Das Python3.3-Tutorial auf Deutsch

Release 3.3

Michael Markert et al

Oktober 08, 2017

Inhaltsverzeichnis

Release 3.3

Date Oktober 08, 2017

Python ist eine einfach zu lernende, aber mächtige Programmiersprache mit effizienten abstrakten Datenstrukturen und einem einfachen, aber effektiven Ansatz zur objektorientierten Programmierung. Durch die elegante Syntax und die dynamische Typisierung ist Python als interpretierte Sprache sowohl für Skripte als auch für schnelle Anwendungsentwicklung (Rapid Application Development) hervorragend geeignet.

Der Python-Interpreter und die umfangreiche Standardbibliothek sind als Quelltext und in binärer Form für alle wichtigen Plattformen auf der Webseite http://www.python.org frei verfügbar, und können frei weiterverbreitet werden. Auf der gleichen Seite finden sich Distributionen von Drittanbietern, Verweise auf weitere freie Module, Programme und Werkzeuge, sowie zusätzliche Dokumentation.

Der Python-Interpreter kann auf einfache Weise um neue Funktionen und Datentypen erweitert werden, die in C oder C++ (oder andere Sprachen, die sich von C aus ausführen lassen) implementiert sind. Auch als Erweiterungssprache für anpassbare Applikationen ist Python hervorragend geeignet.

Dieses Tutorial stellt die Grundkonzepte und Eigenschaften der Sprache und des Systems Python vor. Zwar ist es hilfreich, einen Python-Interpreter griffbereit zu haben, um praktische Erfahrungen zu sammeln, aber alle Beispiele sind eigenständig, so dass das Tutorial auch offline gelesen werden kann.

Eine Beschreibung der Standardobjekte und -module ist in der Referenz der Pythonbibliothek (Python Library Reference) zu finden. Das Python Referenzhandbuch (Python Reference Manual) bietet eine formellere Definition der Sprache. Um Erweiterungen in C oder C++ zu schreiben, sollte man sich Extending and Embedding the Python Interpreter sowie die Python/C API Reference anschauen. Es gibt auch zahlreiche Bücher, die Python tiefergehend behandeln.

Ziel dieses Tutorials ist nicht die umfangreiche und vollständige Behandlung sämtlicher Sprachmerkmale von Python, sondern die Vermittlung der wichtigsten Eigenschaften, um einen Eindruck von dem zu bekommen, was Python ist. Nach der Lektüre sollte man in der Lage sein, Python-Module und -Programme zu schreiben. Außerdem verfügt man dann über die notwendigen Grundlagen, um sich mit weiteren Modulen der Standardbibliothek zu beschäftigen, die in der Python Library Reference beschrieben werden.

Es lohnt sich ebenfalls das *Glossar* durchzugehen.

KAPITEL 1

Um dich auf den Geschmack zu bringen

Wer viel am Computer arbeitet, kommt irgendwann zu dem Schluss, dass es Aufgaben gibt, die er gern automatisieren würde. Beispielsweise ein Suchen-und-Ersetzen für eine Vielzahl von Dateien oder eine Möglichkeit, einen Haufen Fotodateien auf komplizierte Art umzubenennen oder umzuräumen. Oder man hätte gerne eine kleine Datenbank nach Maß, eine spezialisierte GUI-Anwendung oder ein einfaches Spiel.

Als professioneller Softwareentwickler muss man vielleicht mit mehreren C/C++/Java-Bibliotheken arbeiten, findet aber den üblichen Schreiben/Kompilieren/Testen/Re-Kompilieren-Zyklus zu langsam. Wer eine Testsuite für solch eine Bibliothek schreibt, hält es vielleicht für eine ermüdende Aufgabe, den Testcode zu schreiben. Vielleicht hat der ein oder andere auch ein Programm geschrieben, das eine Erweiterungssprache gebrauchen könnte, will aber keine ganz neue Sprache für sein Programm entwerfen und implementieren.

Dann ist Python genau die richtige Sprache!

Man könnte natürlich Unix-Shellskripte oder Windows-Batchdateien für ein paar dieser Aufgaben schreiben. Mit Shellskripten lassen sich gut Dateien verschieben und Textdaten verändern, zur Entwicklung von GUI-Applikationen oder Spielen sind sie aber weniger geeignet. Man könnte ein entsprechendes C/C++/Java-Programm dafür schreiben, aber es kostet in der Regel bereits viel Entwicklungszeit, um überhaupt einen ersten Programmentwurf zu entwickeln. Python ist einfacher zu nutzen, verfügbar für Windows-, Mac OS X- und Unix-Betriebssysteme und hilft, die Aufgabe schneller zu erledigen.

Python ist einfach in der Anwendung, aber eine echte Programmiersprache, die viel mehr Struktur und Unterstützung für große Programme bietet, als Shellskripte oder Batchdateien es könnten. Auf der anderen Seite bietet Python auch mehr Fehlerüberprüfungen als C und hat, als stark abstrahierende Hochsprache, mehr abstrakte Datentypen wie flexible Arrays und Wörterbücher (Dictionaries) eingebaut. Aufgrund seiner allgemeineren Datentypen ist Python in ausgedehnteren Problembereichen einsetzbar als Awk oder sogar Perl, und dennoch sind viele Dinge in Python mindestens so einfach wie in diesen Sprachen.

Python erlaubt die Aufteilung von Programmen in Module, die in anderen Python-Programmen wiederverwendet werden können. Es kommt mit einer großen Sammlung von Standardmodulen, die als Grundlage für eigene Programme genutzt werden können; oder als Beispiele, um in Python Programmieren zu lernen. Manche der Module stellen Datei-I/O, Systemaufrufe, Sockets und sogar Schnittstellen zu GUI-Toolkits wie Tk bereit.

Python ist eine interpretierte Sprache, wodurch sich bei der Programmentwicklung erheblich Zeit sparen lässt, da Kompilieren und Linken nicht nötig sind. Der Interpreter kann interaktiv genutzt werden, so dass man einfach mit

den Fähigkeiten der Sprache experimentieren, Wegwerf-Code schreiben oder Funktionen während der Bottom-Up-Programmentwicklung testen kann. Es ist auch ein praktischer Tischrechner.

Python ermöglicht die Entwicklung von kompakten und lesbaren Programmen. Programme, die in Python geschrieben sind, sind aus mehreren Gründen viel kürzer als C/C++/Java-Äquivalente:

- Die abstrakten Datentypen erlauben es, komplexe Operationen in einer einzigen Anweisung auszudrücken;
- Anweisungen werden durch Einrückungen und nicht durch öffnende und schließende Klammern gruppiert;
- Variablen- oder Argumentdeklarationen sind nicht nötig.

Python ist *erweiterbar*: Wer in C programmieren kann, kann einfach eine neue eingebaute Funktion oder ein Modul zum Interpreter hinzuzufügen. Entweder um zeitkritische Operationen mit maximaler Geschwindigkeit auszuführen oder um Python-Programme zu Bibliotheken zu linken, die nur in binärer Form (wie beispielsweise herstellerspezifische Grafikbibliotheken) verfügbar sind. Wenn man erst einmal mit Python vertraut ist, kann man den Python-Interpreter zu in C geschriebenen Applikationen linken und Python als Erweiterung oder Kommandosprache für diese Applikation nutzen.

So nebenbei: Die Sprache ist nach der BBC-Sendung “Monty Python’s Flying Circus” benannt und hat nichts mit Reptilien zu tun. Anspielungen auf Monty Python-Sketche in Dokumentation zu benutzen ist nicht nur erlaubt, sondern gern gesehen.

Jetzt, da du nun ganz heiß auf Python bist, wirst du mehr wissen und lernen wollen. Der beste Weg, um eine Sprache zu erlernen, ist ganz sicher der, sie einzusetzen. Darum lädt das Tutorial gerade dazu ein, während des Lesens mit dem Python-Interpreter zu experimentieren.

Im nächsten Kapitel wird der technische Teil der Nutzung des Interpreters erläutert. Das ist eher nüchterne Information, aber wichtig, um die später gezeigten Beispiele ausprobieren zu können.

Der Rest des Tutorials stellt anhand von Beispielen unterschiedliche Möglichkeiten und Sprachelemente von Python vor. Dies beginnt mit einfachen Ausdrücken (Expressions), Anweisungen (Statements) und Datentypen und geht weiter mit Funktionen und Modulen. Danach werden fortgeschrittene Konzepte wie Ausnahmen (Exceptions) und benutzerdefinierte Klassen behandelt.

KAPITEL 2

Verwendung des Python-Interpreters

Aufrufen des Interpreters

Sofern der Python-Interpreter auf einem Rechner installiert ist, findet man ihn normalerweise unter `/usr/local/bin/python/python3.3`. Wenn man `/usr/local/bin` in den Suchpfad der Unix-Shell setzt, kann man den Interpreter aufrufen durch[1]:

```
python3.3
```

Die Auswahl des Installationspfades für den Interpreter ist eine Installationsoption, so dass auch eine Installation an anderer Stelle möglich ist. Das ist mit dem örtlichen Python-Guru oder dem Systemadministrator zu klären. (Eine populäre Alternative ist etwa `/usr/local/python`)

Auf Windows-Rechnern befindet sich die Pythoninstallation meist unter `C:\Python33`, auch wenn man das während des Installationsvorgangs ändern kann. Um dieses Verzeichnis zum Suchpfad hinzuzufügen, kann man folgendes Kommando in die DOS-Eingabeaufforderung eingeben:

```
set path=%path%;C:\python33
```

Durch Eingabe eines End-Of-File-Zeichens (EOF; `Strg-D` unter Unix, `Strg-Z` unter Windows) in der Eingabeaufforderung des Interpreters wird der Interpreter mit dem Rückgabewert Null beendet. Falls er das nicht tut, kann man den Interpreter durch folgende Befehlszeile beenden: `quit()`.

Die Möglichkeiten des Interpreters hinsichtlich des Editierens der Eingabe sind ziemlich beschränkt, lassen sich aber durch Einsatz der GNU-readline-Bibliothek erweitern. Ob diese erweiterten Möglichkeiten verfügbar sind, lässt sich überprüfen, indem man ein `Strg-P` in die Eingabeaufforderung tippt. Wenn es piepst, ist die "readline"-Unterstützung vorhanden. In diesem Fall findet man im Anhang *Interaktive Eingabe-Bearbeitung und Ersetzung des Verlaufs* eine Einführung zu den einzelnen Tasten. Falls kein Piepton zu hören ist oder `^P` erscheint, ist keine "readline"-Unterstützung vorhanden und die einzige Möglichkeit zum Editieren ist die Verwendung der Rücktaste (Backspace), um Zeichen in der aktuellen Eingabezeile zu entfernen.

[1] Unter Unix wird der Python 3.1 Interpreter nicht standardmäßig als ausführbare Datei namens `python` installiert, damit es nicht zu einer Kollision mit einer gleichzeitig installierten Python-2.x-Version kommt.

Grundsätzlich ist der Interpreter ähnlich zu bedienen wie eine Unix-Shell: Wird er mit einem tty-Gerät als Standardeingabe aufgerufen, liest und führt er interaktiv Befehle aus. Wird er mit einem Dateinamen als Argument oder mit einer Datei als Standardeingabe aufgerufen, liest und führt es ein *Skript* von dieser Datei aus.

Eine zweite Möglichkeit zum Starten des Python-Interpreters ist `python -c Befehl [arg] ...`, wodurch die Anweisung(en) in diesem *Befehl* ausgeführt werden, analog zur `-c`-Option der Shell. Da Python-Anweisungen oft Leerzeichen oder sonstige Zeichen enthält, die von der Shell besonders behandelt werden, sollte man den kompletten *Befehl* in einfache Anführungszeichen setzen.

Einige Python-Module sind auch als Skripte nützlich und können mit `python -m Modul [arg] ...` aufgerufen werden. Dadurch wird der Quelltext von *Modul* ausgeführt, so als hätte man den vollständigen Namen in die Kommandozeile eingegeben.

Achtung: Es gibt einen Unterschied zwischen `python Datei` und `python < Datei`! Im zweiten Fall werden Eingabeanfragen des Programms, wie beispielsweise der Aufruf `sys.stdin.read()`, von *Datei* erledigt. Da diese Datei aber schon vom Parser bis zum Ende gelesen wurde, bevor mit der Ausführung begonnen wird, trifft das Programm sofort auf ein End-Of-File. In ersterem Fall passiert das, was man normalerweise erwartet: Die Eingabeanfragen werden durch diejenige Datei oder das Gerät erledigt, die bzw. das als Standardeingabe zur Verfügung steht.

Wenn eine Skriptdatei verwendet wird, ist es oft hilfreich, das Skript auszuführen und danach in den interaktiven Modus zu wechseln. Dies erreicht man durch die Option `-i` vor dem Skript.

Übergabe von Argumenten

Werden dem Interpreter ein Skriptname und zusätzliche Argumente übergeben, dann werden diese in eine Liste von Zeichenketten gewandelt und an die Variable `argv` im `sys` Modul übergeben. Zugriff darauf erhält man mittels `import sys`. Wenn kein Skript und keine Argumente übergeben wurden, dann ist `sys.argv[0]` eine leere Zeichenkette. Wenn der Skriptname als `'-'` angegeben ist (das entspricht der Standardeingabe), dann wird `sys.argv[0]` auf `'-'` gesetzt. Wird `-c` *Befehl* verwendet, dann erhält `sys.argv[0]` den Wert `'-c'`, bei Verwendung von `-m` *Modul* den vollständigen Namen des gefundenen Moduls. Optionen, die nach `-c` *Befehl* oder `-m` *Modul* angegeben werden, werden nicht vom Python-Interpreter verarbeitet, sondern werden als Werte an `sys.argv` übergeben.

Interaktiver Modus

Wenn Befehle von einem tty (in der Regel wird das eine Konsole sein) gelesen werden, spricht man vom *interaktiven Modus* des Interpreters. In diesem Modus wartet der Interpreter mit der *primären Eingabeaufforderung*, die normalerweise aus drei größer-als-Zeichen besteht (>>>), auf Eingaben des Anwenders. Nach Fortsetzungszeilen zeigt der Interpreter die *sekundäre Eingabeaufforderung*, das sind normalerweise drei Punkte (`...`). Außerdem zeigt der Interpreter nach dem Start zunächst einen kurzen Informationstext an, der unter anderem die Versionsnummer des Interpreters und einen Hinweis zum Urheberrecht enthält.

```
$ python3.3
Python 3.3 (py3k, Apr 1 2010, 13:37:42)
[GCC 4.4.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Fortsetzungszeilen ergeben sich, wenn mehrzeilige Konstrukte eingegeben werden, wie zum Beispiel bei der folgenden `if`-Anweisung:

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
```

```
...
Be careful not to fall off!
```

Der Interpreter und seine Umgebung

Fehlerbehandlung

Tritt ein Fehler auf, dann zeigt der Interpreter eine Fehlermeldung mit einem Verlaufsbericht (*Stacktrace*) an. Im interaktiven Modus kehrt er dann zurück zur primären Eingabeaufforderung. Wenn die Eingabe von einer Datei kam, beendet er sich nach der Ausgabe des Fehlerberichts mit einem Rückgabewert ungleich Null. Ausnahmen (Exceptions), die in einem `try-except`-Block verarbeitet werden, gelten in diesem Zusammenhang nicht als Ausnahmen. Manche Fehler führen zum sofortigen Abbruch des Interpreters mit einem Rückgabewert ungleich Null. Dies gilt etwa bei internen Inkonsistenzen oder Speichermangel. Alle Fehlermeldungen werden in den Standardfehlerausgabestrom, gewöhnliche Ausgaben von ausgeführten Befehlen wird in die Standardausgabe geschrieben.

Die Eingabe des Interrupt-Zeichens (normalerweise `Strg-C` oder ENTF) bei der primären oder sekundären Eingabeaufforderung bricht die Eingabe ab und kehrt zur primären Eingabeaufforderung zurück.[2] Ein Interrupt während einer Befehlsausführung verursacht eine `KeyboardInterrupt`-Ausnahme, die durch eine `try`-Anweisung behandelt werden kann.

Ausführbare Python-Skripte

Auf BSD-ähnlichen Unixsystemen kann ein Pythonskript - ähnlich einem Shellskript - direkt ausführbar gemacht werden, indem man folgende Zeile (shebang) an den Anfang des Skripts schreibt

```
#!/usr/bin/env python3.3
```

Dabei wird vorausgesetzt, dass sich der Pfad zum Interpreter im `PATH` des Benutzers befindet. Die `#!` müssen die ersten zwei Zeichen der Datei sein. Auf manchen Plattformen muss diese erste Zeile mit einem unixoiden Zeilenende (`'\n'`) enden und nicht mit einem Windows-Zeilenende (`'\r\n'`). Hinweis: Die Raute `'#'` dient in Python dazu, einen Kommentar zu beginnen.

Einem solchen Skript können dann Ausführungsrechte mit Hilfe des Befehls **`chmod`** verliehen werden:

```
$ chmod +x myscript.py
```

Auf Windowssystemen gibt es den Begriff der "Ausführungsrechte" nicht. Das Python-Installationsprogramm verknüpft automatisch `.py`-Dateien mit `python.exe`, sodass ein Doppelklick auf eine Python-Datei diese als Skript ausführt. Die Dateinamenserweiterung kann auch `.pyw` lauten, in diesem Fall wird das normalerweise auftauchende Konsolenfenster unterdrückt.

Kodierung von Quellcode

Standardmäßig werden Python-Quelltextdateien als in UTF-8 kodiert behandelt. In dieser Kodierung können die Zeichen der meisten Sprachen gleichzeitig in Stringliteralen, Bezeichnern und Kommentaren verwendet werden. Die Standardbibliothek verwendet allerdings nur ASCII-Zeichen für Bezeichner - eine Konvention, der jeder portable Code folgen sollte. Um alle diese Zeichen korrekt darzustellen, muss ein Editor erkennen, dass die Datei UTF-8 kodiert ist und einen Font benutzen, der alle Zeichen der Datei unterstützt.

