Allerdings wird `division` nur einmal aufgerufen, während `wurzel` achtzehnmal aufgerufen wird. Damit ist der Beitrag von `wurzel` zur Gesamtlaufzeit erheblich größer als der Beitrag von `division`.

Es kann sinnvoll sein, die in der Spalte `ncalls` angegebene Anzahl der Aufrufe einer Funktion auf Plausibilität zu überprüfen. Gelegentlich stellt sich dabei heraus, dass eine Funktion unnötigerweise mehrfach aufgerufen wird. So kann es vorkommen, dass eine Funktion in einer Schleife aufgerufen wird, obwohl sich die Funktionsargumente in der Schleife nicht ändern. Eine entsprechende Anpassung des Programms kann dann auf einfache Weise zu einer Beschleunigung führen.

Mit den beschriebenen Ausgaben lässt sich nun feststellen, in welchen Teilen des Programms der größte Anteil der Rechenzeit verstreicht. Man kann sich somit bei der Optimierung des Programms auf diese Teile konzentrieren. Dabei kann es natürlich vorkommen, dass nach einer Optimierung andere Programmteile in den Fokus rücken. Es kann aber auch sein, dass man feststellen muss, dass die meiste Rechenzeit in einem Programmteil benötigt wird, der sich nicht mehr optimieren lässt. Dann muss man sich die Frage stellen, ob es sinnvoll ist, die Optimierungsbemühungen überhaupt fortzusetzen, da eine Optimierung der anderen Programmteile kaum eine Auswirkung auf die Gesamtrechenzeit haben wird. Um die Situation einschätzen zu können, sind Laufzeitanalysen, wie wir sie hier vorgestellt haben, praktisch unerlässlich.

7.5 Zeilenorientierte Laufzeitbestimmung

Gelegentlich kann es vorkommen, dass die im letzten Abschnitt beschriebene funktionsbasierte Laufzeitauswertung nicht ausreicht, um ein in Hinblick auf die Laufzeit kritisches Codestück zu identifizieren. In diesem Fall kann man zu einer zeilenorientierten Laufzeitmessung greifen. Wir beschreiben hier das von Robert Kern entwickelte Modul `line_profiler`[4].

Der besseren Übersichtlichkeit wegen empfiehlt es sich, eine zeilenorientierte Laufzeitmessung auf eine einzelne Funktion oder nur wenige Funktionen zu beschränken. Dazu bestimmt man am besten mit den zuvor beschriebenen Methoden die zeitkritischsten Funktionen. Für Funktionen, die mit einem `@profile`-Dekorator versehen sind, wird eine zeilenorientierte Laufzeitmessung durchgeführt. Wir wollen speziell die Funktionen `wurzel` und `wurzel_startwert` betrachten. Der entsprechende Codeteil sieht dann folgendermaßen aus.

[4] Die Quellen zu diesem Modul findet man unter https://github.com/rkern/line_profiler.

```
1     @profile
2     def wurzel_startwert(quadrat):
3         str_quadrat = str(quadrat)
4         nrdigits = len(str_quadrat)
5         keepdigits = 12
6         if nrdigits % 2:
7             keepdigits = keepdigits+1
8         lead_sqrt_estimate = sqrt(float(str_quadrat[:keepdigits]))
9         return int(lead_sqrt_estimate)*10**((nrdigits-keepdigits)//2)+1
10
11    @profile
12    def wurzel(quadrat):
13        x = wurzel_startwert(quadrat)
14        xold = 0
15        while x != xold:
16            xold = x
17            x = xold*xold+quadrat
18            x = x//(2*xold)
19        return x
```

Der restliche Code bleibt unverändert. Wesentlich sind hier die beiden `@profile`-Dekoratoren. Für die folgende Diskussion haben wir den Iterationsschritt des Newton-Verfahrens in zwei Zeilen (17 und 18) aufgeteilt. Außerdem haben wir einen Docstring entfernt, der hier nicht wesentlich ist.

Von der Befehlszeile kann man nun das Skript unter Verwendung der zeilenorientierten Laufzeitmessung ausführen:

```
kernprof -l -v pi.py
```

`kernprof` ist der Name eines Skripts, das die Verwendung des Moduls `line_profiler` automatisiert, wenn man die Option `-l` angibt. Die Option `-v` gibt man an, wenn die Ausgabe direkt angezeigt werden soll. In jedem Fall werden die relevanten Daten ähnlich wie beim `cProfile`-Modul in eine Binärdatei geschrieben. Sofern nicht mit der Option `-o` etwas anderes angegeben wird, ergibt sich der Name der Datei durch Anhängen der Endung `.lprof`. In unserem Falle heißt sie also `pi.py.lprof`. Aus ihr kann man mit

```
python -m line_profiler pi.py.lprof
```

die folgende Ausgabe erzeugen:

```
Timer unit: 1e-06 s

Total time: 8.71038 s
File: pi.py
Function: wurzel_startwert at line 10

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    10                                           @profile
    11                                           def wurzel_startwert(quadrat):
    12        18      8621108 478950.4     99.0      str_quadrat = str(quadrat)
    13        18           61      3.4      0.0      nrdigits = len(str_quadrat)
    14        18           20      1.1      0.0      keepdigits = 12
    15        18           39      2.2      0.0      if nrdigits % 2:
    16                                                   keepdigits = keepdigits+1
    17        18          207     11.5      0.0      lead_sqrt_estimate =
                                                       sqrt(float(str_
↪quadrat[:keepdigits]))
    18        18        88949   4941.6      1.0      return int(lead_sqrt_estimate)
                                                       *10**((nrdigits-keepdigits)/
↪/2)+1

Total time: 49.5045 s
```

```
File: pi.py
Function: wurzel at line 20

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    20                                           @profile
    21                                           def wurzel(quadrat):
    22        18      8710713 483928.5     17.6      x = wurzel_startwert(quadrat)
    23        18           31      1.7      0.0      xold = 0
    24       288          898      3.1      0.0      while x != xold:
    25       270          254      0.9      0.0          xold = x
    26       270      3026189  11208.1      6.1          x = xold*xold+quadrat
    27       270     37766390 139875.5     76.3          x = x//(2*xold)
    28        18           31      1.7      0.0      return x
```

In der Ausgabe zur Funktion `wurzel_startwert` haben wir die Zeilen 17 und 18 wegen der Zeilenlänge nachträglich umgebrochen. Die Ausgabe zeigt uns in der Funktion `wurzel_startwert` nun deutlich, welcher Teil der Funktion für die Ausführungsdauer von fast 9 Sekunden verantwortlich ist, nämlich die Umwandlung eines Integers in einen String. Dieser Schritt ist hier erforderlich, um die Zahl der Ziffern in dem Integer `quadrat` zu bestimmen.