[2] Ein Problem mit dem GNU-readline-Paket kann dies verhindern.

Will man eine andere Kodierung als UTF-8 für eine Quelltextdatei verwenden, dann muss unmittelbar unterhalb der #! Zeile eine weitere, spezielle Kommentarzeile eingefügt werden, durch die die Kodierung festgelegt wird

```
# -*- coding: Kodierung -*-
```

Mit dieser Angabe wird alles in der Quelltextdatei so behandelt, als hätte es die Kodierung *Kodierung* an Stelle von UTF-8. Die Liste der möglichen Kodierungen findet man in der Python Library Reference, in der Sektion zu `codecs`.

Wenn ein Editor beispielsweise keine UTF-8 kodierten Dateien unterstützt und auf die Benutzung einer anderen Kodierung besteht, sagen wir mal Windows-1252, kann man durch folgende Kodierungszeile

```
# -*- coding: cp-1252 -*-
```

immernoch alle Zeichen des Windows-1252 Zeichensatzes im Quelltext verwenden. Dieser spezielle Kodierungskommentar muss in der *ersten oder zweiten* Zeile der Datei stehen.

Die interaktive Startup-Datei

Wenn Python interaktiv genutzt wird, ist es gelegentlich hilfreich, bei jedem Start des Interpreters einige Standardbefehle automatisch auszuführen. Das lässt sich erreichen, indem man eine Umgebungsvariable namens `PYTHONSTARTUP` erstellt, die auf eine Datei mit den Startup-Befehlen verweist. Dies ist vergleichbar mit der `.profile`-Datei von Unixshells.

Diese Datei wird nur in interaktiven Sitzungen gelesen. Wenn der Interpreter ein Skript ausführt oder `/dev/tty` explizit als Quelle angegeben wird - was ansonsten einer interaktiven Sitzung entspricht -, wird die Startup-Datei nicht berücksichtigt. Ausgeführt wird sie im selben Namensraum wie interaktive Befehle, so dass Objekte, die in der Startup-Datei definiert oder importiert werden, ohne Qualifizierung in der interaktiven Sitzung genutzt werden können. Auch die Eingabeaufforderungen `sys.ps1` und `sys.ps2` lassen sich in dieser Datei festlegen.

Sollen noch weitere Startup-Dateien aus dem aktuellen Verzeichnis gelesen werden, dann lässt sich dies durch Code wie `if os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py').read())` in der globalen Datei erreichen. Soll die Startup-Datei in einem Skript verwendet werden, muss das explizit in diesem Skript geschehen:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    exec(open(filename).read())
```

Die Customization Module

Python bietet zwei Haken, um es anzupassen: `sitecustomize` und `usercustomize`. Um es auszuprobieren, musst du zuerst den Ort deines Benutzer `site-packages` Ordners herausfinden. Starte Python und gib dies ein:

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.2/site-packages'
```

Dort kannst du eine Datei namens `usercustomize.py` anlegen und alles gewünschte eingeben. Es wird jeden Aufruf von Python beeinflussen, es sei denn der Aufruf enthält die Option `-s`, um den automatischen Import zu verhindern.

`sitecustomize` funktioniert genauso, aber es wird typischerweise von einem Administrator im globalen `site-packages` Ordner erstellt und vor `usercustomize` importiert. Mehr dazu in der Dokumentation des `site`-Moduls.

KAPITEL 3

Eine informelle Einführung in Python

Zum Ausprobieren der folgenden Beispiele muss alles eingetippt werden, was auf die Eingabeaufforderung (>>> oder ...) folgt. Zeilen die nicht mit einer solchen Eingabeaufforderung beginnen, sind Ausgaben des Interpreters. Steht die sekundäre Eingabeaufforderung ... allein in einer Zeile, dann muss eine Leerzeile eingegeben werden. Dadurch werden mehrzeilige Befehle abgeschlossen.

Viele Beispiele in diesem Tutorial enthalten Kommentare - auch solche, die im interaktiven Modus eingegeben werden. Kommentare beginnen in Python mit einer Raute # und gelten bis zum Ende der physikalischen Zeile. Ein Kommentar kann am Anfang einer Zeile beginnen oder im weiteren Verlauf einer Zeile, allerdings nicht innerhalb eines Zeichenkettenliterals. Eine Raute innerhalb eines Zeichenkettenliterals ist einfach nur eine Raute. Da Kommentare dazu dienen, Code zu erklären und von Python nicht interpretiert werden, können sie beim Abtippen weggelassen werden.

Ein paar Beispiele:

```
# Das ist der erste Kommentar
spam = 1                    # und dies ist der zweite Kommentar
                            # ... und jetzt ein dritter!
string = "# Dies ist kein Kommentar."
```

Benutzung von Python als Taschenrechner

Wir wollen ein paar einfache Python-Befehle ausprobieren: Starte den Interpreter und warte auf die primäre Eingabeaufforderung, >>>.

Zahlen

Der Interpreter kann wie ein Taschenrechner eingesetzt werden: Man kann einen Ausdruck eingeben und der Interpreter berechnet das Ergebnis. Die Syntax für solche Ausdrücke ist einfach: Die Operatoren +, −, * und / wirken genauso wie in den meisten anderen Sprachen (beispielsweise Pascal oder C); Klammern können zur Gruppierung benutzt werden. Zum Beispiel:

```
>>> 2 + 2
4
>>> # Dies ist ein Kommentar
... 2 + 2
4
>>> 2 + 2  # und dies ist ein Kommentar in derselben Zeile wie Code
4
>>> (50 - 5 * 6) / 4
5.0
>>> 8 / 5 # Brüche gehen nicht verloren, wenn man Ganzzahlen teilt
1.6
```

Anmerkung: Möglicherweise liefert die letzte Berechnung bei dir nicht genau das gleiche Ergebnis, weil sich Ergebnisse von Fließkommaberechnungen von Computer zu Computer unterscheiden können. Im weiteren Verlauf wird noch darauf eingegangen, wie man die Darstellung bei der Ausgabe von Fließkommazahlen festlegen kann. Siehe *Fließkomma-Arithmetik: Probleme und Einschränkungen* für eine ausführliche Diskussion von einigen Feinheiten von Fließkommazahlen und deren Repräsentation.

Um eine Ganzzahldivision auszuführen, die ein ganzzahliges Ergebnis liefert und den Bruchteil des Ergebnisses vernachlässigt, gibt es den Operator `//`:

```
>>> # Ganzzahldivision gibt ein abgerundetes Ergebnis zurück:
... 7 // 3
2
>>> 7 // -3
-3
```

Das Gleichheitszeichen (`'='`) wird benutzt um einer Variablen einen Wert zuzuweisen. Danach wird kein Ergebnis vor der nächsten interaktiven Eingabeaufforderung angezeigt:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

Ein Wert kann mehreren Variablen gleichzeitig zugewiesen werden:

```
>>> x = y = z = 0  # Null für x, y und z
>>> x
0
>>> y
0
>>> z
0
```

Variablen müssen "definiert" sein, bevor sie benutzt werden können, sonst tritt ein Fehler auf. Diese Definition geschieht durch eine Zuweisung:

```
>>> # Versuche eine undefinierte Variable abzurufen
... n
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Python bietet volle Unterstützung für Fließkommazahlen. Werden Operanden verschiedener Zahlentypen durch einen Operator verknüpft, dann werden ganzzahlige Operanden in Fließkommazahlen umgewandelt:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Auch komplexe Zahlen werden unterstützt. Der Imaginärteil wird mit dem Suffix `j` oder `J` angegeben. Komplexe Zahlen mit einem Realanteil, der von Null verschieden ist, werden als `(real+imagj)` geschrieben oder können mit der Funktion `complex(real, imag)` erzeugt werden.

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0, 1)
(-1+0j)
>>> 3 + 1j * 3
(3+3j)
>>> (3 + 1j) * 3
(9+3j)
>>> (1 + 2j) / (1 + 1j)
(1.5+0.5j)
```

Komplexe Zahlen werden immer durch zwei Fließkommazahlen repräsentiert, dem Realteil und dem Imaginärteil. Um diese Anteile einer komplexen Zahl z auszuwählen, stehen `z.real` und `z.imag` zur Verfügung.

```
>>> a = 1.5 + 0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

Die Konvertierungsfunktionen in Fließkommazahlen und Ganzzahlen (`float()`, `int()`) stehen für komplexe Zahlen nicht zur Verfügung. Man kann `abs(z)` verwenden, um den Betrag einer komplexen Zahl (als Fließkommazahl) zu berechnen, oder `z.real`, um den Realteil zu erhalten:

```
>>> a = 3.0 + 4.0j
>>> float(a)
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)   # sqrt(a.real**2 + a.imag**2)
5.0
>>>
```

Im interaktiven Modus wird der zuletzt ausgegebene Ausdruck der Variablen _ zugewiesen. Die ist besonders hilfreich, wenn man den Python-Interpreter als Taschenrechner einsetzt

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

```
>>>
```

Die Variable _ sollte man so behandeln, als wäre sie schreibgeschützt und ihr nicht explizit einen Wert zuweisen. Dadurch würde eine unabhängige lokale Variable mit demselben Namen erzeugt, die die eingebaute Variable _ mit ihrem speziellen Verhalten verdeckt.

Zeichenketten (Strings)

Außer mit Zahlen kann Python auch mit Zeichenketten umgehen, die auf unterschiedliche Weise darstellbar sind. Sie können in einfache oder doppelte Anführungszeichen eingeschlossen werden:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Ja," hat er gesagt.'
'"Ja," hat er gesagt.'
>>> "\"Ja,\" hat er gesagt."
'"Ja," hat er gesagt.'
>>> '"Isses nich\'," sagte sie.'
'"Isses nich\'," sagte sie.
```

Der Interpreter gibt das Ergebnis von Zeichenketten-Operationen auf die gleiche Weise aus, wie sie eingegeben werden: Innerhalb von Anführungszeichen und mit durch Backslashes maskierten Anführungszeichen oder anderen seltsamen Zeichen, um den exakten Wert wiederzugeben. Die Zeichenkette wird von doppelten Anführungszeichen eingeschlossen, wenn sie ein einfaches Anführungszeichen, aber keine doppelten enthält, sonst wird sie von einfachen Anführungszeichen eingeschlossen. Die Funktion `print()` produziert eine lesbarere Ausgabe.

Es gibt mehrere Möglichkeiten, mehrzeilige Zeichenkettenliterale zu erzeugen, zum Beispiel durch Fortsetzungszeilen, die mit einem Backslash am Ende der physikalischen Zeile anzeigen, dass die nächste Zeile die logische Fortsetzung der aktuellen ist:

```
hello = "Dies ist eine ziemlich lange Zeichenkette,\n\
die mehrere Zeilen Text enthält und wie man sie auch in C schreiben würde.\n\
    Achtung: Leerzeichen am Anfang haben eine Bedeutung\
für die Darstellung."

print(hello)
```

Zu beachten ist, dass Zeilenumbrüche immer noch in den Zeichenkette mit Hilfe von `\n` eingebettet werden müssen. Der auf den Backslash folgende Zeilenumbruch gehört allerdings nicht mit zur Zeichenkette. Die vom Beispiel erzeugte Ausgabe sieht so aus :

```
Dies ist eine ziemlich lange Zeichenkette,
die mehrere Zeilen Text enthält und wie man sie auch in C schreiben würde.
    Achtung: Leerzeichen am Anfang haben eine Bedeutung für die Darstellung.
```

Zeichenketten können auch mit einem Paar von dreifachen Anführungszeichen umgeben werden: `"` oder `'''`. Zeilenenden müssen nicht hierbei escaped werden, sondern werden in die Zeichenkette übernommen. Deshalb wird im folgende Beispiel das erste Zeilenende escaped, um die unerwünschte führende Leerzeile zu vermeiden:

```
print("""\
Usage: thingy [OPTIONS]
     -h                        Display this usage message
```

```
    -H hostname               Hostname to connect to
""")
```

Das erzeugt folgende Ausgabe:

```
Usage: thingy [OPTIONS]
     -h                       Display this usage message
     -H hostname              Hostname to connect to
```

Wenn wir den Zeichenkettenliteral zu einem "raw"-String machen, wird `\n` nicht in einen Zeilenumbruch umgewandelt; auch der Backslash am Ende und das Zeilenumbruch-Zeichen im Quellcode sind Teil der Zeichenkette. Das Beispiel:

```
hello = r"Dies ist eine ziemlich lange Zeichenkette,\n\
die mehrere Zeilen Text enthält und wie man sie auch in C schreiben würde."

print(hello)
```

führt zu folgender Ausgabe:

```
Dies ist eine ziemlich lange Zeichenkette,\n\
die mehrere Zeilen Text enthält und wie man sie auch in C schreiben würde.
```

Zeichenketten können mit dem +-Operator verkettet und mit $\star$ wiederholt werden:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Zwei Zeichenkettenliterale nebeneinander werden automatisch miteinander verknüpft. Die erste Zeile im obigen Beispiel hätte also auch `word = 'Help' 'A'` lauten können. Das funktioniert allerdings nur mit zwei Literalen, nicht mit beliebigen String-Ausdrücken:

```
>>> 'str' 'ing'             #Das ist ok
'string'
>>> 'str'.strip() + 'ing'   #Das ist ok
'string'
>>> 'str'.strip() 'ing'     #Das ist ungültig
 File "<stdin>", line 1, in ?
   'str'.strip() 'ing'
                 ^
SyntaxError: invalid syntax
```

Zeichenketten können indiziert werden, wobei das erste Zeichen einer Zeichenkette wie in C den Index 0 hat ("nullbasierte Zählung"). Es gibt keinen speziellen Zeichentyp (wie `char` in C) — ein Zeichen ist einfach eine Zeichenkette der Länge eins. Wie in der Programmiersprache Icon können Teile einer Zeichenkette mittels *Slice-Notation* (Ausschnittschreibweise) festgelegt werden. Angegeben werden zwei Indizes getrennt durch einen Doppelpunkt (`:`).

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

Slice-Indizes haben nützliche Standardwerte: Wird der erste Index ausgelassen, beginnt der Ausschnitt mit dem ersten Zeichen der Zeichenkette (Index 0), wird der zweite Index ausgelassen, reicht der Ausschnitt bis zum Ende der Zeichenkette

```
>>> word[:2]    # Die ersten beiden Zeichen
'He'
>>> word[2:]    # Alles außer den ersten beiden Zeichen
'lpA'
```

Im Unterschied zu einem C-String kann ein Python-String nicht verändert werden — Zeichenketten sind *unveränderbar* (*immutable*). Der Versuch, einer indizierten Position einer Zeichenkette etwas zuzuweisen, führt zu einer Fehlermeldung

```
>>> word[0] = 'x'
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
TypeError: 'str' object does not support item assignment
>>> word[:1] = 'Splat'
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
TypeError: 'str' object does not support slice assignment
```

Stattdessen erzeugt man einfach eine neue Zeichenkette mit dem kombinierten Inhalt

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[4]
'SplatA'
```

Noch ein Beispiel: `s[:i] + s[i:]` entspricht `s`.

```
>>> word[:2] + word[2:]
'HelpA'
>>> word[:3] + word[3:]
'HelpA'
```

Werden bei der Slice-Notation Indizes angegeben, die die tatsächliche Länge einer Zeichenkette überschreiten, führt dies nicht zu einer Fehlermeldung: Ein zu großer zweiter Index wird durch die Länge der Zeichenkette ersetzt und Ausschnitte, die keine Zeichen enthalten, liefern eine leere Zeichenkette zurück.

```
>>> word[1:100]
'elpA'
>>> word[10:]
''
>>> word[2:1]
''
```

Indizes können auch negative Zahlen sein — dann wird von rechts nach links gezählt. Zum Beispiel:

```
>>> word[-1]     # Das letzte Zeichen
'A'
>>> word[-2]     # Das vorletzte Zeichen
'p'
>>> word[-2:]    # Die letzten zwei Zeichen
'pA'
>>> word[:-2]    # Alles außer den letzten beiden Zeichen
'Hel'
```

Achtung: -0 ist dasselbe wie 0. Das heißt, die Zählung beginnt ganz normal von links!

```
>>> word[-0]     # (da -0 gleich 0)
'H'
```

Das automatische Kürzen bei Verwendung von Indizes, die außerhalb der tatsächlichen Länge der Zeichenkette liegen, funktioniert allerdings nur bei der Slice-Notation, nicht beim Zugriff auf ein einzelnes Zeichen mittels Indexschreibweise:

```
>>> word[-100:]
'HelpA'
>>> word[-10]    # Fehler
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
IndexError: string index out of range
```

Man kann sich die Indizes beim Slicing so vorstellen, dass sie *zwischen* den Zeichen liegen — wobei die linke Ecke des ersten Zeichens den Index 0 hat und die rechte Ecke des letzten Zeichens einer *n* Zeichen langen Zeichenkette den Index *n*. Ein Beispiel

```
 +---+---+---+---+---+
 | H | e | l | p | A |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

Die erste Zahlenreihe gibt die Position der Indizes 0...5 im String an, die zweite Reihe die entsprechenden negativen Indizes. Der Ausschnitt von *i* bis *j* besteht aus allen Zeichen zwischen den Positionen, die durch *i* beziehungsweise *j* gekennzeichnet werden.

Bei Verwendung von nicht-negativen Indizes entspricht die Länge des dadurch festgelegten Ausschnitts der Differenz der beiden Indizes, sofern beide innerhalb der tatsächlichen Grenzen der Zeichenkette liegen. Die Länge von `word[1:3]` ist zum Beispiel 2.

Die eingebaute Funktion `len()` gibt die Länge eines Strings zurück:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Siehe auch:

Sequence Types Zeichenketten gehören zu den *Sequenztypen* und verfügen über alle Operationen, die von diesen Typen unterstützt werden.

String Methods Strings verfügen über eine große Zahl an Methoden für grundlegende Transformationen und Suche.

String Formatting Informationen über Stringformatierung mit `str.format()` sind hier zu finden.

Old String Formatting Operations Die alten Formatierungsoperationen, die aufgerufen werden, wenn Strings und Unicodestrings die linken Operanden des `%`-Operators sind, werden hier ausführlich beschrieben.

Über Unicode

Beginnend mit Python 3.0 unterstützen alle Strings Unicode.

Unicode hat den Vorteil, dass es eine Ordnungszahl für jedes Zeichen in jedem Schriftstück bereitstellt, das in modernen und antiken Texten benutzt wird. Davor waren nur 256 Ordnungszahlen für Schriftzeichen möglich. Texte waren typischerweise an eine Codepage gebunden, die die Ordnungszahlen den Schriftzeichen zugeordnet hat. Das führte zu

großer Verwirrung, vor allem im Hinblick auf Internationalisierung von Software (üblicherweise `i18n` — `'i'` + 18 Zeichen + `'n'`) . Unicode löst diese Probleme, indem es eine Codepage für alle Schriftzeichen definiert.

Will man spezielle Zeichen in eine Zeichenketten einbinden, erreicht man das durch die Verwendung von Pythons *Unicode-Escape*-Schreibweise. Das folgende Beispiel zeigt wie:

```
>>> 'Hello\u0020World !'
'Hello World !'
```

Die Escapesequenz `\u0020` gibt an, dass das Unicodezeichen mit der Ordnungszahl 0x0020 (das Leerzeichen) an der gegebenen Position eingefügt werden soll.

Andere Zeichen werden interpretiert, indem ihre jeweiligen Ordnungszahlen direkt als Unicode-Ordnungszahlen benutzt werden. Hat man Zeichenkettenliterale in der normalen Latin-1-Kodierung, die in vielen westlichen Ländern benutzt wird, dann entsprechen die ersten 256 Zeichen von Unicode denselben Zeichen der Latin-1-Kodierung.

Neben diesen Standardkodierungen stellt Python eine ganze Reihe anderer Möglichkeiten bereit, Unicodestrings zu erstellen, sofern man die verwendete Kodierung kennt.

Zur Konvertierung von Zeichenketten in Bytefolgen stellen Stringobjekte die Methode `encode()` bereit, die den Namen der Kodierung als Argument entgegennimmt, und zwar möglichst in Kleinbuchstaben.

```
>>> "Äpfel".encode('utf-8')
b'\xc3\x84pfel'
```

Listen

Python kennt viele zusammengesetzte Datentypen (*compound data types*), die zur Gruppierung unterschiedlicher Werte verwendet werden können. Die flexibelste davon ist die Liste (*list*): Eine Liste von Werten (Elemente), die durch Kommas getrennt und von eckigen Klammern eingeschlossen werden. Listenelemente müssen nicht alle denselben Typ haben.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Ebenso wie die Indizierung bei Zeichenketten ist auch die Listenindizierung nullbasiert — das erste Element hat also den Index 0. Auch das von Zeichenketten bekannte Slicing sowie die Verkettung und Vervielfachung + bzw. $\star$ sind mit Listen möglich

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
```

Alle Slicing-Operationen geben eine neue Liste zurück, die die angeforderten Elemente enthält. Das bedeutet, dass die folgende Operation eine flache Kopie (*shallow copy*) der Liste `a` zurückgibt:

```
>>> a[:]
['spam', 'eggs', 100, 1234]
```

Im Unterschied zu Zeichenketten sind Listen allerdings *veränderbar* (*mutable*), so dass es möglich ist, innerhalb einer Liste Veränderungen vorzunehmen

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Selbst Zuweisungen an Slices sind möglich. Dadurch kann man die Länge einer Liste verändern oder sie sogar ganz leeren

```
>>> # Ein paar Elemente ersetzen:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Ein paar entfernen:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Ein paar einfügen:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> # (Eine Kopie von) sich selbst am Anfang einfügen:
>>> a[:0] = a
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
>>> # Die Liste leeren: Alle Elemente durch eine leere Liste  ersetzen
>>> a[:] = []
>>> a
[]
```

Die eingebaute Funktion `len()` lässt sich auch auf Listen anwenden:

```
>>> a = ['a', 'b', 'c', 'd']
>>> len(a)
4
```

Es ist auch möglich Listen zu verschachteln (*nest*), das heißt, Listen zu erzeugen, die andere Listen enthalten. Ein Beispiel:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
```

Man kann auch etwas ans Ende einer Liste hängen:

```
>>> p[1].append('xtra')
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Beachte, dass im letzten Beispiel `p[1]` und `q` wirklich auf dasselbe Objekt zeigen! Wir kommen später zur *Objektsemantik* zurück.

Erste Schritte zur Programmierung

Natürlich kann man Python für kompliziertere Aufgaben verwenden, als nur zwei und zwei zu addieren. Beispielsweise lassen sich die ersten Glieder der *Fibonacci-Folge* folgendermaßen erzeugen:

```
>>> # Fibonacci-Folge:
... # Die Summe der letzten beiden Elemente ergibt das nächste
... a, b = 0, 1
>>> while b < 10:
...     print(b)
...     a, b = b, a+b
...
1
1
2
3
5
8
```

Dieses Beispiel stellt ein paar neue Eigenschaften vor.

- Die erste Zeile enthält eine *Mehrfachzuweisung* (*multiple assignment*): Die Variablen `a` und `b` bekommen gleichzeitig die neuen Werte 0 und 1. In der letzten Zeile wird sie erneut eingesetzt, um zu zeigen, dass zuerst alle Ausdrücke auf der rechten Seite ausgewertet werden, bevor irgendeine Zuweisung vorgenommen wird! Die Ausdrücke auf der rechten Seite werden von links nach rechts ausgewertet.
- Die `while` Schleife wird solange ausgeführt, wie die Bedingung (hier: `b < 10`) wahr ist. In Python wie in C ist jede von Null verschiedene Zahl wahr (*True*), Null ist unwahr (*False*). Die Bedingung kann auch ein String- oder Listenwert sein, eigentlich sogar jede Sequenz. Alles mit einer von Null verschiedenen Länge ist wahr, leere Sequenzen sind unwahr. Die Bedingung im Beispiel ist ein einfacher Vergleich. Die normalen Vergleichsoperatoren werden wie in C geschrieben: < (kleiner als), > (größer als), == (gleich), <= (kleiner oder gleich), >= (größer oder gleich) und `!=` (ungleich).
- Der *Schleifenrumpf* ist *eingerückt* (*indented*): Durch Einrückung wird in Python eine Gruppierung vorgenommen. In der interaktiven Eingabeaufforderung muss man Tabs oder Leerzeichen für jede eingerückte Zeile eingeben. In der Praxis wird man kompliziertere Codestücke mit einem Texteditor vorbereiten und alle vernünftigen Editoren haben eine Möglichkeit, um automatisch einzurücken. Wenn eine zusammengesetzte Anweisung (*compound statement*) interaktiv eingegeben wird, muss eine Leerzeile darauf folgen, um anzuzeigen, dass sie komplett ist, da der Parser nicht erahnen kann, wenn man die letzte Zeile eingegeben hat. Beachte, dass jede Zeile in einem zusammengehörigen Block gleich eingerückt sein muss.
- Die Funktion `print()` gibt den Wert des Ausdrucks aus, der ihr übergeben wurde. Die Ausgabe unterscheidet sich bei Mehrfachausdrücken, Fließkommazahlen und Zeichenketten von der Ausgabe, die man erhält, wenn man die Ausdrücke einfach so eingibt (wie wir es vorher in den Taschenrechnerbeispielen gemacht haben). Zeichenketten werden ohne Anführungszeichen ausgegeben, und bei Angabe mehrere Argumente wird zwischen

je zwei Argumenten ein Leerzeichen eingefügt. So lassen sich einfache Formatierungen vornehmen, wie das Beispiel zeigt

```
>>> i = 256 * 256
>>> print('Der Wert von i ist', i)
Der Wert von i ist 65536
```

Durch Verwendung des Schlüsselwortarguments *end* kann der Zeilenumbruch nach der Ausgabe verhindert oder die Ausgabe mit einem anderen String beendet werden.

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print(b, end=' ')
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

KAPITEL 4

Mehr Werkzeuge zur Ablaufsteuerung

Neben der `while`-Anweisung, die gerade vorgestellt wurde, kennt Python – abgesehen von wenigen Abweichungen — die üblichen Anweisungen zur Ablaufsteuerung, die von anderen Sprachen bekannt sind.

`if`-Anweisungen

Ein Beispiel zur `if`-Anweisung

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...      x = 0
...      print('Negative changed to zero')
... elif x == 0:
...      print('Zero')
... elif x == 1:
...      print('Single')
... else:
...      print('More')
...
More
```

`else`-Zweig oder `elif`-Zweige sind optional. Im Unterschied zum `else`-Zweig, der nur einmal vorkommen kann, ist eine Abfolge von mehreren `elif`-Zweigen möglich; dadurch lassen sich verschachtelte Einrückungen vermeiden. Eine Abfolge von `if` ... `elif` ... `elif`-Zweigen ersetzt die `switch`- oder `case`-Konstrukte anderer Programmiersprachen.

`for`-Anweisungen

Die `for`-Anweisung in Python unterscheidet sich ein wenig von der, die man von C oder Pascal her kennt. Man kann nicht nur über eine Zahlenfolge iterieren (wie in Pascal) oder lediglich Schrittweite und Abbruchbedingung festlegen

(wie in C), sondern über eine beliebige Sequenz (also z. B. eine Liste oder Zeichenkette), und zwar in der Reihenfolge, in der die Elemente in der Sequenz vorkommen. Zum Beispiel:

```
>>> # Die Längen einiger Zeichenketten ermitteln:
... a = ['Katze', 'Fenster', 'rauswerfen']
>>> for x in a:
...     print(x, len(x))
...
Katze 5
Fenster 7
rauswerfen 10
```

Vorsicht ist geboten, wenn man versucht, Veränderungen an einer Sequenz vorzunehmen, über die gerade iteriert wird (was natürlich nur bei veränderbaren Sequenztypen, wie etwa Listen, passieren kann). Will man eine Liste verändern, über die man iteriert, um beispielsweise ausgewählte Elemente zu duplizieren, muss man über eine Kopie iterieren. Die Slice-Notation macht dies sehr einfach:

```
>>> for x in a[:]: # benutze eine Kopie der gesamten Liste
...    if len(x) > 7: a.insert(0, x)
...
>>> a
['rauswerfen', 'Katze', 'Fenster', 'rauswerfen']
```

Die Funktion `range()`

Wenn man wirklich über eine Zahlenfolge iterieren muss, bietet sich die eingebaute Funktion `range()` an, die arithmetische Folgen erzeugt.

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

Wird nur ein Argument angegeben, so beginnt der erzeugte Bereich bei Null und endet mit dem um 1 kleineren Wert des angegebenen Arguments. `range(10)` erzeugt die Zahlen von 0 bis einschließlich 9. Das entspricht den gültigen Indizes einer Sequenz mit zehn Elementen. Es ist ebenfalls möglich, den Bereich mit einem anderen Wert als Null zu beginnen oder auch eine bestimmte Schrittweite (*step*) festzulegen — sogar negative Schrittweiten sind möglich.

```
range(5, 10)
   5 bis 9

range(0, 10, 3)
   0, 3, 6, 9

range(-10, -100, -30)
  -10, -40, -70
```

Will man über die Indizes einer Sequenz iterieren, kann man `range()` und `len()` wie folgt kombinieren:

```
>>> a = ['Mary', 'hatte', 'ein', 'kleines', 'Lamm']
>>> for i in range(len(a)):
```

```
...     print(i, a[i])
...
0 Mary
1 hatte
2 ein
3 kleines
4 Lamm
```

Eleganter ist es jedoch, in solchen Fällen die Funktion `enumerate()` zu benutzen, siehe *Schleifentechniken*.

Etwas Seltsames passiert, wenn man einfach ein *range*-Objekt ausgeben will:

```
>>> print(range(10))
range(0, 10)
```

Zwar verhält sich das von `range()` zurückgegebene Objekt in etwa wie eine Liste, es ist jedoch in Wahrheit keine Liste. `range()` liefert ein Objekt zurück, das der Reihe nach die einzelnen Zahlen der Folge zurückliefert, die durch die an `range()` übergebenen Argumente festgelegt wurde. Dadurch lässt sich gegenüber der Erzeugung einer Liste Speicherplatz sparen.

Wir nennen solch ein Objekt *Iterable*, und es kann überall da eingesetzt werden, wo ein Objekt erwartet wird, das eine Folge von Elementen der Reihe nach "produziert", bis sein Vorrat erschöpft ist. Beispielsweise fungiert die `for`-Anweisung als ein solcher *Iterator*. Auch die Funktion `list()` ist ein solcher Iterator, die als Argument ein Iterable erwartet und eine Liste daraus macht

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

Später werden noch weitere Funktionen behandelt, die Iterables zurückgeben und Iterables als Argument aufnehmen.

`break`- und `continue`-Anweisungen und der `else`-Zweig bei Schleifen

Eine `break`-Anweisung in einem Schleifenrumpf bewirkt — wie in C — dass an dieser Stelle mit sofortiger Wirkung die sie unmittelbar umgebende Schleife verlassen wird.

Auch Schleifen-Anweisungen können einen `else`-Zweig haben. Dieser wird genau dann ausgeführt, wenn die Schleife *nicht* durch eine `break`-Anweisung abgebrochen wurde. Das folgende Beispiel zur Berechnung von Primzahlen veranschaulicht das.

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # Schleife wurde durchlaufen, ohne dass ein Faktor gefunden wurde
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
```

```
8 equals 2 * 4
9 equals 3 * 3
```

(Ja, dieser Code ist korrekt. Schau genau hin: Die `else` Klausel gehört zur `for`-Schleife, **nicht** zur `if`-Anweisung.)

Wenn die `else` Klausel bei einer Schleife benutzt wird, hat sie mehr mit dem `else` einer `try` Anweisung gemein, als mit dem der `if` Anweisung: Die `else` Klausel eines `try` wird ausgeführt, wenn keine Ausnahme auftritt und die einer Schleife, wenn kein `break` ausgeführt wird. Mehr Informationen zur `try` Anweisung und Ausnahmen gibt es unter *Ausnahmen behandeln*.

Entsprechend bewirkt die `continue`-Anweisung — ebenso von C entliehen — , dass sofort mit der nächsten Iteration der Schleife fortgefahren wird:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found a number", num)
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

`pass`-Anweisungen

Die `pass`-Anweisung tut nichts. Sie wird eingesetzt, wenn syntaktisch eine Anweisung benötigt wird, das Programm jedoch nichts tun soll. Ein Beispiel:

```
>>> while True:
...     pass  # geschäftiges Warten auf den Tastatur-Interrupt (Strg+C)
...
```

Auch bei der Erzeugung einer minimalen Klasse kann `pass` zum Einsatz kommen:

```
>>> class MyEmptyClass:
...     pass
...
```

`pass` lässt sich auch sinnvoll einsetzen als Platzhalter für den Rumpf einer Funktionen oder Schleife bei der “Top-Down”-Programmierung, um so zunächst auf einer abstrakteren Ebene zu denken

```
>>> def initlog(*args):
...     pass   # Implementieren nicht vergessen!
...
```

Funktionen definieren

Im folgenden Beispiel wird eine Funktion definiert, die die Fibonacci-Folge bis zu einer beliebigen Grenze ausgibt:

```
>>> def fib(n):    # die Fibonacci-Folge bis n ausgeben
...     """Print the Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Jetzt rufen wir die Funktion auf, die wir gerade definiert haben:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Das Schlüsselwort `def` leitet die *Definition* einer Funktion ein. Darauf folgt der Funktionsname und eine Auflistung der formalen Parameter, die allerdings auch leer sein kann. Die Anweisungen, die den Funktionskörper bilden, beginnen in der nächsten Zeile und müssen eingerückt sein.

Die erste Anweisung des Funktionskörpers kann auch ein Zeichenkettenliteral sein, ein so genannter Dokumentationsstring der Funktion, auch *Docstring* genannt. (Mehr zu Docstrings kann im Abschnitt *Dokumentationsstrings* nachgelesen werden.) Es gibt Werkzeuge, die Docstrings verwenden, um automatisch Online-Dokumentation oder gedruckte Dokumentation zu erzeugen oder es dem Anwender ermöglichen, interaktiv den Code zu durchsuchen. Die Verwendung von Docstrings ist eine gute Konvention, an die man sich bei der Programmierung nach Möglichkeit halten sollte.

Beim *Aufruf* einer Funktion kommt es zur Bildung eines lokalen Namensraums, der sich auf alle Bezeichner erstreckt, die im Funktionsrumpf (durch Zuweisung oder als Elemente der Parameterliste) neu definiert werden. Diese Bezeichner werden mit den ihnen zugeordneten Objekten in einer lokalen Symboltabelle abgelegt.

Wenn im Funktionsrumpf ein Bezeichner vorkommt, wird der Name zunächst in der lokalen Symboltabelle gesucht, danach in den lokalen Symboltabellen der umgebenden Funktionen, dann in der globalen Symboltabelle und schließlich in der Symboltabelle der eingebauten Namen. Darum ist es ohne weiteres nicht möglich, einer globalen Variablen innerhalb des lokalen Namensraums einer Funktion einen Wert zuzuweisen. Dadurch würde stattdessen eine neue, namensgleiche lokale Variable definiert, die die namensgleiche globale Variable überdeckt und dadurch auch den lesenden Zugriff auf diese globale Variable verhindert. Ein lesender Zugriff auf globale Variablen ist ansonsten immer möglich, ein schreibender Zugriff nur unter Verwendung der `global`-Anweisung.

Die konkreten Parameter (Argumente), die beim Funktionsaufruf übergeben werden, werden den formalen Parametern der Parameterliste zugeordnet und gehören damit zur lokalen Symboltabelle der Funktion. Das heißt, Argumente werden über *call by value* übergeben (wobei der *Wert* allerdings immer eine *Referenz* auf ein Objekt ist, nicht der Wert des Objektes selbst)[1]. Wenn eine Funktion eine andere Funktion aufruft, wird eine neue lokale Symboltabelle für diesen Aufruf erzeugt.