Interessant ist auch die Funktion `wurzel`, die für einen größten Teil der Laufzeit verantwortlich ist. In den Zeilen 26 und 27 sehen wir, dass der Großteil der Zeit im Newton-Iterationsschritt verbracht wird. Dabei spielt die Berechnung des Quadrats von `xold` kaum eine Rolle. Es ist vielmehr die Division in Zeile 27, die einen sehr hohen Zeitaufwand erfordert. Zwar ist die Zeit für die Berechnung des Startwerts in Zeile 22 auf einen einzelnen Aufruf bezogen größer, aber nachdem die Division 270-mal aufgerufen wird, ist sie für mehr als Dreiviertel der Laufzeit der Funktion `wurzel` verantwortlich.

Bei der Programmentwicklung kann es praktisch sein, das Modul `line_profiler` in IPython zu verwenden. Im Folgenden ist ein Beispiel gezeigt, das einen Vergleich zwischen der Wurzelfunktion aus dem `math`-Modul und der Wurzelberechnung mit Hilfe des Newton-Verfahrens anstellt.

```
In [1]: %load_ext line_profiler

In [2]: import math

In [3]: def newton_sqrt(quadrat):
   ...:     x = 1
   ...:     while abs(quadrat-x*x) > 1e-13:
   ...:         x = 0.5*(x*x+quadrat)/x
   ...:     return x
   ...:

In [4]: def comparison(x):
   ...:     sqrt1 = math.sqrt(x)
   ...:     sqrt2 = newton_sqrt(x)
   ...:     print(sqrt1, sqrt2)
   ...:

In [5]: %lprun -f newton_sqrt comparison(500)
22.360679774997898 22.360679774997898
Timer unit: 1e-06 s

Total time: 7e-05 s
File: <ipython-input-3-e6f13bf0d844>
Function: newton_sqrt at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def newton_sqrt(quadrat):
     2         1            4      4.0      5.7      x = 1
     3        10           34      3.4     48.6      while abs(quadrat-x*x) > 1e-
↪13:
```

```
     4         9           29      3.2     41.4          x = 0.5*(x*x+quadrat)/x
     5         1            3      3.0      4.3      return x
```

Zunächst lädt man in Eingabe 1 `line_profiler` als Erweiterung. Nachdem man die nötigen Funktionen definiert und für dieses Beispiel auch noch das `math`-Modul geladen hat, kann man in Eingabe 5 mit Hilfe von `%lprun` die zeilenorientierte Laufzeitmessung ausführen. Dazu gibt man mit der Option `-f` die Funktion an, in der die Laufzeitmessung benötigt wird. Diese Option ersetzt also den `@profile`-Dekorator. Bei Bedarf kann die Option `-f` auch mehrfach angegeben werden. Am Ende steht der Aufruf der Funktion, mit der der gewünschte Code ausgeführt wird, hier also `comparison(500)`.

Nachdem wir uns in diesem Kapitel auf die Messung von Laufzeiten konzentriert hatten, sei abschließend noch angemerkt, dass man auch die Entwicklung des Speicherbedarfs während der Ausführung eines Skripts messen kann. Dies ist besonders dann nützlich, wenn man mit größeren Arrays arbeitet oder an die Grenzen des Arbeitsspeichers stößt. Um im Skript zeilenweise die Entwicklung des Speicherbedarfs messen zu können, verwendet man das Modul `memory_profiler`.

KAPITEL 8

Aspekte des parallelen Rechnens

Bereits in normalen Rechnern sind heutzutage CPUs verbaut, die mehrere Rechenkerne enthalten. Da man es gerade beim numerischen Rechnen immer wieder mit Problemen zu tun hat, bei denen sich Aufgaben parallel erledigen lassen, stellt sich die Frage, wie man in Python diese Parallelverarbeitung realisieren und mehrere Rechenkerne gleichzeitig beschäftigen kann. Damit ließe sich die Ausführung des Programms entsprechend beschleunigen. Dies gilt umso mehr als oft auch Rechencluster zur Verfügung stehen, die viele CPUs enthalten und es erlauben, die numerische Arbeit auf viele Rechenkerne zu verteilen.

Im Zusammenhang mit dem parallelen Rechnen ist zu beachten, dass Python einen sogenannten *Global Interpreter Lock* (GIL) besitzt, der verhindert, dass im Rahmen eines einzigen Python-Prozesses eine echte Parallelverarbeitung realisiert werden kann. Auf diese Problematik werden wir im nächsten Abschnitt eingehen.

Trotz des GIL ist eine Parallelverarbeitung möglich, wenn mehrere Prozesse gestartet werden. Wie dies in Python realisiert wird, werden wir uns im Abschnitt *Parallelverarbeitung in Python* am konkreten Beispiel der Berechnung der Mandelbrotmenge ansehen. Dieses Problem zeichnet sich dadurch aus, dass die parallel zu bearbeitenden Teilaufgaben unabhängig voneinander sind, so dass während der Bearbeitung keine Kommunikation, also zum Beispiel Austausch von Daten, erforderlich ist. Man spricht dann von einem Problem, das *embarrassingly parallel* ist. Wir wollen uns auf diese Klasse von Problemen beschränken, da die Kommunikation zwischen verschiedenen Prozessen bei der Parallelverarbeitung eine Reihe von Problemen aufwirft, deren Diskussion hier den Rahmen sprengen würde.

Im letzten Abschnitt werden wir noch auf Numba eingehen, das schon als sogenannter *Just in Time Compiler* (JIT Compiler) zu einer Beschleunigung der Programmausführung führt. Zusätzlich kann Numba aber auch die parallele Abarbeitung von Python-Skripten unterstützen.

8.1 Threads, Prozesse und der GIL

Moderne Betriebssysteme erlauben es selbst auf nur einem Rechenkern, verschiedene Vorgänge scheinbar parallel ablaufen zu lassen. Dies geschieht dadurch, dass diese Vorgänge abwechselnd Rechenzeit zugewiesen bekommen, so dass ein einzelner Vorgang normalerweise nicht die Ausführung der anderen Vorgänge über eine längere Zeit blockieren kann.

Dabei muss man zwei Arten von Vorgängen unterscheiden, nämlich Prozesse und Threads. Prozesse verfügen jeweils über ihren eigenen reservierten Speicherbereich und über einen eigenen Zugang zu Systemressourcen. Dies bedeutet aber auch, dass das Starten eines Prozesses mit einem gewissen Aufwand verbunden ist. Ein Prozess startet zunächst einen und anschließend eventuell auch mehrere Threads, um verschiedene Aufgaben zu bearbeiten. Threads unterscheiden sich dabei von Prozessen vor allem dadurch, dass sie einen gemeinsamen Speicherbereich

besitzen und den gleichen Zugang zu den Systemressourcen benutzen. Einen Thread zu starten, ist somit deutlich weniger aufwändig als das Starten eines Prozesses.