Eine Funktionsdefinition fügt den Funktionsnamen in die lokale Symboltabelle ein. Der Wert des Funktionsnamens hat einen Typ, der vom Interpreter als benutzerdefinierte Funktion erkannt wird. Dieser Wert kann einem anderen Namen zugewiesen werden, der dann ebenfalls als Funktion genutzt werden kann und so als Möglichkeit zur Umbenennung dient.

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Wer Erfahrung mit anderen Programmiersprachen hat, wird vielleicht einwenden, dass `fib` gar keine Funktion, sondern eine Prozedur ist, da sie keinen Wert zurückgibt. Tatsächlich geben aber auch Funktionen *ohne* eine `return`-Anweisung einen Wert zurück, wenn auch einen eher langweiligen, nämlich den eingebauten Namen `None` (“nichts”).

[1] Eigentlich wäre *call by object reference* eine bessere Beschreibung, denn wird ein veränderbares Objekt übergeben, sieht der Aufrufende jegliche Veränderungen, die der Aufgerufene am Objekt vornimmt (beispielsweise Elemente in eine Liste einfügt)

Die Ausgabe des Wertes `None` wird normalerweise vom Interpreter unterdrückt, wenn es der einzige Wert wäre, der ausgegeben wird. Möchte man ihn sehen, kann man ihn mittels `print()` sichtbar machen.:

```
>>> fib(0)
>>> print(fib(0))
None
```

Statt eine Abfolge von Zahlen in einer Funktion auszugeben, kann man auch eine Liste dieser Zahlen als Objekt zurückliefern.

```
>>> def fib2(n): # gibt die Fibonacci-Folge bis  n zurück
...     """Return a list containing the Fibonacci series up to n."""
...     result = list()
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # siehe unten
...         a, b = b, a + b
...     return result
...
>>> f100 = fib2(100)    # ruf es auf
>>> f100                # gib das Ergebnis aus
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Dieses Beispiel zeigt einige neue Eigenschaften von Python:

- Die `return`-Anweisung gibt einen Wert von einer Funktion zurück. Ohne einen Ausdruck als Argument gibt `return None` zurück; das gleiche gilt, wenn eine `return`-Anweisung fehlt.
- Die Anweisung `result.append(a)` ruft eine *Methode* des Listenobjektes in `result` auf. Eine Methode ist eine Funktion, die zu einem Objekt 'gehört' und wird mittels Punktnotation (`obj.methodname`) dargestellt. Dabei ist `obj` irgendein Objekt (es kann auch ein Ausdruck sein) und `methodname` der Name einer Methode, die vom Typ des Objektes definiert wird. Unterschiedliche Typen definieren verschiedene Methoden. Methoden verschiedener Typen können denselben Namen haben ohne doppeldeutig zu sein. (Es ist auch möglich, eigene Objekttypen zu erstellen, indem man *Klassen* benutzt, siehe *Klassen*.) Die Methode `append()`, die im Beispiel gezeigt wird, ist für Listenobjekte definiert. Sie hängt ein neues Element an das Ende der Liste an. Im Beispiel ist es äquivalent zu `result = result + [a]`, aber effizienter.

Mehr zum Definieren von Funktion

Funktionen lassen sich auch mit einer variablen Anzahl von Argumenten definieren. Dabei sind drei Varianten zu unterscheiden, die auch kombiniert werden können.

Standardwerte für Argumente

Die nützlichste Variante ist, einen Standardwert für ein oder mehrere Argumente anzugeben. Das erzeugt eine Funktion, die mit weniger Argumenten aufgerufen werden kann, als sie definitionsgemäß erlaubt. Zum Beispiel:

```
def ask_ok(prompt, retries=4, complaint='Bitte Ja oder Nein!'):
   while True:
        ok = input(prompt)
        if ok in ('j', 'J', 'ja', 'Ja'): return True
        if ok in ('n', 'N', 'ne', 'Ne', 'Nein'): return False
        retries = retries - 1
        if retries < 0:
```

```
            raise IOError('Benutzer abgelehnt!')
        print(complaint)
```

Diese Funktion könnte auf mehrere Arten aufgerufen werden:

- Indem man nur das vorgeschriebene Argument übergibt: `ask_ok("Willst du wirklich aufhören?")`
- Indem man zusätzlich ein optionales Argument übergibt: `ask_ok("Willst du die Datei überschreiben?", 2)`
- Oder indem man sogar alle übergibt: `ask_ok("Willst du die Datei überschreiben?", 2, "Komm schon, nur Ja oder Nein")`

Das Beispiel führt auch noch das Schlüsselwort `in` ein. Dieses überprüft ob ein gegebener Wert in einer Sequenz gegeben ist.

Die Standardwerte werden zum Zeitpunkt der Funktionsdefinition im *definierenden* Gültigkeitsbereich ausgewertet, so dass:

```
i = 5

def f(arg=i):
   print(arg)

i = 6
f()
```

`5` ausgeben wird.

Wichtige Warnung: Der Standardwert wird nur *einmal* ausgewertet. Das macht einen Unterschied, wenn der Standardwert veränderbares Objekt, wie beispielsweise eine Liste, ein Dictionary oder Instanzen der meisten Klassen, ist. Zum Beispiel häuft die folgende Funktion alle Argumente an, die ihr in aufeinanderfolgenden Aufrufen übergeben wurden:

```
def f(a, L=[]):
   L.append(a)
   return L

print(f(1))
print(f(2))
print(f(3))
```

Und sie gibt folgendes aus:

```
[1]
[1, 2]
[1, 2, 3]
```

Wenn man nicht will, dass der Standardwert von aufeinanderfolgenden Aufrufen gemeinsam benutzt wird, kann man die Funktion folgendermaßen umschreiben:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

Schlüsselwortargumente

Funktionen können auch mit *Schlüsselwortargumenten* in der Form `Schlüsselwort=Wert` aufgerufen werden. Zum Beispiel die folgende Funktion:

```
def parrot(voltage, state='völlig steif', action='fliegen', type='norwegische Blauling
↪'):
    print("-- Der Vogel würde selbst dann nicht", action, end=' ')
    print("selbst wenn Sie ihm ", voltage, "Volt durch den Schnabel jagen täten")
    print("-- Ganz erstaunlicher Vogel, der", type, "! Wunderhübsche Federn!")
    print("-- Er is", state, "!")
```

mindestens ein Argument (`voltage`) akzeptiert drei optionale Argumente (`state`, `action` und `type`) und kann mit allen folgenden Varianten aufgerufen werden:

```
parrot(4000)
parrot(action = 'VOOOOOM', voltage = 1000000)
parrot('Viertausend', state = 'an den Gänseblümchen riechen')
parrot('eine Million', 'keine Spur leben', 'springen')
```

die folgenden Aufrufe wären allerdings alle ungültig:

```
parrot()                     # das benötigte Argument fehlt
parrot(voltage=5.0, 'tot')   # auf ein Schlüsselwortargument folgt ein normales
parrot(110, voltage=220)     # doppelter Wert für ein Argument
parrot(actor='John Cleese')  # unbekanntes Schlüsselwort
```

Bei einem Funktionsaufruf müssen Schlüsselwortargumente nach positionsabhängigen Argumenten kommen. Alle übergebenen Schlüsselwortargumente müssen jeweils auf eines der Argumente passen, die die Funktion akzeptiert (beispielsweise ist `actor` kein gültiges Argument für die `parrot` Funktion), wobei ihre Reihenfolge aber unwichtig ist. Das gilt auch für nicht-optionale Argumente (beispielsweise ist `parrot(voltage=1000)` auch gültig). Kein Argument darf mehr als einen Wert zugewiesen bekommen. Ein Beispiel, das wegen dieser Einschränkung scheitert:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

Ist ein Parameter der Form `**name` in der Definition enthalten, bekommt dieser ein Dictionary (siehe Mapping Types), das alle Schlüsselwortargumente enthält, bis auf die, die in der Definition vorkommen. Dies kann mit einem Parameter der Form `*name`, der im nächsten Unterabschnitt beschrieben wird, kombiniert werden. Dieser bekommt ein Tupel, das alle positionsabhängigen Argumente enthält, die über die Anzahl der definierten hinausgehe. (`*name` muss aber vor `**name` kommen.) Wenn wir zum Beispiel eine Funktion wie diese definieren:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Haben sie", kind, "?")
    print("-- Tut mir leid,", kind, "ist leider gerade aus.")
    for arg in arguments:
        print(arg)
    print("-" * 40)
    keys = sorted(keywords.keys())
    for kw in keys:
        print(kw, ":", keywords[kw])
```

könnte sie so aufgerufen werden:

```
cheeseshop("Limburger", "Der ist sehr flüssig, mein Herr.",
           "Der ist wirklich sehr, SEHR flüssig, mein Herr.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

und natürlich würde sie folgendes ausgeben:

```
-- Haben sie Limburger ?
-- Tut mir leid, Limburger ist leider gerade aus.
Der ist sehr flüssig, mein Herr.
Der ist wirklich sehr, SEHR flüssig, mein Herr.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

Man beachte, dass die Liste der Schlüsselwortargumente erzeugt wird, indem das Ergebnis der Methode `keys()` sortiert wird, bevor dessen Inhalt ausgegeben wird. Tut man das nicht, ist die Reihenfolge der Ausgabe undefiniert.

Beliebig lange Argumentlisten

Die am wenigsten gebräuchliche Möglichkeit ist schließlich, festzulegen, dass eine Funktion mit einer beliebigen Zahl von Argumenten aufgerufen werden kann, die dann in ein Tupel (siehe *Tupel und Sequenzen*) verpackt werden. Vor diesem speziellen Argument kann eine beliebige Menge normaler Argumente vorkommen.

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Normalerweise wird dieses spezielle Argument an das Ende der Argumentliste gesetzt, weil es alle verbleibenden Argumente, mit denen die Funktion aufgerufen wird, aufnimmt. Alle Argumente, die in der Definition auf ein `*args` folgen, sind nur durch Schlüsselwortargumente zu übergeben (*'keyword-only'*) und nicht durch positionsabhängige.

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("Erde", "Mars", "Venus")
'Erde/Mars/Venus'
>>> concat("Erde", "Mars", "Venus", sep=".")
'Erde.Mars.Venus'
```

Argumentlisten auspacken

Die umgekehrte Situation ereignet sich, wenn die Argumente schon in einer Liste oder einem Tupel stecken, aber für einen Funktionsaufruf ausgepackt werden müssen, der separate positionsabhängige Argumente erfordert. Zum Beispiel erwartet die eingebaute Funktion `range()` getrennte Argumente für *Start* und *Stop*. Wenn sie aber nicht getrennt vorhanden sind, kann man im Funktionsaufruf den `*`-Operator benutzen, um die Argumente aus einer Liste oder einem Tupel auszupacken.

```
>>> list(range(3, 6))    # normaler Aufruf mit getrennten Argumenten
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))   # Aufruf mit Argumenten, die aus einer Liste ausgepackt werden
[3, 4, 5]
```

Analog können Dictionaries Schlüsselwortargumente mit dem **-Operator bereitstellen:

```
>>> def parrot(voltage, state=' völlig steif',
...     action='fliegen', type='norwegische Blauling'):
...     print("-- Der Vogel würde selbst dann nicht", action, end=' ')
...     print("selbst wenn Sie ihm ", voltage, "Volt durch den Schnabel jagen täten.")
...     print("-- Er is", state, "!")

>>> d = {"voltage": "vier Millionen", "state": "verdammt nochmal tot!", "action":
↪"FLIEGEN"}
>>> parrot(**d)
-- Der Vogel würde selbst dann nicht FLIEGEN selbst wenn sie ihm vier Millionen Volt␣
↪durch den Schnabel jagen täten.
-- Er is verdammt nochmal tot!
```

Lambda-Form - anonyme Funktion

Aufgrund der hohen Nachfrage, haben ein paar Merkmale, die in funktionalen Programmiersprachen wie Lisp üblich sind, Einzug in Python gehalten. Mit dem Schlüsselwort `lambda` können kleine anonyme Funktionen erstellt werden. Hier eine Funktion, die die Summe seiner zwei Argumente zurückgibt: `lambda a, b: a + b`. `lambda` kann überall genutzt werden, wo ein Funktionsobjekt benötigt wird. Semantisch ist es nur syntaktischer Zucker für eine normale Funktionsdefinition. Wie verschachtelte Funktionsdefinitionen, können in einer `lamdba`-Form Variablen der umgebenden Namensräume referenziert werden:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

Dokumentationsstrings

Hier nun ein paar Konventionen zum Inhalt und Formatieren von Dokumentationsstrings.

Die erste Zeile sollte immer eine kurze, prägnante Zusammenfassung des Zwecks des Objekts sein. Wegen der Kürze, sollte es nicht explizit auf den Namen oder den Typ des Objekts hinweisen, da diese durch andere Wege verfügbar sind (es sei denn, wenn der Name ein Verb ist, das den Ablauf der Funktion beschreibt). Dieser Zeile sollte mit einem Großbuchstaben anfangen und mit einem Punkt enden.

Enthält der Dokumentationsstring mehrere Zeilen, dann sollte die zweite Zeile leer sein, um die Zusammenfassung visuell vom Rest der Beschreibung zu trennen. Die folgenden Zeilen sollten aus einem oder mehrere Absätzen bestehen, die die Konventionen zum Aufruf des Objektes erläutern, seine Nebeneffekte etc.

Der Python-Parser entfernt die Einrückung von mehrzeiligen Zeichenkettenliteralen nicht, sodass Werkzeuge, die die Dokumentation verarbeiten, die Einrückung entfernen müssen, sofern das gewünscht ist. Dies geschieht aufgrund folgender Konvention: Die erste nicht-leere Zeile *nach* der ersten Zeile bestimmt den Umfang der Einrückung für den gesamten Dokumentationsstring. (Die erste Zeile kann man dafür nicht benutzen, da sie normalerweise neben dem öffnenden Anführungszeichen des Zeichenkettenliterals, sodass die Einrückung im Literal nicht erscheint.) Entsprechend dieser Einrückung werden vom Anfang jeder Zeile der Zeichenkette Leerzeichen entfernt. Zeilen, die weniger eingerückt sind, sollten nicht auftauchen, falls doch sollten alle führenden Leerzeichen der Zeile entfernt werden. Die Entsprechung der Leerzeichen sollte nach der Expansion von Tabs (üblicherweise zu 8 Leerzeichen) überprüft werden.

Hier ein Beispiel eines mehrzeiligen Docstrings:

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print(my_function.__doc__)
Do nothing, but document it.

    No, really, it doesn't do anything.
```

Intermezzo: Schreibstil

Jetzt da Du längere, komplexere Stücke in Python schreibst, ist es an der Zeit einmal über den Schreibstil (*coding style*) zu sprechen. Viele Sprachen können in verschiedenen Stilen geschrieben (präziser: *formatiert*) werden; davon sind manche lesbarer als andere. Es anderen leichter zu machen Deinen Code zu lesen ist immer eine gute Idee und sich einen schönen Schreibstil anzugewöhnen hilft dabei ungemein.

Für Python hat sich **PEP 8** als der Styleguide herauskristallisiert, dem die meisten Projekte folgen. Es fördert einen sehr lesbaren Schreibstil, der angenehm zu lesen ist. Jeder Pythonentwickler sollte ihn irgendwann einmal lesen, hier jedoch die wichtigsten Punkte:

- Benutze eine Einrückung von 4 Leerzeichen, keine Tabs.

  4 Leerzeichen sind ein guter Kompromiss zwischen geringer Einrückung, die eine größere Verschachtelungstiefe ermöglicht, und größerer Einrückung, die den Code leichter lesbar macht. Tabs führen zu Unordnung und sollten deshalb vermieden werden.

- Breche Zeilen so um, dass sie nicht über 79 Zeichen hinausgehen.

  Das ist hilfreich für Benutzer mit kleinen Bildschirmen und macht es auf größeren möglich mehrere Dateien nebeneinander zu betrachten.

- Benutze Leerzeilen, um Funktion und Klassen, sowie größere Codeblöcke innerhalb von Funktionen zu trennen.

- Verwende eine eigene Zeile für Kommentare, sofern das möglich ist.

- Schreibe Docstrings.

- Benutze Leerzeichen um Operatoren herum und nach Kommas, jedoch nicht direkt innerhalb von Klammerkonstrukten: `a = f(1, 2) + g(3, 4)`.

- Benenne Deine Klassen und Funktionen konsistent: Die Konvention schlägt `CamelCase` für Klassen und `klein_geschrieben_mit_unterstrichen` für Funktionen und Methoden vor. Benutze immer `self` als Namen für das erste Methoden Argument (mehr zu Klassen und Methoden, siehe *Eine erste Betrachtung von Klassen*).

- Benutze keine ausgefallenen Dateikodierungen, wenn Dein Code für ein internationales Publikum vorgesehen ist. Pythons Standardkodierung – UTF-8 – oder sogar einfaches ASCII ist in jedem Fall am Besten.

- Benutze auch keine nicht-ASCII-Zeichen in Bezeichnern, wenn es auch nur den Hauch einer Chance gibt, dass der Code von Menschen gelesen oder gewartet wird, die eine andere Sprache sprechen.

KAPITEL 5

Datenstrukturen

Dieses Kapitel beschreibt einige Dinge, die schon vorkamen, detaillierter und stellt auch ein paar neue Dinge vor.

Mehr zu Listen

Der Datentyp `list` hat noch ein paar andere Methoden. Hier sind alle Methoden von Listenobjekten:

`list.`**`append`**(*x*)
: Hängt ein neues Element an das Ende der Liste an. Äquivalent zu `a[len(a):] = [x]`.

`list.`**`extend`**(*L*)
: Erweitert die Liste, indem es alle Elemente der gegebenen Liste anhängt. Äquivalent zu `a[len(a):] = L`.

`list.`**`insert`**(*i*, *x*)
: Fügt ein Element an der gegebenen Position ein. Das erste Argument ist der Index des Elements, vor dem eingefügt werden soll, so fügt `a.insert(0, x)` am Anfang der Liste ein und `a.insert(len(a), x)` ist äquivalent zu `a.append(x)`.

`list.`**`remove`**(*x*)
: Entfernt das erste Element, dessen Wert *x* ist. Es gibt eine Fehlermeldung, wenn solch ein Element nicht existiert.

`list.`**`pop`**(`[`*i*`]`)
: Entfernt das Element an der gegebenen Position und gibt es zurück. Ist kein Index gegeben, entfernt `a.pop()` das letzte Element der Liste und gibt es zurück. (Die eckigen Klammern um das *i* in der Methodensignatur herum, zeigen an, dass das Argument optional ist und nicht, dass man dort eckige Klammern eintippen sollte. Du wirst diese Notation des öfteren in der Referenz der Pythonbibliothek sehen.

`list.`**`index`**(*x*)
: Gibt den Index des ersten Elements in der Liste zurück, dessen Wert *x* ist. Es gibt eine Fehlermeldung, wenn solch ein Element nicht existiert.

`list.`**`count`**(*x*)
: Gibt zurück, wie oft *x* in der Liste vorkommt.

`list.`**`sort`**()
: Sortiert die Elemente der Liste im selben Exemplar (*in place*).

`list.`**`reverse`**`()`
Kehrt die Reihenfolge der Listenelemente im selben Exemplar (*in place*) um.

Ein Beispiel, das die meisten Methoden von Listen benutzt:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

Vielleicht ist dir aufgefallen, dass bei Methoden wie `insert`, `remove` oder `sort`, die die Liste verändern, keinen Rückgabewert gedruckt wird – sie geben `None` zurück.[1] Dies ist ein Designprinzip für alle veränderlichen Datenstrukturen in Python.

Benutzung von Listen als Stack

Die Methoden von Listen, machen es sehr einfach eine Liste als Stapel (*Stack*) zu benutzen, bei dem das zuletzt hinzugekommene als Erstes abgerufen wird (“last-in, first-out”). Um ein Element auf den Stack zu legen, benutzt man `append()`. Um ein Element abzurufen, benutzt man `pop()` ohne expliziten Index. Zum Beispiel:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

[1] Andere Sprachen geben möglicherweise das veränderte Objekt selbst zurück, was die Verkettung von Methoden erlaubt, wie z.B. `d->insert(ä")->remove("b")->sort();`

Benutzung von Listen als Queue

Listen lassen sich auch bequem als Schlange (*Queue*) benutzen, wo das zuerst hinzugekommene Element auch zuerst abgerufen wird ("first-in, first-out"). Allerdings sind Listen nicht effizient für diesen Zweck. Während `append()` und `pop()` am Ende der Liste schnell sind, sind `insert()` und `pop()` am Anfang der Liste langsam (da alle anderen Elemente um eine Stelle verschoben werden muessen).

Um eine Queue zu implementieren benutzt man `collections.deque`, die so entworfen wurde, um beidseitig schnelle appends und pops bereitzustellen. Zum Beispiel:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry kommt an
>>> queue.append("Graham")          # Graham kommt an
>>> queue.popleft()                 # Der Erste geht jetzt
'Eric'
>>> queue.popleft()                 # Der Zweite geht jetzt
'John'
>>> queue                           # Verbleibende Schlange in der
>>>                                 # Reihenfolge der Ankunft
deque(['Michael', 'Terry', 'Graham'])
```

List Comprehensions

List Comprehensions bieten einen prägnanten Weg, um Listen zu erzeugen. Übliche Anwendungen sind solche, in denen man Listen erstellt, in denen jedes Element das Ergebnis eines Verfahrens ist, das auf jedes Mitglied einer Sequenz oder einem iterierbaren Objekt angewendet wird oder solche, in denen eine Teilfolge von Elementen, die eine bestimmte Bedingung erfüllen, erstellt wird.

Zum Beispiel, wenn wir eine Liste von Quadratzahlen erstellen wollen, wie:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Dann können wir das gleiche mit:

```
squares = [x**2 for x in range(10)]
```

erreichen. Dies ist auch äquivalent zu `squares = list(map(lambda x: x**2, range(10)))`, aber es ist kürzer und lesbarer.

Jede List Comprehension besteht aus eckigen Klammern, die einen Ausdruck gefolgt von einer `for`-Klausel, enthalten. Danach sind beliebig viele `for`- oder `if`-Klauseln zulässig. Das Ergebnis ist eine neue Liste, deren Elemente durch das Auswerten des Ausdrucks im Kontext der `for`- und `if`-Klauseln, die darauf folgen, erzeugt werden. Zum Beispiel kombiniert diese List Comprehension die Elemente zweier Listen, wenn sie nicht gleich sind:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

was äquivalent zu folgendem ist:

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Beachte, dass die Reihenfolge der `for` und `if` Anweisungen in beiden Codestücken gleich ist.

Falls der Ausdruck ein Tupel ist (z.B. `(x, y)` im vorigen Beispiel), muss er geklammert werden.

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # erzeuge eine neue Liste mit den verdoppelten Werten
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filtere die negativen Zahlen heraus
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # wende eine Funktion auf alle Elemente an
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # rufe eine Methode auf allen Elementen auf
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # erstelle eine Liste von 2-Tupeln der Form (zahl, quadrat)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # das Tupel muss geklammert werden, sonst wird ein Fehler erzeugt
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1, in ?
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # verflache eine Liste durch eine LC mit zwei 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List Comprehensions können auch komplexe Ausdrücke und verschachtelte Funktionen enthalten:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

Verschachtelte List Comprehensions

Der erste Ausruck in einer List Comprehension kann ein beliebiger Ausdruck sein, auch andere List Comprehensions.

Beim folgenden Beispiel wird eine 3x4 Matrix mit drei Listen der Länge vier implementiert:

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
```

```
...     [9, 10, 11, 12],
... ]
```

Die folgende List Comprehension vertauscht Listen und Spalten:

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Wie wir im vorherigen Abschnitt gesehen haben, wird die verschachtelte List Comprehension im Kontext des nachfolgenden `for` ausgewertet, damit ist das Beispiel äquivalent zu folgendem:

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

was wiederum gleich zu folgendem ist:

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

In Produktivcode sollte man aber eingebaute Funktionen komplexen Anweisungen vorziehen. Die Funktion `zip()` würde in diesem Fall gute Dienste leisten:

```
>>> list(zip(*mat))
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Details zum Sternchen sind unter *Argumentlisten auspacken* zu finden.

Die `del`-Anweisung

Es gibt einen Weg ein Listenelement durch seinen Index, statt durch seinen Wert zu löschen: Die `del`-Anweisung. Sie unterscheidet sich von der `pop()`-Methode, da sie keinen Wert zurückgibt. Die `del`-Anweisung kann auch dazu benutzt werden Abschnitte einer Liste zu löschen oder sie ganz zu leeren (was wir vorhin durch die Zuweisung einer leeren Liste an einen Abschnitt getan haben). Zum Beispiel:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
```

```
>>> a
[]
```

`del` kann auch dazu benutzt werden, ganze Variablen zu löschen:

```
>>> del a
```

Danach den Namen `a` zu referenzieren führt zu einer Fehlermeldung (zumindest bis dem Namen ein anderer Wert zugewiesen wird). Später werden werden wir noch andere Einsatzmöglichkeiten besprechen.

Tupel und Sequenzen

Wir haben gesehen, dass Listen und Zeichenketten viele Eigenschaften, wie Slicing und Indizierung, gemein haben. Beide sind Exemplare von *Sequenzdatentypen* (siehe Sequence Types). Da sich Python weiterentwickelt können auch noch andere Sequenzdatentypen hinzukommen. Es gibt aber noch einen weiteren standardmäßigen Sequenzdatentyp: Das Tupel.

Ein Tupel besteht aus mehreren Werten, die durch Kommas von einander getrennt sind, beispielsweise:

```
>>> t = 12345, 54321, 'Hallo!'
>>> t[0]
12345
>>> t
(12345, 54321, 'Hallo!')
>>> # Tupel können verschachtelt werden:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'Hallo!'), (1, 2, 3, 4, 5))
>>> # Tupel sind unveränderlich:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # aber sie können veränderliche Objekte enthalten:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Wie man sehen kann, werden die ausgegebenen Tupel immer von Klammern umgeben, sodass verschachtelte Tupel richtig interpretiert werden. Sie können mit oder ohne Klammern eingegeben werden, obwohl Klammern trotzdem sehr oft benötigt werden (wenn das Tupel Teil eines größeren Ausdrucks ist). Es ist nicht möglich den individuellen Elementen etwas zuzuweisen, aber es ist möglich Tupel zu erstellen, die veränderliche Objekte, wie beispielsweise Listen, enthalten.

Obwohl Tupel ähnlich zu Listen erscheinen, werden sie oft an verschiedenen Situationen und zu einem anderen Zweck verwendet. Tupel sind *unveränderlich* und enthalten üblicherweise eine heterogene Sequenz von Elementen, auf die über *unpacking* (siehe unten) oder über einen Index (oder sogar über Attribute im Fall von `namedtuples`) zugegriffen wird. Lisen sind *veränderlich* und ihre Elemente sind üblicherweise homogen und der Zugriff erfolgt durch das iterieren über die Liste.

Ein spezielles Problem ergibt sich in der Darstellung von Tupeln, die 0 oder 1 Elemente haben: Die Syntax hat ein paar Eigenheiten um das Problem zu lösen. Leere Tupel lassen sich mit einem leeren Klammerpaar darstellen und ein Tupel mit einem Element wird erstellt, indem dem Wert ein Komma nachgestellt wird, es reicht jedoch nicht, das Element nur in Klammern zu schreiben. Hässlich aber effektiv. Zum Beispiel:

```
>>> empty = ()
>>> singleton = 'Hallo',    # <-- das angehängte Komma nicht vergessen
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('Hallo',)
```

Die Anweisung `t = 12345, 54321, 'Hallo!'` ist ein Beispiel für das *Tupel packen* (*tuple packing*): Die Werte `12345`, `54321`, `'Hallo!'` werden zusammen in ein Tupel gepackt. Das Gegenteil ist ebenso möglich:

```
>>> x, y, z = t
```

Das wird passenderweise *Sequenz auspacken* (*sequence unpacking*) genannt und funktioniert mit jeder Sequenz auf der rechten Seite der Zuweisung. Die Anzahl der Namen auf der linken Seite muss genauso groß sein, wie die Länge der Sequenz. Eine Mehrfachzuweisung ist eigentlich nur eine Kombination von Tupel packen und dem Auspacken der Sequenz.

Mengen

Python enthält auch einen Datentyp für Mengen (*sets*). Eine Menge ist eine ungeordnete Sammlung ohne doppelte Elemente. Sie werden vor allem dazu benutzt, um zu testen, ob ein Element in der Menge vertreten ist und doppelte Einträge zu beseitigen. Mengenobjekte unterstützen ebenfalls mathematische Operationen wie Vereinigungsmenge, Schnittmenge, Differenz und symmetrische Differenz.

Geschweifte Klammern oder die Funktion `set()` können dazu genutzt werden Mengen zu erzeugen. Wichtig: Um eine leere Menge zu erzeugen muss man `set()` benutzen, `{}` ist dagegen nicht möglich. Letzteres erzeugt ein leeres Dictionary, eine Datenstruktur, die wir im nächsten Abschnitt besprechen.

Hier eine kurze Demonstration:

```
>>> basket = {'Apfel', 'Orange', 'Apfel', 'Birne', 'Orange', 'Banane'}
>>> print(basket)                      # zeigt, dass die Duplikate entfernt wurden
{'Orange', 'Birne', 'Apfel', 'Banane'}
>>> 'Orange' in basket                 # schnelles Testen auf Mitgliedschaft
True
>>> 'Fingerhirse' in basket
False

>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # einzelne Buchstaben in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # in a aber nicht in b
{'r', 'd', 'b'}
>>> a | b                              # in a oder b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # sowohl in a, als auch in b
{'a', 'c'}
>>> a ^ b                              # entweder in a oder b
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Wie *für Listen* gibt es auch eine “Set Comprehension”-Syntax:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

Dictionaries

Ein weiterer nützlicher Datentyp, der in Python eingebaut ist, ist das *Dictionary* (siehe Mapping Types). Dictionaries sind in manch anderen Sprachen als "assoziativer Speicher" oder "assoziative Arrays" zu finden. Anders als Sequenzen, die über Zahlen indizierbar sind, sind Dictionaries durch *Schlüssel* (*keys*), als die jeder unveränderbare Typ dienen kann, indizierbar; aus Zeichenketten und Zahlen kann immer solch ein Schlüssel gebildet werden. Tupel können als Schlüssel benutzt werden, wenn sie nur aus Zeichenketten, Zahlen oder Tupel bestehen; enthält ein Tupel direkt oder indirekt ein veränderbares Objekt, kann es nicht als Schlüssel genutzt werden. Listen können nicht als Schlüssel benutzt werden, da sie direkt veränderbar sind, sei es durch Indexzuweisung, Abschnittszuweisung oder Methoden wie `append()` and `extend()`.

Am Besten stellt man sich Dictionaries als ungeordnete Menge von *Schlüssel: Wert*-Paaren vor, mit der Anforderung, dass die Schlüssel innerhalb eines Dictionaries eindeutig sind. Ein Paar von geschweiften Klammern erstellt ein leeres Dictionary: `{}`. Schreibt man eine Reihe von Komma-getrennten Schlüssel-Wert-Paaren in die Klammern, fügt man diese als Anfangspaare dem Dictionary hinzu; dies ist ebenfalls die Art und Weise, wie Dictionaries ausgegeben werden.

Die Hauptoperationen, die an einem Dictionary durchgeführt werden, sind die Ablage eines Wertes unter einem Schlüssel und der Abruf eines Wertes mit dem gegebenen Schlüssel. Es ist auch möglich ein Schlüssel-Wert-Paar per `del` zu löschen. Legt man einen Wert unter einem Schlüssel ab, der schon benutzt wird, überschreibt man den alten Wert, der vorher mit diesem Schlüssel verknüpft war. Einen Wert mit einem nicht-existenten Schlüssel abrufen zu wollen, erzeugt eine Fehlermeldung.

Der Aufruf `list(d.keys())` auf ein Dictionary gibt eine Liste aller Schlüssel in zufälliger Reihenfolge zurück (will man sie sortiert haben, verwendet man einfach die Funktion `sorted(d.keys())` stattdessen).[2] Um zu überprüfen ob ein einzelner Schlüssel im Dictionary ist, lässt sich das Schlüsselwort `in` benutzen.

Hier ein kleines Beispiel wie man Dictionaries benutzt:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> list(tel.keys())
['irv', 'guido', 'jack']
>>> sorted(tel.keys())
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

Der `dict()`-Konstruktor erstellt Dictionaries direkt von Sequenzen von Schlüssel-Wert-Paare:

[2] Beim Aufruf von `d.keys()` wird ein *dictionary view*-Objekt zurückgeben. Es unterstützt Operationen wie Mitgliedschaftsprüfung (membership testing) und Iteration, aber sein Inhalt ist abhängig vom ursprünglichen Dictionary – es ist nur eine Ansicht (*view*).

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

Außerdem können "Dict Comprehensions" benutzt werden, um Dictionaries von willkürlichen Schlüssel und Wert Ausdrücken zu erstellen:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Sind die Schlüssel einfache Zeichenketten, ist es manchmal einfacher, die Paare als Schlüsselwort-Argumente anzugeben:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

Schleifentechniken

Wenn man über Dictionaries iteriert, lassen sich der Schlüssel und der entsprechende Wert gleichzeitig durch die Methode `items()` abrufen.

```
>>> knights = {'Gallahad': 'der Reine', 'Robin': 'der Mutige'}
>>> for k, v in knights.items():
...     print(k, v)
...
Gallahad der Reine
Robin der Mutige
```

Iteriert man über eine Sequenz, lassen sich der Index und das entsprechende Objekt gleichzeitig über die Funktion `enumerate()` abrufen.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

Um über mehrere Sequenzen gleichzeitig zu iterieren, können die Einträge mit Hilfe der `zip()`-Funktion gruppiert werden.

```
>>> questions = ['Name', 'Auftrag']
>>> answers = ['Lancelot', 'die Suche nach dem Heiligen Gral']
>>> for q, a in zip(questions, answers):
...     print('Was ist dein {0}?  Er ist {1}.'.format(q, a))
...
Was ist dein Name?  Er ist Lancelot.
Was ist dein Auftrag?  Er ist die Suche nach dem Heiligen Gral.
```

Um über eine Sequenz in umgekehrter Reihenfolge zu iterieren, gibt man die Sequenz zuerst in richtiger Reihenfolge an und ruft dann die Funktion `reversed()` auf.

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
```

```
7
5
3
1
```

Um über eine Sequenz in sortierter Reihenfolge zu iterieren, gibt es die Funktion `sorted()`, die eine neue, sortierte Liste zurückgibt, die ursprüngliche Sequenz, aber nicht anrührt.