Da sich Threads einen gemeinsamen Speicherbereich teilen, können sie sehr leicht auf die gleichen Daten zugreifen oder Daten untereinander austauschen. Die Kommunikation von Threads ist also mit wenig Aufwand verbunden. Allerdings birgt der Zugriff auf gemeinsame Daten auch Gefahren. Trifft man nämlich keine geeigneten Vorkehrungen, um das Lesen und Schreiben von Daten in der erforderlichen Reihenfolge zu gewährleisten, kann es dazu kommen, dass ein Thread nicht die richtigen Daten bekommt. Da das Auftreten solcher Fehler davon abhängt, wann genau welcher Thread welche Aufgaben ausführt, sind diese Fehler nicht ohne Weiteres reproduzierbar und daher nicht immer leicht zu identifizieren. Es gibt Techniken wie man den Datenaustausch zwischen Threads in geordnete Bahnen lenken kann, so dass eine parallele Abarbeitung von Threads, das so genannte *Multithreading*, möglich wird. Wir wollen hier jedoch darauf verzichten, diese Techniken weiter zu diskutieren.

Die am häufigsten verwendete Implementation von Python, nämlich das in C geschriebene CPython, verwendet einen sogenannten *Global Interpreter Lock* (GIL). Dieser verhindert, dass ein einzelner Python-Prozess mehrere Threads parallel ausführen kann. Es ist zwar durchaus möglich, in Python[1] Multithreading zu verwenden. Dann sorgt aber der GIL dafür, dass die verschiedenen Threads in Wirklichkeit abwechselnd immer wieder Rechenzeit bekommen, so dass nur der Anschein von paralleler Verarbeitung erweckt wird.

Wenn die Abarbeitung eines Programms durch die Rechenzeit begrenzt ist, führt die Verwendung von Multithreading in Python somit zu keiner Beschleunigung. Im Gegenteil wird der Mehraufwand, der durch den Wechsel zwischen verschiedenen Threads entsteht, eher zu einer Verlangsamung des Programms führen. Es gibt jedoch auch Probleme, deren Bearbeitungsgeschwindigkeit durch Ein- und Ausgabevorgänge bestimmt wird. Ein Beispiel wäre ein Programm, das von vielen Webseiten Daten herunterladen muss, um diese zu bearbeiten. Da ein Thread während des Wartens auf Daten ohnehin untätig ist, kann es in einem solchen Fall auch in Python sinnvoll sein, mehrere Threads zu starten, die dann problemlos ihre benötigte Rechenzeit erhalten können.

Da Programme im numerischen Bereich normalerweise nicht durch Ein- und Ausgabe verlangsamt werden, sondern durch die erforderliche Rechenzeit, werden wir uns im Folgenden nicht mit Multithreading beschäftigen, sondern uns auf die Parallelverarbeitung von Daten durch das Starten von mehreren Prozessen (*multiprocessing*) konzentrieren.

Abschließend sei noch erwähnt, dass Multithreading im numerischen Bereich auch in Python eine Rolle spielen kann, wenn numerische Bibliotheksroutinen zum Einsatz kommen, die beispielsweise in C geschrieben sind und dann nicht mehr unter der Kontrolle des GIL ausgeführt werden müssen. Ein Beispiel hierfür sind eine Reihe von Operationen aus dem Bereich der linearen Algebra bei der Verwendung einer geeignet kompilierten Version von NumPy. Hierzu zählt das mit der Anaconda-Distribution ausgelieferte, mit der Intel® Math Kernel Library (Intel® MKL) kompilierte NumPy. Eine andere Möglichkeit, den GIL zu umgehen, bietet Cython[2], mit dem C-Erweiterungen aus Python-Code erzeugt werden können. Dabei lassen sich Code-Teile, die keine Python-Objekte verwenden, in einem `nogil`-Kontext außerhalb der Kontrolle des GIL ausführen (siehe auch das Ende des Abschnitts *Kontext mit with-Anweisung*).

8.2 Parallelverarbeitung in Python

Die Verwendung von parallelen Prozessen in Python wollen wir anhand eines konkreten Beispiels diskutieren, nämlich der Berechnung der Mandelbrotmenge, die in einer graphischen Darstellung die sogenannten Apfelmännchen ergibt. Die Mandelbrotmenge ist mathematisch als die Menge der komplexen Zahlen c definiert, für die die durch die Iterationsvorschrift

$$z_{n+1} = z_n^2 + c$$

gegebene Reihe mit dem Anfangselement $z_0 = 0$ beschränkt bleibt. Da bekannt ist, dass die Reihe nicht beschränkt ist, wenn $|z| > 2$ erreicht wird, genügt es, die Iteration bis zu diesem Schwellwert durchzuführen. Die graphische Darstellung wird dann besonders ansprechend, wenn man die Punkte außerhalb der Mandelbrotmenge farblich in Abhängigkeit von der Zahl der Iterationsschritte darstellt, die bis zum Überschreiten des Schwellwerts von 2 erforderlich waren. Da die Iterationen für verschiedene Werte von c vollkommen unabhängig voneinander sind, ist

[1] Wenn wir hier von Python sprechen, meinen wir immer die CPython-Implementation. Eine Implementation von Python ohne GIL ist zum Beispiel das in Java geschriebene Jython.

[2] Cython sollte nicht mit CPython verwechselt werden.

dieses Problem *embarrassingly parallel* und man kann sehr leicht verschiedenen Prozessen unterschiedliche Werte von c zur Bearbeitung zuordnen. Am Ende muss man dann lediglich alle Ergebnisse einsammeln und graphisch darstellen.

Wir beginnen zunächst mit einer einfachen Ausgangsversion eines Programms zur Berechnung der Mandelbrotmenge.