```
>>> basket = ['Apfel', 'Orange', 'Apfel', 'Birne', 'Orange', 'Banane']
>>> for f in sorted(set(basket)):
...     print(f)
...
Apfel
Banane
Birne
Orange
```

Mehr zu Bedingungen

Die Bedingungen, die in `while`- und `if`-Anweisungen benutzt werden, können jegliche Operatoren enthalten, nicht nur Vergleiche.

Die Vergleichsoperatoren `in` und `not in` überprüfen, ob ein Wert in einer Sequenz (nicht) vorkommt. Die Operatoren `is` und `is not` vergleichen auf Objektidentität, d.h. ob zwei Objekte dieselben sind; dies ist aber nur wichtig für veränderbare Objekte wie Listen. Alle Vergleichsoperatoren haben dieselbe Priorität, die geringer als die von numerischen Operatoren ist.

Vergleiche können aneinandergereiht werden. Zum Beispiel überprüft `a < b == c`, ob `a` kleiner als `b` ist und darüberhinaus `b` gleich `c` ist.

Vergleiche können kombiniert werden, indem man die boolschen Operatoren `and` und `or` benutzt. Das Ergebnis des Vergleiches (oder jedes anderen boolschen Ausdrucks) kann mit `not` negiert werden. Sie haben eine geringere Priorität als Vergleichsoperatoren; von ihnen hat `not` die höchste und `or` die niedrigste Priorität, sodass `A and not B or C` äquivalent zu `(A and (not B)) or C` ist. Wie üblich können auch hier Klammern benutzt werden, um die gewünschte Gruppierung auszudrücken.

Die boolschen Operatoren `and` und `or` sind sogenannte *Kurzschluss-* (*short-circuit*) Operatoren: Ihre Argumente werden von links nach rechts ausgewertet und die Auswertung wird abgebrochen, sobald das Ergebnis feststeht. Zum Beispiel, wenn `A` und `C` wahr, aber `B` unwahr ist, wird `C` von `A and B and C` nicht ausgewertet. Werden sie als genereller Wert und nicht als Boolean benutzt, ist der Rückgabewert eines Kurzschluss-Operators das zuletzt ausgewertete Argument.

Es ist möglich das Ergebnis eines Vergleiches oder eines anderen boolschen Ausdruck einer Variablen zuzuweisen. Zum Beispiel:

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Tanz'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Wichtig ist, dass in Python, anders als in C, Zuweisungen nicht innerhalb eines Ausdrucks vorkommen können. C-Programmierer mögen darüber murren, aber diese Einschränkung vermeidet eine in C übliche Fehlerklasse, in einem Ausdruck = , statt des beabsichtigten ==, zu schreiben.

Vergleich von Sequenzen mit anderen Typen

Sequenzobjekte können mit anderen Objekten desselben Sequenztyps verglichen werden. Der Vergleich benutzt eine *lexikographische* Ordnung: Zuerst werden die ersten beiden Elemente verglichen, unterscheiden sie sich, bestimmt das das Ergebnis des Vergleiches. Sind sie gleich, werden die nächsten zwei Elemente verglichen, und so weiter, bis sich eine der beiden erschöpft. Sind zwei Elemente, die verglichen werden sollen wiederum Sequenzen desselben Sequenztyps, wird der lexikographische Vergleich rekursiv durchgeführt. Sind alle Elemente zweier Sequenzen gleich, werden die Sequenzen für gleich befunden. Ist eine Sequenz eine anfängliche Teilfolge der anderen, so ist die kürzere die kleinere (geringere). Die Lexikographische Ordnung von Zeichenketten benutzt die Nummer des Unicode-Codepoints, um einzelne Zeichen zu ordnen. Ein paar Beispiele für Vergleiche zwischen Sequenzen des gleichen Typs:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Das Vergleichen von Objekten verschiedenen Typs durch < oder > ist erlaubt, sofern die Objekte passende Vergleichsmethoden haben. Zum Beispiel werden numerische Typen anhand ihres numerischen Wertes verglichen, sodass 0 0.0 gleicht, usw. Andernfalls wird der Interpreter eine `TypeError`-Ausnahme verursachen, statt eine willkürliche Ordnung bereitzustellen.

KAPITEL 6

Module

Wenn man den Python-Interpreter beendet und erneut startet, gehen alle vorgenommenen Definitionen (Funktionen und Variablen) verloren. Deshalb benutzt man vorzugsweise einen Text-Editor, um längere und komplexere Programme zu schreiben und verwendet eine Datei als Eingabe für den Interpreter. Diese Datei wird auch als Skript bezeichnet. Wird ein Programm länger, teilt man es besser in mehrere Dateien auf, um es leichter warten zu können. Außerdem ist es von Vorteil, nützliche Funktionen in mehreren Programmen verwenden zu können, ohne sie in jedem Programm erneut definieren zu müssen.

Hierfür bietet Python die Möglichkeit, etwas in einer Datei zu definieren und diese in einer anderen Datei oder in der interaktiven Konsole wieder zu verwenden. So eine Datei wird als *Modul* bezeichnet. Definitionen eines Moduls können in anderen Modulen oder in das *Hauptmodul importiert* werden, welches die Gesamtheit aller Funktionen und Variablen enthält, auf die man in einem Skript zugreifen kann.

Ein Modul ist eine Datei, die Python-Definitionen und -Anweisungen beinhaltet. Der Dateiname mit dem `.py`-Suffix entspricht dem Namen des Moduls. Innerhalb eines Moduls ist der Modulname als `__name__` verfügbar (globale Variable des Typs String). Zum Beispiel: Öffne einen Editor Deiner Wahl und erstelle eine Datei im aktuellen Verzeichnis mit dem Namen `fibo.py` und folgendem Inhalt:

```
# Fibonacci-Zahlen-Modul

def fib(n):    # schreibe Fibonacci-Folge bis n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n): # gib die Fibonacci-Folge zurück bis n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Öffne danach Deinen Python-Interpreter und importiere das Modul mit folgendem Befehl:

```
>>> import fibo
```

Dieser Befehl fügt die von `fibo.py` definierten Funktionen nicht automatisch in die globale Symboltabelle (*symbol table*) ein, sondern nur den Modulnamen `fibo`. Um die Funktionen anzusprechen, benutzt man den Modulnamen:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Wenn man plant, eine Funktion öfters zu verwenden, kann man diese an einen lokalen Namen binden.

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Mehr zum Thema Module

Ein Modul kann sowohl ausführbare Anweisungen, als auch Definitionen einer Funktion enthalten. Diese Anweisungen sind dazu gedacht, das Modul zu initialisieren, und werden nur ausgeführt, wenn das Modul zum ersten Mal importiert wird.

Jedes Modul hat seine eigene, private Symboltabelle, welche wiederum als globale Symboltabelle von allen Funktion in diesem Modul verwendet wird. Daher kann der Verfasser eines Moduls ohne Bedenken globale Variablen in seinem Modul verwenden, da sie sich nicht mit den globalen Variablen des Benutzers überschneiden können. Andererseits kann man (wenn man weiß, was man tut) auch die globalen Variablen eines Moduls verändern, indem man die gleiche Schreibweise verwendet, um auch dessen Funktionen anzusprechen, `modname.itemname`.

Module können andere Module importieren. Es ist üblich, aber nicht zwingend notwendig, dass man alle `import`-Anweisungen an den Anfang eines Moduls setzt (oder in diesem Fall Skripts). Die Namen der importierten Module werden in die Symboltabelle des importierenden Moduls eingefügt.

Es gibt eine Variante der `import`-Anweisung, welche bestimmte Namen aus einem Modul direkt in die Symboltabelle des importierenden Moduls einfügt. Beispielsweise:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Diese Variante fügt allerdings nicht den Modulnamen, aus dem die Namen importiert werden, in die lokale Symboltabelle ein, sondern nur die aufgeführten. In diesem Beispiel wird `fibo` also nicht eingefügt.

Zusätzlich gibt es eine Variante um alle Namen eines Moduls zu importieren:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 37
```

Hiermit werden alle Namen, sofern sie nicht mit einem Unterstrich (_) beginnen , importiert. In den meisten Fällen wird diese Variante nicht verwendet, denn dadurch werden unbekannte Namen in den Interpreter importiert und so kann es vorkommen, dass einige Namen überschrieben werden, die bereits definiert worden sind.

Beachte, dass der `*`-Import eines Moduls oder Paketes im allgemeinen verpönt ist, da er oft schlecht lesbaren Code verursacht. Allerdings ist es in Ordnung ihn im interaktiven Interpreter zu benutzen, um weniger tippen zu müssen.

Bemerkung: Aus Effizienzgründen wird jedes Modul nur einmal durch eine Interpreter-Sitzung importiert. Deshalb muss man den Interpreter bei Veränderung der Module neustarten - oder man benutzt `imp.reload()`, beispielsweise `import imp; imp.reload(modulename)`, falls es nur ein Modul ist, welches man interaktiv testen will.

Module als Skript aufrufen

Wenn man ein Python-Modul folgendermaßen aufruft:

```
python fibo.py <Argumente>
```

wird der Code im Modul genauso ausgeführt, als hätte man das Modul importiert. Der einzige Unterschied ist, dass `__name__` jetzt `"__main__"` ist und nicht mehr der Name des Moduls. Wenn man nun folgende Zeilen an das Ende des Moduls anfügt:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

kann man die Datei sowohl als Skript als auch als importierbares Modul nutzen, da der Code, der die Kommandozeile auswertet, nur ausgeführt wird, wenn das Modul direkt als *Hauptdatei* ausgeführt wird:

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

Beim Import des Moduls wird dieser Code nicht ausgeführt:

```
>>> import fibo
>>>
```

Dies wird oft dazu verwendet, um entweder eine bequeme Benutzerschnittstelle zum Modul bereitzustellen oder zu Testzwecken (wenn das Modul als Skript ausgeführt wird, wird eine Testsuite gestartet).

Der Modul-Suchpfad

Wenn ein Modul mit dem Namen `spam` importiert wird, sucht der Interpreter zuerst nach einem eingebauten Modul mit diesem Namen. Wird es nicht gefunden, sucht er nach einer Datei, die `spam.py` benannt ist, in der Liste von Verzeichnissen der Variable `sys.path`. `sys.path` wird mit

- dem Verzeichnis, das das Eingabeskript enthält (oder das aktuelle Verzeichnis),
- `PYTHONPATH` (eine Liste von Verzeichnissen, mit derselben Syntax wie die Shell-Variable `PATH`)
- der installationsabhängige Standard.

Nach der Initialisierung können Python-Programme `sys.path` modifizieren. Das Verzeichnis, das das laufende Skript enthält befindet sich am Anfang des Suchpfads, noch vor den Pfaden der Standardbibliothek. Das bedeutet, dass Module in diesem Verzeichnis anstelle von Modulen der Standardbibliothek geladen werden, wenn sie denselben Namen haben. Dies ist ein Fehler, sofern man das Modul der Standardbibliothek nicht ersetzen will. Siehe Abschnitt *Standardmodule* für mehr Informationen.

“Kompilierte” Python-Dateien

Um den Start von kurzen Programmen, die viele Standardmodule verwenden, schneller zu machen, werden Dateien erstellt, welche bereits “byte-kompiliert” sind. Existiert eine Datei mit dem Namen `spam.pyc`, so ist das eine “byte-kompilierte” Version der Datei `spam.py` und des Moduls `spam`. Der Zeitpunkt an dem die Datei `spam.py` zuletzt geändert wurde, wird in `spam.pyc` festgehalten. Falls die Zeiten nicht übereinstimmen, wird die `.pyc` ignoriert.

Normalerweise muss man nichts tun, damit die `spam.pyc`-Datei erstellt wird. Immer, wenn `spam.py` erfolgreich kompiliert wird, wird auch versucht die kompilierte Version in `spam.pyc` zu schreiben. Es wird kein Fehler geworfen, wenn der Vorgang scheitert; wenn aus irgendeinem Grund die Datei nicht vollständig geschrieben sein sollte, wird die daraus resultierende `spam.pyc` automatisch als fehlerhaft erkannt und damit später ignoriert. Der Inhalt der `spam.pyc` ist plattformunabhängig, wodurch man ein Modul Verzeichnis mit anderen Maschinen ohne Rücksicht auf ihre Architektur teilen kann.

Einige Tipps für Experten:

- Wird der Python-Interpreter mit dem `-O`-Flag gestartet, so wird der optimierte Code in `.pyo`-Dateien gespeichert. Optimierter Code hilft momentan nicht viel, da er lediglich `assert`-Anweisungen entfernt. Wird `-O` verwendet, wird der *komplette Bytecode* optimiert; `.pyc` werden ignoriert und `.py`-Dateien werden zu optimiertem Bytecode kompiliert.
- Werden dem Python-Interpreter zwei `-O`-Flags übergeben, vollzieht der Bytecode-Compiler Optimierungen, die zu einer Fehlfunktion des Programms führen können. Momentan werden nur `__doc__`-Strings aus dem Bytecode entfernt, was zu kleineren `.pyo`-Dateien führt. Da einige Programme sich darauf verlassen, dass sie verfügbar sind, sollte man diese Option nur aktivieren, wenn man weiß, was man tut.
- Ein Programm wird in keinster Weise schneller ausgeführt, wenn es aus einer `.pyc` oder `.pyo` anstatt aus einer `.py`-Datei gelesen wird. Der einzige Geschwindigkeitsvorteil ist beim Starten der Dateien.
- Wenn ein Skript durch das Aufrufen über die Kommandozeile ausgeführt wird, wird der Bytecode nie in eine `.pyc`- oder `.pyo`-Datei geschrieben. Deshalb kann die Startzeit eines Skripts durch das Auslagern des Codes in ein Modul reduziert werden. Es ist auch möglich eine `.pyc`- oder `.pyo`-Datei direkt in der Kommandozeile auszuführen.
- Es ist möglich, eine `.pyc`- oder `.pyo`-Datei zu haben, ohne dass eine Datei mit dem Namen `spam.py` für selbiges Modul existiert. Dies kann dazu genutzt werden, Python-Code auszuliefern, der relativ schwer rekonstruiert werden kann.
- Das Modul `compileall` kann `.pyc`-Dateien (oder auch `.pyo`, wenn `-O` genutzt wird) aus allen Modulen eines Verzeichnisses erzeugen.

Standardmodule

Python wird mit einer Bibliothek von Standardmodulen ausgeliefert, welche in der Python Library Reference beschrieben werden. Einige Module sind in den Interpreter eingebaut. Diese bieten Zugang zu Operationen, die nicht Teil des Sprachkerns sind, aber trotzdem eingebaut sind. Entweder, um Zugang zu Systemoperationen (wie z. B. Systemaufrufe) bereitzustellen oder aus Effizienzgründen. Die Zusammenstellung dieser Module ist eine Option in der Konfiguration, welche auch von der verwendeten Plattform abhängig ist. Beispielsweise ist das `winreg`-Modul nur unter Windows-Systemen verfügbar. Ein bestimmtes Modul verdient besondere Aufmerksamkeit: `sys`, welches in jeden Python-Interpreter eingebaut ist. Die Variablen `sys.ps1` und `sys.ps2` definieren die primären und sekundären Eingabeaufforderungen, die in der Kommandozeile verwendet werden:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
```

```
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

Diese beiden Variablen werden nur definiert, wenn der Interpreter im interaktiven Modus ist.

Die Variable `sys.path` ist eine Liste von Zeichenketten, die den Suchpfad des Interpreters vorgibt. Sie ist mit einem Standardpfad voreingestellt, der aus der Umgebungsvariable `PYTHONPATH` entnommen wird oder aus einem eingebauten Standardwert, falls `PYTHONPATH` nicht gesetzt ist. Man kann diese Variable mit normalen Listenoperationen verändern:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

Die `dir()`-Funktion

Die eingebaute Funktion `dir()` wird benutzt, um herauszufinden, welche Namen in einem Modul definiert sind. Es wird eine sortierte Liste von Strings zurückgegeben:

```
    >>> import fibo, sys
    >>> dir(fibo)
    ['__name__', 'fib', 'fib2']
    >>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
'__stdin__', '__stdout__', '_getframe', 'api_version', 'argv',
'builtin_module_names', 'byteorder', 'callstats', 'copyright',
'displayhook', 'exc_info', 'excepthook', 'exec_prefix', 'executable',
'exit', 'getdefaultencoding', 'getdlopenflags', 'getrecursionlimit',
'getrefcount', 'hexversion', 'maxint', 'maxunicode', 'meta_path', 'modules',
'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
'ps2', 'setcheckinterval', 'setdlopenflags', 'setprofile',
'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'version',
'version_info', 'warnoptions']
```

Wenn man keine Parameter übergibt, liefert `dir()` eine Liste der aktuell definierten Namen:

```
    >>> a = [1, 2, 3, 4, 5]
    >>> import fibo
    >>> fib = fibo.fib
    >>> dir()
['__builtins__', '__doc__', '__file__', '__name__', 'a', 'fib', 'fibo',
'sys']
```

Zu Beachten ist, dass alle Typen von Namen ausgegeben werden: Variablen, Module, Funktionen, etc.

`dir()` listet allerdings nicht die Namen der eingebauten Funktionen und Variablen auf. Falls man diese auflisten will, muss man das Standardmodul `builtins` verwenden:

```
    >>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis',
'EnvironmentError', 'Exception', 'False', 'FloatingPointError',
'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning',
'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
```

```
'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError',
'RuntimeWarning', 'StopIteration', 'SyntaxError', 'SyntaxWarning',
'SystemError', 'SystemExit', 'TabError', 'True', 'TypeError',
'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
'ValueError', 'Warning', 'ZeroDivisionError', '__build_class__',
'__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'chr',
'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr',
'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter',
'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash',
'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter',
'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min',
'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit',
'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted',
'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

Pakete

Pakete werden dazu verwendet, den Modul-Namensraum von Python zu strukturieren, indem man Modulnamen durch Punkte trennt. Zum Beispiel verweist `A.B` auf ein Untermodul `B` im Paket `A`. Genauso wie die Verwendung von Modulen den Autor von Modulen davor bewahrt, sich Sorgen um andere globale Variablennamen zu machen, so bewahrt die Verwendung von Modulen, die durch mehrere Punkte getrennt sind, den Autor davor, sich Sorgen um andere Modulnamen machen zu müssen.

Angenommen man will eine Sammlung von Modulen (ein "Paket") erstellen, um Audiodateien und -daten einheitlich zu bearbeiten. Es gibt unzählige verschiedene Audioformate (gewöhnlicherweise erkennt man diese an Ihrer Dateiendung, z.B. `.wav`, `.aiff`, `.au`), sodass man eine ständig wachsende Sammlung von Modulen erstellen und warten muss. Außerdem will man auch verschiedene Arbeiten an den Audiodaten verrichten (wie zum Beispiel Mixen, Echo hinzufügen, etc.), also wird man immer wieder Module schreiben, die diese Arbeiten ausführen. Hier eine mögliche Struktur für so ein Paket (ausgedrückt in der Form eines hierarchischen Dateisystems):

```
sound/                          Top-level package
        __init__.py             Initialize the sound package
        formats/                Subpackage for file format conversions
                __init__.py
                wavread.py
                wavwrite.py
                aiffread.py
                aiffwrite.py
                auread.py
                auwrite.py
                ...
        effects/                Subpackage for sound effects
                __init__.py
                echo.py
                surround.py
                reverse.py
                ...
        filters/                Subpackage for filters
                __init__.py
                equalizer.py
                vocoder.py
```

```
                karaoke.py
                ...
```

Wenn man das Paket importiert, sucht Python durch die Verzeichnisse im `sys.path`, um nach dem Paket in einem Unterverzeichnis zu suchen.

Die `__init__.py`-Datei wird benötigt, damit Python das Verzeichnis als Paket behandelt. Dies hat den Grund, dass Verzeichnisse mit einem normalen Namen, wie z.B. `string`, nicht unbeabsichtigt Module verstecken, die weiter hinten im Suchpfad erscheinen. Im einfachsten Fall ist `__init__.py` eine leere Datei, sie kann allerdings auch Initialisierungscode für das Paket enthalten oder die `__all__`-Variable setzen, welche später genauer beschrieben wird.

Benutzer eines Pakets können individuelle Module aus dem Paket importieren:

```
import sound.effects.echo
```

Dieser Vorgang lädt das Untermodul `sound.effects.echo`. Es muss mit seinem kompletten Namen referenziert werden:

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Eine alternative Methode, um dieses Untermodul zu importieren:

```
from sound.effects import echo
```

Diese Variante lädt auch das Untermodul `echo`, macht es aber ohne seinen Paket-Präfix verfügbar. Man verwendet es folgendermaßen:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Eine weitere Möglichkeit ist, die beschriebene Funktion oder Variable direkt zu importieren:

```
from sound.effects.echo import echofilter
```

Genau wie in den anderen Beispielen, lädt dies das Untermodul `echo`. In diesem Fall wird aber die `echofilter()` Funktion direkt verfügbar gemacht:

```
echofilter(input, output, delay=0.7, atten=4)
```

Wenn man `from package import item` verwendet, kann das `item` entweder ein Untermodul und -paket sein oder ein Name, der in diesem Paket definiert ist (z. B. eine Funktion, eine Klasse oder Variable). Die `import`-Anweisung überprüft zuerst, ob das `item` in diesem Paket definiert ist; falls nicht, wird von einem Modul ausgegangen und versucht es zu laden. Wenn nichts gefunden wird, wird eine `ImportError`-Ausnahme geworfen.

Im Gegensatz dazu, muss bei Verwendung von `import item.subitem.subsubitem` jedes `item` ein Paket sein; das letzte `item` kann ein Modul oder ein Paket sein, aber es darf keine Klasse, Funktion oder Variable im darüber geordneten `item` sein.

* aus einem Paket importieren

Was passiert nun, wenn der Benutzer `from sound.effects import *` schreibt? Idealerweise würde man hoffen, dass dies irgendwie an das Dateisystem weitergereicht wird und alle Untermodule, die es im Paket gibt, findet und sie alle importiert. Das könnte sehr lange dauern und Untermodule zu importieren könnte Nebeneffekte hervorrufen, die nur dann auftreten sollten, wenn das Untermodul explizit importiert wird.

Die einzige Lösung ist, dass der Autor des Paketes einen expliziten Index des Paketes bereitstellt. Die `import`-Anweisung folgt folgender Konvention: Definiert die Datei `__init__.py` des Paketes eine Liste namens `__all__`,

wird diese als Liste der Modulnamen behandelt, die bei `from package import *` importiert werden sollen. Es ist Aufgabe des Paketautoren diese Liste aktuell zu halten, wenn er eine neue Version des Paketes veröffentlicht. Paketautoren können sich auch entscheiden es nicht zu unterstützen, wenn sie einen *-Import ihres Paketes für nutzlos halten. Zum Beispiel könnte die Datei `sound/effects/__init__.py` folgenden Code enthalten:

```
__all__ = ["echo", "surround", "reverse"]
```

Dies würde bedeuten, dass `from sound.effects import *` die drei genannten Untermodule des `sound` Paketes importiert.

Ist `__all__` nicht definiert, importiert die Anweisung `from sound.effects import *` *nicht* alle Untermodule des Paketes `sound.effects` in den aktuellen Namensraum; es stellt nur sicher, dass das Paket `sound.effects` importiert wurde (möglicherweise führt es jeglichen Initialisierungscode in `__init__.py` aus) und importiert dann alle Namen, die im Paket definiert wurden. Inklusive der Namen, die in `__init__.py` definiert werden (und Untermodule die explizit geladen werden). Es bindet auch jegliche Untermodule des Paketes ein, die durch vorherige `import`-Anweisungen explizit geladen wurden. Schau Dir mal diesen Code an:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

Hier werden die Module `echo` und `surround` in den aktuellen Namensraum importiert, da sie im Paket `sound.effects` definiert sind, wenn die Anweisung `from ... import` ausgeführt wird. (Das funktioniert auch wenn `__all__` definiert ist.)

Auch wenn manche Module so entworfen wurden, nur Namen, die einem bestimmten Namensschema folgen, bei einem `import *` zu exportieren, wird es dennoch als schlechte Praxis betrachtet.

Aber bedenke, dass an der Benutzung von `from Package import specific_submodule` nichts falsch ist! In der Tat ist es die empfohlene Schreibweise, es sei denn das importierende Modul benutzt gleichnamige Untermodule von anderen Paketen.

Referenzen innerhalb des Paketes

Werden Pakete in Unterpakete strukturiert (wie das `sound`-Paket im Beispiel), kann man absolute Importe benutzen um Untermodule der Geschwisterpakete zu referenzieren. Wenn das Modul `sound.filters.vocoder` beispielsweise das Modul `echo` im Paket `sound.effects` benutzen muss, kann es `from sound.effects import echo` nutzen.

Man kann auch relative Importe in der `from module import name`-Form der `import`-Anweisung nutzen. Diese Importe nutzen führende Punkte, um das aktuelle und die Elternpakete, die im relativen Import beteiligt sind, anzugeben. Aus dem `surround`-Modul heraus, könnte man beispielsweise folgendes nutzen:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Beachte, dass relative Importe auf dem Namen des aktuellen Moduls basieren. Da der Name des Hauptmoduls immer `"__main__"` ist, müssen Module, die als Hauptmodul einer Python-Anwendung gedacht sind, immer absolute Importe nutzen.

Pakete in mehreren Verzeichnissen

Pakete unterstützen ein weiteres besonderes Attribut: `__path__`. Es wird als Liste initialisiert, die den Namen des Verzeichnisses mit der `__init__.py` des Pakets enthält. Dies geschieht, bevor der Code in dieser Datei ausgeführt

wird. Diese Variable kann verändert werden und eine Änderung beeinflusst zukünftige Suchen nach Modulen und Unterpaketen, die im Paket enthalten sind.

Auch wenn dieses Feature nicht oft gebraucht wird, kann sie benutzt werden um die Menge der Module, die in einem Paket gefunden werden, zu erweitern.

KAPITEL 7

Eingabe und Ausgabe

Es gibt verschiedene Arten, die Ausgabe eines Programmes darzustellen: Daten können in menschenlesbarer Form ausgegeben werden oder in eine Datei für die spätere Verwendung geschrieben werden. Dieses Kapitel beschreibt einige Möglichkeiten.

Ausgefallenere Ausgabeformatierung

Bis jetzt sind uns zwei Arten der Ausgabe von Werten begegnet: *Ausdrucksanweisungen* (*expression statements*) und die `print()`-Funktion. (Eine dritte Möglichkeit ist die `write()`-Methode von Dateiobjekten; die Standardausgabedatei kann als `sys.stdout` referenziert werden. In der Bibliotheksreferenz gibt es dazu weitere Informationen.)

Oft will man mehr Kontrolle über die Formatierung der Ausgabe haben als nur Leerzeichen-getrennte Werte auszugeben. Es gibt zwei Arten die Ausgabe zu formatieren: Die erste Möglichkeit ist, dass man die gesamte Verarbeitung der Zeichenketten selbst übernimmt; indem man Slicing- und Verknüpfungsoperationen benutzt, kann man jede denkbare Anordnung zusammenstellen. Der Typ `string` hat einige Methoden, die ein paar nützliche Operationen ausführen, um Zeichenketten auf eine bestimmte Länge aufzufüllen; diese werden wir in Kürze behandeln. Die zweite Möglichkeit ist die Benutzung der `format()`-Methode.

Das `string`-Modul enthält eine Klasse `Template`, die noch einen Weg bietet, Werte in Zeichenketten zu ersetzen.

Eine Frage bleibt natürlich: Wie konvertiert man Werte zu Zeichenketten? Glücklicherweise kennt Python Wege, um jeden Wert in eine Zeichenkette umzuwandeln: Man übergibt den Wert an die `repr()`- oder `str()`-Funktion.

Die `str()`-Funktion ist dazu gedacht eine möglichst menschenlesbare Repräsentation des Wertes zurückzugeben, während `repr()` dazu gedacht ist, vom Interpreter lesbar zu sein (oder einen `SyntaxError` erzwingt, wenn es keine äquivalente Syntax gibt). Für Objekte, die keine besondere menschenlesbare Repräsentation haben, gibt `str()` denselben Wert wie `repr()` zurück. Viele Werte wie Nummern oder Strukturen wie Listen und Dictionaries benutzen für beide Funktionen dieselbe Repräsentation. Besonders Zeichenketten haben zwei verschiedene Repräsentationen.

Ein paar Beispiele:

```
>>> s = 'Hallo Welt!'
>>> str(s)
'Hallo Welt!'
```

```
>>> repr(s)
"'Hallo Welt!'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'Der Wert von x ist ' + repr(x) + ', und y ist ' + repr(y) + '...'
>>> print(s)
Der Wert von x ist 32.5, und y ist 40000...
>>> #repr() bei einer Zeichenkette benutzt Anführungszeichen und Backslashes:
... hello = 'Hallo Welt\n'
>>> hellos = repr(hello)
>>> print(hellos)
'Hallo Welt\n'
>>> # Das Argument für repr() kann jedes Pythonobjekt sein:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

Hier zwei Möglichkeiten, eine Tabelle von Quadrat- und Kubikzahlen zu erstellen:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Achte auf die Benutzung von 'end' in der vorherigen Zeile
...     print(repr(x*x*x).rjust(4))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000

>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(Beachte, dass im ersten Beispiel ein Leerzeichen pro Spalte durch die Funktionsweise von `print()` hinzugefügt wird: Sie trennt ihre Argumente mit Leerzeichen.)

Dieses Beispiel hat die `rjust()`-Methode von Zeichenkettenobjekten gezeigt, die eine Zeichenkette in einem Feld der gegebenen Breite rechtsbündig macht, indem sie diese links mit Leerzeichen auffüllt. Es gibt die ähnlichen Methoden `ljust()` und `center()`. Diese Methoden schreiben nichts, sondern geben eine neue Zeichenkette zurück. Ist die gegebene Zeichenkette zu lang, schneiden sie nichts, sondern geben diese unverändert zurück; dies wird die Anordnung durcheinanderbringen, aber ist meistens besser als die Alternative, dass der Wert verfälscht wird. (Will man

wirklich abschneiden, kann man immer noch eine Slicing-Operation hinzufügen, zum Beispiel `x.ljust(n)[:n]`.)

Es gibt noch eine weitere Methode, `zfill()`, die eine numerische Zeichenkette mit Nullen auffüllt. Sie versteht auch Plus- und Minuszeichen:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

Die einfachste Benutzung der `format()`-Methode sieht so aus:

```
>>> print('Wir sind die {}, die "{}!" sagen.'.format('Ritter', 'Ni'))
Wir sind die Ritter, die "Ni!" sagen.
```

Die Klammern und die Zeichen darin (genannt Formatfelder - *format fields*) werden mit den Objekten ersetzt, die der `format()`-Methode übergeben werden. Eine Nummer in den Klammern bezieht sich auf die Position des Objektes, die der `format()`-Methode übergeben werden.

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

Werden Schlüsselwortargumente in der `format()`-Methode benutzt, können deren Werte durch die Benutzung des Argumentnamens referenziert werden.

```
>>>print('Dieses {Speise} ist {Adjektiv}.'.format(Speise='Spam',
         Adjektiv='absolut schrecklich'))
Dieses Spam ist absolut schrecklich.
```

Positionsabhängige und Schlüsselwortargumente können willkürlich kombiniert werden:

```
>>>print('Die Geschichte von {0}, {1} und {anderer}.'.format('Bill',
         'Manfred', anderer='Georg'))
Die Geschichte von Bill, Manfred und Georg.
```

`'!a'` (wendet `ascii()` an), `'!s'` (wendet `str()` an) und `'!r'` (wendet `repr()` an) können dazu benutzt werden den übergebenen Wert zu konvertieren bevor er formatiert wird:

```
>>> import math
>>> print('Der Wert von PI ist ungefähr {}.'.format(math.pi))
Der Wert von PI ist ungefähr 3.14159265359.
>>> print('Der Wert von PI ist ungefähr {!r}.'.format(math.pi))
Der Wert von PI ist ungefähr 3.141592653589793.
```

Ein optionales `':'` mit Formatspezifizierer (*format specifier*) können auf den Namen des Feldes folgen. Dies erlaubt einem eine größere Kontrolle darüber, wie der Wert formatiert wird. Das folgende Beispiel rundet Pi auf drei Stellen nach dem Komma.

```
>>> import math
>>> print('Der Wert von Pi ist ungefähr {0:.3f}.'.format(math.pi))
Der Wert von Pi ist ungefähr 3.142.
```

Übergibt man einen Integer nach dem `':'`, so legt man eine minimale Breite für dieses Feld an. Das ist nützlich um Tabellen schön aussehen zu lassen.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print('{0:10} ==> {1:10d}'.format(name, phone))
...
Jack       ==>       4098
Dcab       ==>       7678
Sjoerd     ==>       4127
```

Hat man einen wirklich langen Formatstring, den man nicht aufteilen will, wäre es nett, wenn man die zu formatierenden Variablen durch den Namen statt durch die Position referenzieren könnte. Dies kann einfach bewerkstelligt werden, indem man das Dictionary übergibt und auf die Schlüssel über eckige Klammern `'[]'` zugreift

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
          'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Das könnte auch genauso erreicht werden, indem man die Tabelle als Schlüsselwortargumente mit der '**'-Notation übergibt.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Das ist besonders nützlich in Verbindung mit der eingebauten Funktion `vars()`, die ein Dictionary mit allen lokalen Variablen zurückgibt.

Format String Syntax gibt eine komplette Übersicht zur Zeichenkettenformatierung mit `format()`.

Alte Zeichenkettenformatierung

Der `%`-Operator kann auch zur Zeichenkettenformatierung genutzt werden. Er interpretiert das linke Argument genauso wie einen `sprintf()`-artigen Formatstring, der auf das rechte Argument angewendet werden soll und gibt die resultierende Zeichenkette dieser Formatierungsoperation zurück. Zum Beispiel:

```
>>> import math
>>> print('Der Wert von Pi ist ungefähr %5.3f.' % math.pi)
Der Wert von Pi ist ungefähr 3.142.
```

Da `format()` ziemlich neu ist, benutzt viel Pythoncode noch den `%`-Operator. Jedoch sollte `format()` hauptsächlich benutzt werden, da die alte Art der Formatierung irgendwann aus der Sprache entfernt werden wird.

Mehr Informationen dazu gibt es in dem Abschnitt Old String Formatting Operations.

Lesen und Schreiben von Dateien

`open()` gibt ein Dateiobjekt (file object) zurück und wird meistens mit zwei Argumenten aufgerufen: `open(filename, mode)`

```
>>> f = open('/tmp/workfile', 'w')


>>> print(f)
<open file '/tmp/workfile', mode 'w' at 80a0960>
```

Das erste Argument ist eine Zeichenkette, die den Dateinamen enthält. Das zweite Argument ist eine andere Zeichenkette mit ein paar Zeichen, die die Art der Benutzung der Datei beschreibt. *mode* kann `'r'` sein, wenn die Datei nur gelesen wird, `'w'`, wenn sie nur geschrieben wird (eine existierende Datei mit demselben Namen wird gelöscht) und `'a'` öffnet die Datei zum Anhängen; alle Daten, die in die Datei geschrieben werden, werden automatisch ans Ende angehängt. `'r+'` öffnet die Datei zum Lesen und Schreiben. Das *mode*-Argument ist optional, fehlt es, so wird `'r'` angenommen.

Normalerweise werden Dateien im *Textmodus* (*text mode*) geöffnet, das heisst, dass man Zeichenketten von ihr liest beziehungsweise in sie schreibt, die in einer bestimmten Kodierung kodiert werden (der Standard ist UTF-8). Wird `'b'` an das *mode*-Argument angehängt, so öffnet man die Datei im *Binärmodus* (*binary mode*); in ihm werden Daten als Byteobjekte gelesen und geschrieben. Dieser Modus sollte für alle Dateien genutzt werden, die keinen Text enthalten.

Im Textmodus wird beim Lesen standardmäßig das plattformspezifische Zeilenende (`\n` unter Unixen, `\r\n` unter Windows) zu einem einfachen `\n` konvertiert und beim Schreiben `\n` zurück zum plattformspezifischen Zeilenende. Diese versteckte Modifikation ist klasse für Textdateien, wird aber binäre Dateiformate, wie `JPEG`- oder `EXE`-Dateien, beschädigen. Achte sehr sorgfältig darauf, dass Du den Binärmodus benutzt, wenn Du solche Dateien schreibst oder liest.

Methoden von Dateiobjekten

Die übrigen Beispiele in diesem Abschnitt nehmen an, dass ein Dateiobjekt namens `f` schon erstellt wurde.