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

def mandelbrot_iteration(c, nitermax):
    z = 0
    for n in range(nitermax):
        if abs(z) > 2:
            return n
        z = z**2+c
    return nitermax

def mandelbrot(xmin, xmax, ymin, ymax, npts, nitermax):
    data = np.empty(shape=(npts, npts), dtype=np.int)
    dx = (xmax-xmin)/(npts-1)
    dy = (ymax-ymin)/(npts-1)
    for nx in range(npts):
        x = xmin+nx*dx
        for ny in range(npts):
            y = ymin+ny*dy
            data[ny, nx] = mandelbrot_iteration(x+1j*y, nitermax)
    return data

def plot(data):
    plt.imshow(data, extent=(xmin, xmax, ymin, ymax),
               cmap='jet', origin='bottom', interpolation='none')
    plt.show()

nitermax = 2000
npts = 1024
xmin = -2
xmax = 1
ymin = -1.5
ymax = 1.5
data = mandelbrot(xmin, xmax, ymin, ymax, npts, nitermax)
# plot(data)
```

Dabei erfolgt die Auswertung der Iterationsvorschrift in der Funktion `mandelbrot_iteration` und die Funktion `mandelbrot` dient dazu, alle Punkte durchzugehen und die Ergebnisse im Array `data` zu sammeln. Bei der weiteren Überarbeitung ist die Funktion `plot` nützlich, um die korrekte Funktionsweise des Programms auf einfache Weise testen zu können. Für die Bestimmung der Rechenzeit mit Hilfe des `cProfile`-Moduls kommentieren wir den Aufruf der `plot`-Funktion jedoch aus. Die Verwendung von `cProfile` ist im Kapitel *Das Modul cProfile* beschrieben. Im Folgenden sind die wesentlichen Beiträge zur Rechenzeit für zwei verschiedene Prozessoren gezeigt, nämlich einen i7-3770:

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  1048576  306.599    0.000  528.491    0.001 m1.py:4(mandelbrot_iteration)
357051172  221.893    0.000  221.893    0.000 {built-in method builtins.abs}
        1    1.892    1.892  530.383  530.383 m1.py:12(mandelbrot)
```

und einen i5-4690:

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  1048576   95.877    0.000  114.408    0.000 m1.py:4(mandelbrot_iteration)
357051172   18.530    0.000   18.530    0.000 {built-in method builtins.abs}
        1    0.424    0.424  114.832  114.832 m1.py:12(mandelbrot)
```

Es zeigen sich deutliche Unterschiede, wobei aber in beiden Fällen die Berechnung des Absolutbetrags der komplexen Variable `z` für einen wesentlichen Beitrag zur Rechenzeit verantwortlich ist. Besonders deutlich ist dies im ersten Fall, wo dieser Beitrag über 40 Prozent ausmacht. Allerdings ist bei der Interpretation dieses Ergebnisses etwas Vorsicht geboten, da die Berechnung des Absolutbetrags typischerweise lediglich einige zehn Nanosekunden benötigt, hier aber extrem oft aufgerufen wird. In einer solchen Situation verursacht `cProfile` einen erheblichen Zusatzaufwand, der sich in den obigen Daten niederschlägt und auch deutlich wird, wenn man die reine Rechenzeit als Differenz von End- und Startzeit bestimmt. Diese liegt für beide Prozessortypen bei etwas 85 Sekunden.

Obwohl also die Berechnung des Absolutbetrags für die Rechenzeit nicht so relevant ist, wie es zunächst den Anschein hat, ist es für spätere Programmversionen sinnvoll, eine reelle Variante der Mandelbrot-Iteration zu implementieren. Damit verfolgen wir die Strategie, bereits vor der Parallelisierung des Programms den Code möglichst stark zu optimieren, um anschließend durch die Parallelisierung einen weitere Beschleunigung des Programms zu erzielen.

Die Berechnung des Absolutbetrags lässt sich vermeiden, wenn man nicht mit einer komplexen Variable rechnet, sondern Real- und Imaginärteil separat behandelt, wie die folgende Version der Funktionen `mandelbrot_iteration` und `mandelbrot` zeigt.

```
def mandelbrot_iteration(cx, cy, nitermax):
    x = 0
    y = 0
    for n in range(nitermax):
        x2 = x*x
        y2 = y*y
        if x2+y2 > 4:
            return n
        x, y = x2-y2+cx, 2*x*y+cy
    return nitermax

def mandelbrot(xmin, xmax, ymin, ymax, npts, nitermax):
    data = np.empty(shape=(npts, npts), dtype=np.int)
    dx = (xmax-xmin)/(npts-1)
    dy = (ymax-ymin)/(npts-1)
    for nx in range(npts):
        x = xmin+nx*dx
        for ny in range(npts):
            y = ymin+ny*dy
            data[ny, nx] = mandelbrot_iteration(x, y, nitermax)
    return data
```

Durch diese Umschreibung verkürzt sich die Rechenzeit insbesondere für den i7-3770-Prozessor drastisch:

```
 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1048576  121.770    0.000  121.770    0.000 m2.py:4(mandelbrot_iteration)
      1    1.984    1.984  123.754  123.754 m2.py:15(mandelbrot)
```

Aber auch für den i5-4690-Prozessor ergibt sich eine Verkürzung der Rechenzeit:

```
 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1048576   85.981    0.000   85.981    0.000 m2.py:4(mandelbrot_iteration)
      1    0.330    0.330   86.312   86.312 m2.py:15(mandelbrot
```

Tatsächlich wird diese Verkürzung vor allem durch die Verringerung des durch `cProfile` bedingten Zusatzaufwands verursacht. Die tatsächliche Rechenzeit kann durch unsere Änderung sogar größer werden. Dennoch ist die Verwendung der reellen Variante in den folgenden Programmversionen günstiger.

Man kann nun vermuten, dass sich die Rechenzeit mit Hilfe von NumPy verringern lässt. In diesem Fall ist eine separate Behandlung der Iteration nicht mehr sinnvoll, so dass wir statt der Funktionen `mandelbrot_iteration` und `mandelbrot` nur noch eine Funktion `mandelbrot` haben, die folgendermaßen aussieht.

```
def mandelbrot(xmin, xmax, ymin, ymax, npts, nitermax):
    cy, cx = np.mgrid[ymax:ymin:npts*1j, xmin:xmax:npts*1j]
```

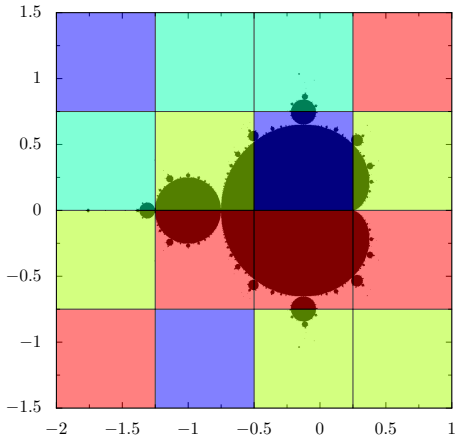


Abb. 8.1: Bearbeitung der einzelnen Teilbereiche zur Berechnung der Mandelbrotmenge durch vier Prozesse, die durch unterschiedliche Farben gekennzeichnet sind.

```
3       x = np.zeros_like(cx)
4       y = np.zeros_like(cx)
5       data = np.zeros(cx.shape, dtype=np.int)
6       for n in range(nitermax):
7           x2 = x*x
8           y2 = y*y
9           notdone = x2+y2 < 4
10          data[notdone] = n
11          x[notdone], y[notdone] = (x2[notdone]-y2[notdone]+cx[notdone],
12                                    2*x[notdone]*y[notdone]+cy[notdone])
13      return data
```

Hierbei benutzen wir *fancy indexing*, da nicht alle Elemente des Arrays bis zum Ende iteriert werden müssen. Es ergibt sich nochmal eine signifikante Reduktion der Rechenzeit. Der i7-3770-Prozessor mit

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1   21.066   21.066   21.088   21.088 m3.py:4(mandelbrot)
```

und der i5-4690-Prozessor mit

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1   20.173   20.173   20.191   20.191 m3.py:4(mandelbrot)
```

unterscheiden sich kaum noch in der benötigten Rechenzeit. Bereits ohne Parallelisierung haben wir durch NumPy mindestens einen Faktor 4 gewonnen.

Nun können wir daran gehen, die Berechnung dadurch weiter zu beschleunigen, dass wir die Aufgabe in mehrere Teilaufgaben aufteilen und verschiedenen Prozessen zur parallelen Bearbeitung übergeben. Seit Python 3.2 stellt die Python-Standardbibliothek hierfür das `concurrent.futures`-Modul zur Verfügung. Der Name `concurrent` deutet hier auf das gleichzeitige Abarbeiten von Aufgaben hin, während sich `futures` auf Objekte beziehen, die zu einem späteren Zeitpunkt das gewünschte Resultat bereitstellen.

Um eine parallele Bearbeitung der Mandelbrotmenge zu ermöglichen, teilen wir den gesamten Wertebereich der zu betrachtenden komplexen Zahlen c in eine Anzahl von Kacheln auf, die von den einzelnen Prozessen bearbeitet werden. Abb. 8.1 zeigt, wie 16 Kacheln von vier Prozessen abgearbeitet wurden, wobei jeder Prozess durch eine eigene Frage dargestellt ist. In diesem speziellen Lauf haben zwei Prozesse nur drei Kacheln bearbeitet, während die beiden anderen Prozesse fünf Kacheln bearbeitet haben.

Im Folgenden sind die wesentlichen Codeteile dargestellt, die für die parallele Berechnung der Mandelbrotmenge benötigen.

Quellcode 8.1: Wesentliche Teile eines Programms zur Berechnung der Mandelbrotmenge unter Verwendung von parallelen Prozessen

```
1  from concurrent import futures
2  from itertools import product
3  from functools import partial
4  import time
5
6  import numpy as np
7
8  def mandelbrot_tile(nitermax, nx, ny, cx, cy):
9      x = np.zeros_like(cx)
10     y = np.zeros_like(cx)
11     data = np.zeros(cx.shape, dtype=np.int)
12     for n in range(nitermax):
13         x2 = x*x
14         y2 = y*y
15         notdone = x2+y2 < 4
16         data[notdone] = n
17         x[notdone], y[notdone] = (x2[notdone]-y2[notdone]+cx[notdone],
18                                   2*x[notdone]*y[notdone]+cy[notdone])
19     return (nx, ny, data)
20
21 def mandelbrot(xmin, xmax, ymin, ymax, npts, nitermax, ndiv, max_workers=4):
22     start = time.time()
23     cy, cx = np.mgrid[ymin:ymax:npts*1j, xmin:xmax:npts*1j]
24     nlen = npts//ndiv
25     paramlist = [(nx, ny,
26                   cx[nx*nlen:(nx+1)*nlen, ny*nlen:(ny+1)*nlen],
27                   cy[nx*nlen:(nx+1)*nlen, ny*nlen:(ny+1)*nlen])
28                  for nx, ny in product(range(ndiv), repeat=2)]
29     with futures.ProcessPoolExecutor(max_workers=max_workers) as executors:
30         wait_for = [executors.submit(partial(mandelbrot_tile, nitermax),
31                                              nx, ny, cx, cy)
32                     for (nx, ny, cx, cy) in paramlist]
33         results = [f.result() for f in futures.as_completed(wait_for)]
34     data = np.zeros(cx.shape, dtype=np.int)
35     for nx, ny, result in results:
36         data[nx*nlen:(nx+1)*nlen, ny*nlen:(ny+1)*nlen] = result
37     return time.time()-start, data
```

Die Funktion `mandelbrot_tile` ist eine leichte Anpassung der zuvor besprochenen Funktion `mandelbrot`. Der wesentliche Unterschied besteht darin, dass in der vorigen Version das NumPy-Array für die Variable c in der Funktion selbst erzeugt wurde. Nun werden zwei Arrays mit Real- und Imaginärteil explizit übergeben. Neu ist die Funktion `mandelbrot` in den Zeilen 21 bis 37. Neben den Grenzen `xmin`, `xmax`, `ymin` und `ymax` der zu betrachtenden Region, der Zahl der Punkte `npts` je Dimension und der maximalen Zahl von Iterationen `nitermax` gibt es noch zwei weitere Variablen. `ndiv` gibt die Zahl der Unterteilungen je Dimension der Gesamtregion an. Ein Wert von 4 entspricht den 16 Bereichen in der vorigen Abbildung. Die maximale Anzahl von parallelen Prozessen ist durch `max_workers` gegeben, das wir defaultmäßig auf den Wert 4 setzen, weil wir von einem Prozessor mit vier Kernen ausgehen.

Da wir die Laufzeit bei mehreren parallelen Prozessen nicht mit dem `cProfile`-Modul bestimmen können, halten wir in Zeile 22 die Startzeit fest und berechnen in Zeile 37 die Laufzeit. Für die Parallelverarbeitung benötigen wir nun zunächst eine Liste von Aufgaben, die durch entsprechende Parameter spezifiziert sind. Dazu werden in Zeile 23 zwei zweidimensionale Arrays angelegt, die das Gitter der komplexen Zahlen c definieren. Außerdem wird in Zeile 24 die Seitenlänge der Unterbereiche bestimmt. Damit kann nun in den Zeilen 25–28 die Parameterliste erzeugt werden. Hierzu gehen wir mit Hilfe von `product` aus dem in Zeile 2 importierten `itertools`-Modul durch alle Indexpaare `(nx, ny)` der Unterbereiche. Die Parameterliste enthält diese Indizes, die wir später wieder benötigen, um das Resultat zusammenzusetzen, sowie die beiden Arrays mit den zugehörigen Werten des Real- und Imaginärteils von c.

Der zentrale Teil folgt nun in den Zeilen 29 bis 33, wo wir in diesem Fall einen Kontext-Manager verwenden.

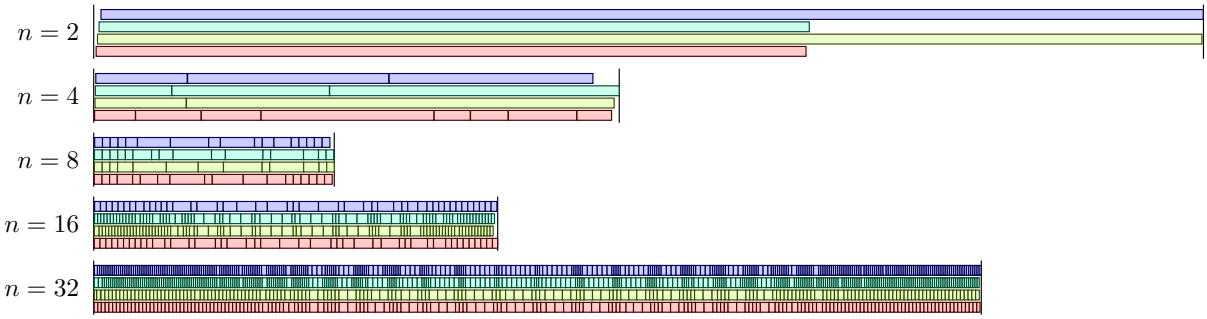


Abb. 8.2: Verteilung der Teilaufgaben für die Berechnung der Mandelbrotmenge auf vier Prozesse in Abhängigkeit von der Anzahl der Segmente je Achse.

Dieses Konzept hatten wir im Abschnitt *Kontext mit with-Anweisung* eingeführt. Es wird ein Pool von Prozessen angelegt, der die Aufgaben ausführen wird, die in Zeile 30 mit Hilfe der zuvor erstellten Parameterliste eingereicht werden. Da die `submit`-Methode als Argumente eine Funktion sowie deren Argumente erwartet, haben wir hier mit Hilfe des in Zeile 3 importierten `functools`-Moduls eine partielle Funktion definiert, deren erstes Argument, also `nitermax`, bereits angegeben ist.

Die Aufgaben werden nun, ohne dass wir uns darum weiter kümmern müssen, nacheinander von den Prozessen abgearbeitet. Wann dieser gesamte Vorgang abgeschlossen sein wird, ist nicht vorhersagbar. Daher wird in Zeile 33 mit Hilfe der Funktion `futures.as_completed` abgewartet, bis alle Aufgaben erledigt sind. Die Resultate werden in einer Liste gesammelt. Es bleibt nun nur noch, in den Zeilen 34 bis 36 das Ergebnis in einem einzigen Array zusammenzufassen, um es zum Beispiel anschließend graphisch darzustellen.

Es zeigt sich, dass auf den getesteten Prozessoren eine minimale Rechenzeit für die Mandelbrotmenge erreicht wird, wenn das zu behandelnde Gebiet in 64 Teilgebiete unterteilt wird, also sowohl die reelle als auch die imaginäre Achse in acht Segmente unterteilt wird. Dann benötigt ein i7-3770-Prozessor noch etwa 4,4 Sekunden, während ein i5-4690-Prozessor 3 Sekunden benötigt. Damit ergibt sich eine Beschleunigung um einen Faktor von 20 bis 30.

Interessant ist, wie die zeitliche Verteilung der Aufgaben auf die vier Prozesse erfolgt. Dies ist in Abb. 8.2 für verschiedene Unterteilungen der Achsen zu sehen. Hat man nur vier Aufgaben für vier Prozesse zur Verfügung, so ist die Rechenzeit durch die am längsten laufende Aufgabe bestimmt. Gleichzeitig sieht man bei $n = 2$, dass der Start des Prozesses bei dem in diesem Fall relativ hohen Speicherbedarf zu einer merklichen Verzögerung führt. Ganz grundsätzlich ist der Kommunikationsbedarf beim Starten und Beenden einer Aufgabe in einem Prozess mit einem gewissen Zeitbedarf verbunden. Insofern ist zu erwarten, dass sich zu viele kleine Aufgaben negativ auf die Rechenzeit auswirken. Für $n = 4$ und $n = 8$ beobachten wir aber zunächst eine Verkürzung der Rechenzeit. Dies hängt zum einen damit zusammen, dass jeder Prozess letztlich ähnlich lange für die Abarbeitung seiner Aufgaben benötigt. Bei $n = 4$ ist deutlich zu sehen, dass sich die Anzahl der bearbeiteten Aufgaben von Prozess zu Prozess erheblich unterscheiden kann.

Außerdem wird die Rechenzeit unter Umständen wesentlich durch den Umfang der in einem Prozess zu bearbeitenden Daten bestimmt. Dies hängt damit zusammen, dass die Versorgung des Prozessors mit Daten einen erheblichen Engpass darstellen kann. Aus diesem Grund werden zwischen dem Hauptspeicher und dem Prozessor so genannte Caches implementiert, die einen schnelleren Datenzugriff erlauben, die jedoch in ihrer Größe begrenzt sind. Daher kann es für die Rechengeschwindigkeit förderlich sein, die für die individuelle Aufgabe erforderliche Datenmenge nicht zu groß werden zu lassen.

Dies wird an Abb. 8.3 deutlich. Betrachten wir zunächst die gestrichelten Kurven, bei denen die Rechenzeit für die Gesamtaufgabe, also für $n = 1$, durch die Rechenzeit für die Parallelverarbeitung für verschiedene Werte von n geteilt wurde. Obwohl nur vier Prozesse verwendet wurden, findet man unter gewissen Bedingungen eine Beschleunigung, die über dem Vierfachen liegt. Bei den zugehörigen Aufgabengrößen können die Caches offenbar sehr gut genutzt werden. Eine vom Verhalten der Caches unabhängige Einschätzung des Einflusses der Parallelisierung erhält man durch Vergleich der parallelen Abarbeitung der Teilaufgaben mit der sequentiellen Abarbeitung der gleichen Teilaufgaben. Die zugehörige Beschleunigung ist durch die durchgezogenen Kurven dargestellt. Hier zeigt sich, dass ein Verhältnis von vier entsprechend der vier Prozesse nahezu erreicht werden kann, wenn die Größe der Teilaufgaben nicht zu groß gewählt ist.

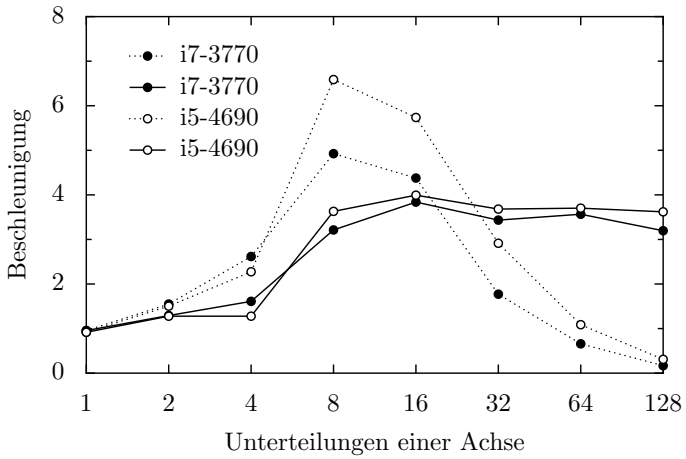


Abb. 8.3: Die Beschleunigung durch Parallelisierung bei der Berechnung der Mandelbrotmenge ist für die zwei Prozessortypen i7-3770 (schwarze Punkte) und i5-4690 (weiße Punkte) als Funktion der Unterteilung der Achsen dargestellt. Die gestrichelten Kurven zeigen das Verhältnis der Rechenzeit für das Gesamtproblem zur Rechenzeit für die parallelisierte Variante, während die durchgezogenen Kurven das Verhältnis der Rechenzeit für die sequentielle Abarbeitung der Teilaufgaben zur Rechenzeit für die parallele Abarbeitung zeigen.

8.3 Numba

Im vorigen Abschnitt haben wir gesehen, wie man mit Hilfe von NumPy und durch Parallelisierung ein Programm beschleunigen kann. Dies ging in dem Beispiel der Mandelbrotmenge relativ einfach, da natürlicherweise Arrays verwendet werden konnten und zudem die Behandlung der einzelnen Teilprobleme keine Kommunikation untereinander erforderte. Neben NumPy und der Parallelisierung gibt es noch andere Optionen, um Code zu beschleunigen, die sich zum Teil aktuell sehr intensiv weiterentwickelt werden, so dass sich die Einsatzmöglichkeiten unter Umständen zukünftig schnell erweitern können. Daher soll in diesem Abschnitt auch nur ein Eindruck von anderen Möglichkeiten gegeben werden, ein Programm zu beschleunigen.

Wir greifen hier speziell Numba[3] heraus, da es unter anderem für das numerische Arbeiten im Zusammenhang mit NumPy konzipiert ist und auch Parallelverarbeitung unterstützt. Zentral für Numba ist die sogenannte *Just in Time* (JIT) Kompilierung. Hierbei werden Funktionen in ausführbaren Code übersetzt, der anschließend schneller ausgeführt werden kann als dies der Python-Intepreter tun würde. Während in Python der Datentyp der Funktionsargumente nicht spezifiziert ist, sieht sich Numba beim Funktionsaufruf die tatsächlich verwendeten Datentypen an und erzeugt entsprechenden ausführbaren Code. Bei nächsten Aufruf mit der gleichen Signatur, also mit den gleichen Datentypen der Argumente, kann auf diesen Code zurückgegriffen werden. Andernfalls wird bei Bedarf eine andere Version des ausführbaren Codes erstellt.

Wir wollen dies an einem einfachen Beispiel illustrieren, in dem näherungsweise die riemannsche Zetafunktion

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}$$

berechnet wird. Der im folgenden Code implementierte Algorithmus ist nicht optimal für die Berechnung der Zetafunktion, aber dies ist für unser Beispiel nicht relevant. Ohne Verwendung von Numba könnte unser Code wie folgt aussehen:

```
1 import time
2
3 def zeta(x, nmax):
4     summe = 0
5     for n in range(1, nmax+1):
6         summe = summe+1/(n**x)
7     return summe
8
```

[3] Für weitere Informationen siehe die jeweils aktuelle Dokumentation.

```
nmax = 100000000
start = time.time()
print(zeta(2, nmax))
print('Zeit:', time.time()-start)
```

Da hier die Summe nur über endlich viele Terme ausgeführt wird sei erwähnt, dass $\zeta(2) = \pi^2/6$.

Für die Verwendung mit Numba müssen wir lediglich Numba importieren (Zeile 2) und die Funktion mit einem Dekorator (Zeile 4) versehen:

```
import time
import numba

@numba.jit
def zeta(x, nmax):
    summe = 0
    for n in range(1, nmax+1):
        summe = summe+1/(n**x)
    return summe

nmax = 100000000
start = time.time()
print(zeta(2, nmax))
print('Zeit:', time.time()-start)
```

Vergleichen wir die beiden Laufzeiten, so erhalten wir auf dem gleichen Rechner im ersten Fall etwa 33,4 Sekunden, im zweiten Fall dagegen nur 0,6 Sekunden. Wir können uns am Ende dieses Codes anzeigen lassen, welche Signatur von Numba kompiliert wurde, indem wir die folgende Zeile anhängen:

```
print(zeta.signatures)
```

Das Ergebnis lautet:

```
[(int64, int64)]
```

Diese Liste von Signaturen enthält nur einen Eintrag, da wir die Funktion `zeta` mit zwei Integer-Argumenten aufgerufen haben. Wie in NumPy können Integers hier nicht beliebig lang werden, sondern sind in diesem Beispiel 8 Bytes lang. Es besteht also die Gefahr des Überlaufs. So kommt es in unserem Beispiel zur einer Division durch Null, wenn man die Variable `x` auf den Wert :math:`3` setzt. Bereits vor der Division durch Null wird aufgrund des Überlaufs durch negative Zahlen dividiert, so dass die Summe unsinnige Werte liefert. Die Gefahr des Überlaufs muss also bedacht werden.

Übergibt man auch Gleitkomma- oder komplexe Zahlen für das Argument `x`, so muss Numba für diese neuen Signaturen eine Kompilation durchführen. Der Code

```
import time
import numba

@numba.jit
def zeta(x, nmax):
    summe = 0
    for n in range(1, nmax+1):
        summe = summe+1/(n**x)
    return summe

nmax = 100000000
for x in (2, 2.5, 2+1j):
    start = time.time()
    print('ζ({}) = {}'.format(x, zeta(x, nmax)))
    print('Zeit: {:5.2f}s\n'.format(time.time()-start))

print(zeta.signatures)
```

liefert die Ausgabe:

```
ζ(2) = 1.644934057834575
Zeit:  0.59s

ζ(2.5) = 1.341487257103954
Zeit:  5.52s

ζ((2+1j)) = (1.1503556987382961-0.43753086346605924j)
Zeit: 13.41s

[(int64, int64), (float64, int64), (complex128, int64)]
```

Wir sehen zum einen, dass die Rechendauer vom Datentyp der Variable `x` abhängt, und zum anderen, dass die Kompilierung in der Tat für drei verschiedene Signaturen durchgeführt wurde.

Mit Hilfe von Numba können wir zudem Funktionen leicht in universelle Funktionen, also *ufuncs* umwandeln, die wir in Abschnitt *Universelle Funktionen* im Zusammenhang mit NumPy eingeführt hatten. Universelle Funktionen sind in der Lage, neben skalaren Argumenten auch Arrays als Argumente zu verarbeiten. Dies erlaubt bereits die Verwendung des Dekorators `jit`. Mit Hilfe des Dekorators `vectorize` kann zudem erreicht werden, dass die Funktionsauswertung für die Werte des Arrays in mehreren Threads parallel ausgeführt wird.

Im folgenden Codebeispiel geben wir als Argumente für den Dekorator die Signatur an, die Numba verwenden soll. Das Argument `x` hat den Datentyp `float64` und kann auch ein entsprechendes Array sein. Das Argument `n` ist vom Datentyp `int64`. Der Datentyp des Resultats ist wiederum `float64` und steht als erstes in der Signatur vor dem Klammerpaar, das die Datentypen der Argumenten enthält. Das Argument `target` bekommt hier den Wert `'parallel'`, um für ein Array die Parallelverarbeitung in mehreren Threads zu erlauben. Wird eine Parallelverarbeitung nicht gewünscht, zum Beispiel weil das Problem zu klein ist und das Starten eines Threads nur unnötig Zeit kosten würde, so kann man auch `target='cpu'` setzen. Hat man einen geeigneten Grafikprozessor, so kann dieser mit `target='cuda'` zur Rechnung herangezogen werden.

Quellcode 8.2: Die Erzeugung einer universellen Funktion mit Hilfe des `vectorize`-Dekorators von Numba wird am Beispiel der Auswertung der Zetafunktion demonstriert.

```
import time
import numpy as np
from numba import vectorize, float64, int64

@vectorize([float64(float64, int64)], target='parallel')
def zeta(x, nmax):
    summe = 0.
    for n in range(nmax):
        summe = summe+1./((n+1)**x)
    return summe

x = np.linspace(2, 10, 200, dtype=np.float64)
start = time.time()
y = zeta(x, 10000000)
print(time.time()-start)
```

In Abb. 8.4 ist die Beschleunigung des Programms als Funktion der verwendeten Threads für einen i7-3770-Prozessor gezeigt, der vier Rechenkerne besitzt, auf dem aber durch sogenanntes Hyperthreading acht Threads parallel laufen können. Bei Verwendung von bis zur vier Threads steigt die Beschleunigung fast wie die Zahl der Threads an, während die Beschleunigung darüber merklich langsamer ansteigt. Dies hängt damit zusammen, dass dann Threads häufiger auf freie Ressourcen warten müssen.

In Numba lassen sich universelle Funktionen mit Hilfe des Dekorators `guvectorize` noch verallgemeinern, so dass in der inneren Schleife auch Arrays verwendet werden können. Bei den üblichen universellen Funktionen wird in der inneren Schleife dagegen mit Skalaren gearbeitet. Um dies an einem Beispiel zu verdeutlichen, kommen wir auf das Mandelbrotbeispiel zurück.

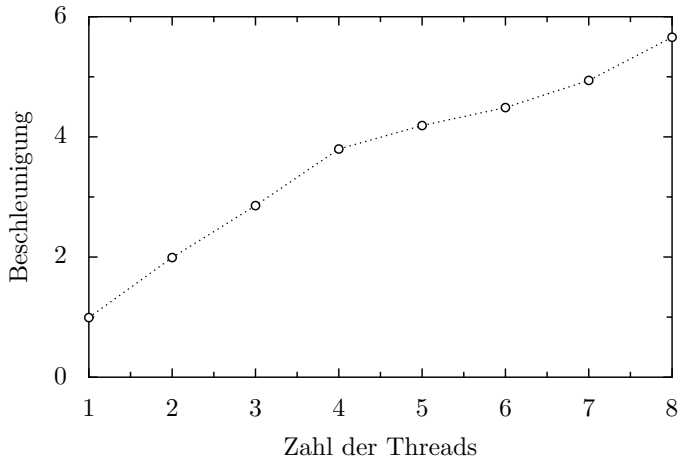


Abb. 8.4: Beschleunigung der Rechengeschwindigkeit für die Berechnung der Zetafunktion mit dem Quellcode 8.2 als Funktion der Anzahl der Threads auf einem Vierkernprozessor mit Hyperthreading.

```
import time

from numba import jit, guvectorize, complex128, int64
import matplotlib.pyplot as plt
import numpy as np

@jit
def mandelbrot_iteration(c, maxiter):
    z = 0
    for n in range(maxiter):
        z = z**2+c
        if z.real*z.real+z.imag*z.imag > 4:
            return n
    return maxiter

@guvectorize([(complex128[:], int64[:], int64[:])], '(n), () -> (n)',
             target='parallel')
def mandelbrot(c, itermax, output):
    nitermax = itermax[0]
    for i in range(c.shape[0]):
        output[i] = mandelbrot_iteration(c[i], nitermax)

def mandelbrot_set(xmin, xmax, ymin, ymax, npts, nitermax):
    cy, cx = np.ogrid[ymin:ymax:npts*1j, xmin:xmax:npts*1j]
    c = cx+cy*1j
    return mandelbrot(c, nitermax)

def plot(data, xmin, xmax, ymin, ymax):
    plt.imshow(data, extent=(xmin, xmax, ymin, ymax),
               cmap='jet', origin='bottom', interpolation='none')
    plt.show()

nitermax = 2000
npts = 1024
xmin = -2
xmax = 1
ymin = -1.5
ymax = 1.5
start = time.time()
data = mandelbrot_set(xmin, xmax, ymin, ymax, npts, nitermax)
ende = time.time()
print(ende-start)
```

```
43 plot(data, xmin, xmax, ymin, ymax)
```

Unser besonderes Augenmerk richten wir hier auf die Funktion `mandelbrot`, die mit dem `guvectorize`-Dekorator versehen ist und einige Besonderheiten aufweist. Die Funktion `mandelbrot` besitzt drei Argumente, von denen hier zwei, nämlich `c` und `itermax`, an die Funktion übergeben werden, während das dritte Argument, also `output` für die Rückgabe des Ergebnisses vorgesehen ist. Dies kann man dem zweiten Argument des Dekorators, dem sogenannten Layout, entnehmen. Diesem kann man entnehmen, dass das zurückgegebene Array `output` die gleiche Form wie das Argument `c` besitzt. Da wir ein zweidimensionales Array `c` übergeben, ist das Argument `c[i]` der Funktion `mandelbrot_iteration` selbst wieder ein Array. Andererseits muss man bedenken, dass das Argument `itermax` ein Array ist, so dass hier zur Verwendung als Skalar das Element `0` herangezogen wird.

Auf einem i7-3770-Prozessor, der durch Hyperthreading bis zu acht Threads unterstützt, wird dieses Programm in knapp 0,48 Sekunden ausgeführt. Wir erreichen somit eine Beschleunigung gegenüber unserem bisher schnellsten Quellcode 8.1 um fast eine Größenordnung. Gegenüber unserer allerersten Version haben wird auf diesem Prozessortyp sogar eine Beschleunigung um einen Faktor von fast 200 erreicht.