Um den Inhalt einer Datei zu lesen, kann man `f.read(size)` aufrufen, was einen Teil der Daten ausliest und diese als Zeichenketten- oder Byteobjekt zurückgibt. *size* ist ein optionales, numerisches Argument. Wird es ausgelassen oder ist es negativ, so wird der gesamte Inhalt der Datei ausgelesen und zurückgegeben, falls die Datei doppelt so groß wie der Speicher Deiner Maschine ist, so ist das Dein Problem. Andernfalls werden höchstens *size* Byte ausgelesen und zurückgegeben. Ist das Ende der Datei erreicht, so gibt `f.read()` eine leere Zeichenkette (`''`) zurück.

```
>>> f.read()
'Das ist die ganze Datei.\n'
>>> f.read()
''
```

`f.readline()` liest eine einzelne Zeile aus einer Datei; ein Zeilenumbruchszeichen (`\n`) bleibt am Ende der Zeichenkette und wird nur ausgelassen, falls die letzte Zeile nicht in einem Zeilenumbruch endet. Dies macht den Rückgabewert eindeutig: Falls `f.readline()` eine leere Zeichenkette zurückgibt, so ist das Ende der Datei erreicht, während eine Leerzeile durch `'\n'`, eine Zeichenkette, die nur einen einzelnen Zeilenumbruch enthält, dargestellt wird.

```
>>> f.readline()
'Dies ist die erste Zeile der Datei\n'
>>> f.readline()
'Zweite Zeile der Datei\n'
>>> f.readline()
''
```

`f.readlines()` gibt eine Liste zurück die alle Zeilen der Datei enthält. Wird ein optionaler Paramenter *sizehint* übergeben, liest es mindestens so viele Bytes aus der Datei und zusätzlich noch so viele, dass die nächste Zeile komplett ist und gibt diese Zeilen zurück. Dies wird oft benutzt, um ein effizientes Einlesen der Datei anhand der Zeilen zu ermöglichen, ohne die gesamte Datei in den Speicher laden zu müssen. Nur komplette Zeilen werden zurückgegeben.

```
>>> f.readlines()
['Dies ist die erste Zeile der Datei\n', 'Zweite Zeile der Datei\n']
```

Ein alternativer Ansatz Zeilen auszulesen ist, über das Dateiobjekt zu iterieren. Das ist speichereffizient, schnell und führt zu einfacherem Code:

```
>>> for line in f:
...     print(line, end='')
...
Dies ist die erste Zeile der Datei.
Zweite Zeile der Datei
```

Der alternative Ansatz ist einfacher, bietet aber keine feinkörnige Kontrolle. Da beide Ansätze die Pufferung von Zeilen unterschiedlich handhaben, sollten sie nicht vermischt werden.

`f.write(string)` schreibt den Inhalt von *string* in die Datei und gibt die Anzahl der Zeichen, die geschrieben wurden, zurück.

```
>>> f.write('Dies ist ein Test\n')
18
```

Um etwas anderes als eine Zeichenkette zu schreiben, muss es erst in eine Zeichenkette konvertiert werden:

```
>>> value = ('Die Antwort', 42)
>>> s = str(value)
>>> f.write(s)
19
```

`f.tell()` gibt eine Ganzzahl zurück, die die aktuelle Position des Dateiobjektes innerhalb der Datei angibt, gemessen in Bytes vom Anfang der Datei. Um die Position des Dateiobjektes zu ändern, gibt es `f.seek(offset, from_what)`. Die Position wird berechnet indem *offset* zu einem Referenzpunkt addiert wird, dieser wird durch das Argument *from_what* festgelegt. Bei einem *from_what* des Wertes 0, wird von Beginn der Datei gemessen, bei 1 von der aktuellen Position, bei 2 vom Ende der Datei. *from_what* kann ausgelassen werden und hat den Standardwert 0, das den Anfang der Datei als Referenzpunkt benutzt.

```
>>> f = open('/tmp/workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Gehe zum 6. Byte der Datei
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # Gehe zum drittletzten Byte
13
>>> f.read(1)
b'd'
```

In Textdateien (die, die ohne ein `b` im Modus geöffnet werden) sind nur Positionierungen vom Anfang der Datei aus erlaubt (mit der Ausnahme, dass mit `f.seek(0, 2)` zum Ende der Datei gesprungen werden kann).

Wenn man mit einer Datei fertig ist, ruft man `f.close()` auf, um sie zu schließen und jegliche Systemressource freizugeben, die von der offenen Datei belegt wird. Nach dem Aufruf von `f.close()` schlägt automatisch jeder Versuch fehl das Objekt zu benutzen.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

Die optimale Vorgehensweise ist es, das Schlüsselwort `with` zu benutzen, wenn man mit Dateiobjekten arbeitet. Das

hat den Vorteil, dass die Datei richtig geschlossen wird, sobald die Befehle des Blocks abgearbeitet sind, auch wenn innerhalb eine Ausnahme verursacht wird. Das ist auch viel kürzer als einen äquivalenten `try-finally`-Block zu schreiben:

```
>>> with open('/tmp/workfile', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

Dateiobjekte haben noch ein paar zusätzliche Methoden, wie `isatty()` und `truncate()`, die weniger häufig genutzt werden. Ein komplettes Handbuch zu Dateiobjekten kann in der Bibliotheksreferenz gefunden werden.

Das `pickle`-Modul

Zeichenketten können einfach in eine Datei geschrieben und aus ihr gelesen werden. Zahlen sind ein bisschen aufwändiger, da die `read()`-Methode nur Zeichenketten zurückgibt. Diese müssen an eine Funktion wie `int()` übergeben werden, die eine Zeichenkette wie `'123'` nimmt und deren numerischen Wert 123 zurückgibt. Wenn man jedoch komplexere Datentypen wie Listen, Dictionaries oder Klasseninstanzen speichern will, wird die Angelegenheit viel komplizierter.

Anstatt die Benutzer ständig Code schreiben und debuggen zu lassen, um komplexere Datentypen zu speichern, stellt Python ein Standardmodul namens `pickle` bereit. Dies ist ein fantastisches Modul, das fast jedes Pythonobjekt (sogar ein paar Formen von Pythoncode!) nehmen kann und es in eine Zeichenkettenrepräsentation konvertieren kann; dieser Prozess wird *pickling* ("einwecken") genannt. Das Objekt aus der Zeichenkettenrepräsentation zu rekonstruieren wird *unpickling* genannt. Zwischen pickling und unpickling, kann die Zeichenkettenrepräsentation in Daten oder Dateien gespeichert werden oder über ein Netzwerk an eine entfernte Maschine geschickt werden.

Hat man ein Objekt `x` und ein Dateiobjekt `f`, das zum Schreiben geöffnet wurde, benötigt der einfachste Weg das Objekt zu picklen nur eine Zeile Code:

```
pickle.dump(x, f)
```

Um das Objekt wieder zu unpicklen reicht, wenn `f` ein Dateiobjekt ist, das zum Lesen geöffnet wurde:

```
x = pickle.load(f)
```

(Es gibt auch andere Varianten, die benutzt werden, wenn man viele Objekte pickled oder falls man gepicklete Daten nicht in einer Datei speichern will; siehe `pickle` in der Python Bibliotheksreferenz.)

`pickle` ist der normale Weg ein Pythonobjekt zu erzeugen, das gespeichert und von anderen Programmen oder demselben Programm wiederbenutzt werden kann; der Fachbegriff für so etwas ist ein *persistentes* Objekt. Weil `pickle` so weitläufig benutzt wird, stellen viele Programmierer, die Pythonerweiterungen schreiben sicher, dass neue Datentypen, wie Matrizen, richtig gepickled und unpickled werden können.

KAPITEL 8

Fehler und Ausnahmen

Bis jetzt wurden Fehlermeldungen nur am Rande erwähnt, aber wenn Du die Beispiele ausprobiert hast, hast Du sicherlich schon einige gesehen. Es gibt (mindestens) zwei verschiedene Arten von Fehlern: *Syntaxfehler* (engl.: syntax errors) und *Ausnahmen* (engl.: exceptions).

Syntaxfehler

Syntaxfehler, auch Parser-Fehler genannt, sind vielleicht die häufigsten Fehlermeldungen, die Du bekommt, wenn Du Python lernst:

```
>>> while True print('Hallo Welt')
  File "<stdin>", line 1, in ?
    while True print('Hallo Welt')
                   ^
SyntaxError: invalid syntax
```

Der Parser wiederholt die störende Zeile und zeigt mit einem kleinen 'Pfeil' auf die Stelle, an der der Fehler entdeckt wurde. Der Fehler ist an dem Token aufgetreten (oder wurde zumindest dort entdeckt), welches *vor* dem Pfeil steht: In dem Beispiel wurde der Fehler bei der `print()`-Funktion entdeckt, da ein Doppelpunkt (`':'`) vor der Funktion fehlt. Des weiteren werden der Dateiname und die Zeilennummer ausgegeben, sodass Du weißt, wo Du suchen musst, falls die Eingabe aus einem Skript kam.

Ausnahmen

Selbst wenn eine Anweisung oder ein Ausdruck syntaktisch korrekt ist, kann es bei der Ausführung zu Fehlern kommen. Fehler, die bei der Ausführung auftreten, werden *Ausnahmen* (engl: exceptions) genannt und sind nicht notwendigerweise schwerwiegend: Du wirst gleich lernen, wie Du in Python-Programmen mit ihnen umgehst. Die meisten Ausnahmen werden von Programmen aber nicht behandelt, und erzeugen Fehlermeldungen, wie dieses Beispiel zeigt:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: int division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: Can't convert 'int' object to str implicitly
```

Die letzte Zeile der Fehlermeldung gibt an, was passiert ist. Es gibt verschiedene Typen von Ausnahmen, und der Typ der Ausnahme wird als Teil der Meldung ausgegeben: Die Typen in diesem Beispiel sind `ZeroDivisionError`, `NameError` und `TypeError`. Die Zeichenkette, die als Ausnahmetyp ausgegeben wird, ist der Name der eingebauten Ausnahme, die aufgetreten ist. Dies gilt für alle eingebauten Ausnahmen, muss aber für benutzerdefinierte Ausnahmen nicht zutreffen (es ist aber eine nützliche Konvention). Standard-Ausnahmen sind eingebaute Bezeichner (keine reservierten Schlüsselwörter).

Der Rest der Zeile gibt Details an, die auf dem Ausnahmetyp und darauf, was die Ausnahme ausgelöst hat, basieren.

Der vorangehende Teil der Fehlermeldung zeigt den Zusammenhang, in dem die Ausnahme auftrat, in Form eines Traceback. Im Allgemeinen wird dort der Programmverlauf mittels der entsprechenden Zeilen des Quellcodes aufgelistet; es werden jedoch keine Zeilen ausgegeben, die von der Standardeingabe gelesen wurden.

Built-in Exceptions listet alle eingebauten Ausnahmen und ihre Bedeutung auf.

Ausnahmen behandeln

Es ist möglich, Programme zu schreiben, welche ausgewählte Ausnahmen behandeln. Schau Dir das folgende Beispiel an, welches den Benutzer solange um Eingabe bittet, bis eine gültige Ganzzahl eingegeben wird, es dem Benutzer aber ermöglicht, das Programm abzubrechen (mit `Strg-C` oder was das Betriebssystem sonst unterstützt); ein solcher vom Benutzer erzeugter Abbruch löst eine `KeyboardInterrupt`-Ausnahme aus:

```
>>> while True:
...     try:
...         x = int(input("Bitte gib eine Zahl ein: "))
...         break
...     except ValueError:
...         print("Ups! Das war keine gültige Zahl. Versuche es noch einmal...")
...
```

Die `try`-Anweisung funktioniert folgendermaßen:

- Zuerst wird der *try-Block* (die Anweisung(en) zwischen den Schlüsselwörtern `try` und `except`) ausgeführt.
- Wenn dabei keine Ausnahme auftritt, wird der *except-Block* übersprungen, und die Ausführung der `try`-Anweisung ist beendet.
- Wenn während der Ausführung des try-Blocks eine Ausnahme auftritt, wird der Rest des Blockes übersprungen. Wenn dann der Typ dieser Ausnahme der Ausnahme gleicht, welche nach dem `except`-Schlüsselwort folgt, wird der except-Block ausgeführt, und danach ist die Ausführung der `try`-Anweisung beendet.
- Wenn eine Ausnahme auftritt, welche nicht der Ausnahme im except-Block gleicht, wird sie an äußere `try`-Anweisungen weitergegeben; wenn keine passende `try`-Anweisung gefunden wird, ist die Ausnahme eine

unbehandelte Ausnahme (engl: unhandled exception), und die Programmausführung stoppt mit einer Fehlermeldung wie oben gezeigt.

Eine `try`-Anweisung kann mehr als einen `except`-Block enthalten, um somit verschiedene Aktionen für verschiedene Ausnahmen festzulegen. Es wird höchstens ein except-Block ausgeführt. Ein Block kann nur die Ausnahmen behandeln, welche in dem zugehörigen try-Block aufgetreten sind, nicht jedoch solche, welche in einem anderen except-Block der gleichen try-Anweisung auftreten. Ein `except`-Block kann auch mehrere Ausnahmen gleichzeitig behandeln, dies wird in einem Tupel angegeben:

... except (RuntimeError, TypeError, NameError): ... pass

Der letzte except-Block kann ohne Ausnahme-Name(n) gelassen werden, dies fungiert als Wildcard. Benutze diese Möglichkeit nur sehr vorsichtig, denn dadurch können echte Programmierfehler verdeckt werden! Auf diese Weise kann man sich auch Fehlermeldungen ausgeben lassen und dann die Ausnahme erneut auslösen (sodass der Aufrufer diese Ausnahme ebenfalls behandeln kann):

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as err:
    print("I/O error: {0}".format(err))
except ValueError:
    print("Konnte Daten nicht in Ganzzahl umwandeln.")
except:
    print("Unbekannter Fehler:", sys.exc_info()[0])
    raise
```

Die `try` ... `except`-Anweisung erlaubt einen optionalen *else-Block*, welcher, wenn vorhanden, nach den except-Blöcken stehen muss. Er ist nützlich für Code, welcher ausgeführt werden soll, falls der try-Block keine Ausnahme auslöst. Zum Beispiel:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print('Kann', arg, 'nicht öffnen')
    else:
        print(arg, 'hat', len(f.readlines()), 'Zeilen')
        f.close()
```

Die Benutzung eines `else`-Blockes ist besser, als zusätzlichen Code zum `try`-Block hinzuzufügen. Sie verhindert, dass aus Versehen Ausnahmen abgefangen werden, die nicht von dem Code ausgelöst wurden, welcher von der `try` ... `except`-Anweisung geschützt werden soll.

Wenn eine Ausnahme auftritt, kann sie einen zugehörigen Wert haben, das sogenannte *Argument* der Ausnahme. Ob ein solches Argument vorhanden ist und welchen Typ es hat, hängt vom Typ der Ausnahme ab.

Der `except`-Block kann einen Variablennamen nach dem Ausnahme-Namen spezifizieren. Der Variablenname wird an eine Ausnahmeinstanz gebunden und die Ausnahme-Argumente werden in `instance.args` gespeichert. Für die bessere Benutzbarkeit definiert eine Ausnahmeinstanz `__str__()`, sodass die Argumente direkt ausgegeben werden können, ohne dass `.args` referenziert werden muss. Man kann außerdem eine Ausnahme instantiieren bevor man sie auslöst, um weitere Attribute nach Bedarf hinzuzufügen:

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
```

```
...     print(type(inst))     # Die Ausnahmeinstanz
...     print(inst.args)      # Argumente gespeichert in .args
...     print(inst)           # __str__ erlaubt direkte Ausgabe von .args,
...                           # kann aber in Subklassen überschrieben werden
...     x, y = inst.args      # args auspacken
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

Wenn eine Ausnahme Argumente hat, werden diese als letzter Teil ('detail') der Fehlermeldung unbehandelter Ausnahmen ausgegeben.

Ausnahme-Handler behandeln nicht nur Ausnahmen, welche direkt im `try`-Block auftreten, sondern auch solche Ausnahmen, die innerhalb von Funktionsaufrufen (auch indirekt) im `try`-Block ausgelöst werden. Zum Beispiel:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Behandle Laufzeitfehler:', err)
...
Behandle Laufzeitfehler: int division or modulo by zero
```

Ausnahmen auslösen

Die `raise`-Anweisung erlaubt es dem Programmierer, das Auslösen einer bestimmten Ausnahme zu erzwingen. Zum Beispiel:

```
>>> raise NameError('HeyDu')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HeyDu
```

Das einzige Argument des Schlüsselwortes `raise` gibt die Ausnahme an, die ausgelöst werden soll. Es muss entweder eine Ausnahme-Instanz sein oder eine Ausnahmeklasse (eine Klasse, die von `Exception` erbt).

Wenn Du herausfinden willst, ob eine Ausnahme ausgelöst wurde, sie aber nicht behandeln willst, erlaubt Dir eine einfachere Form der `raise`-Anweisung, eine Ausnahme erneut auszulösen:

```
>>> try:
...     raise NameError('HeyDu')
... except NameError:
...     print('Eine Ausnahme flog vorbei!')
...     raise
...
Eine Ausnahme flog vorbei!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HeyDu
```

Benutzerdefinierte Ausnahmen

Programme können ihre eigenen Ausnahmen benennen, indem sie eine neue Ausnahmeklasse erstellen (Unter *Klassen* gibt es mehr Informationen zu Python-Klassen). Ausnahmen sollten standardmäßig von der Klasse `Exception` erben, entweder direkt oder indirekt. Zum Beispiel:

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print('Meine Ausnahme wurde ausgelöst, Wert:', e.value)
...
Meine Ausnahme wurde ausgelöst, Wert:: 4
>>> raise MyError('ups!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'ups!'
```

In diesem Beispiel wurde die Methode `__init__()` der Klasse `Exception` überschrieben. Das neue Verhalten erzeugt schlicht das Attribut *value*, es ersetzt das Standardverhalten, ein Attribut *args* zu erzeugen.

Ausnahmeklassen können alle Möglichkeiten nutzen, die bei der Definition von Klassen zur Verfügung stehen, werden jedoch meist recht einfach gehalten; oft bieten sie nur eine Reihe von Attributen, welche genauere Informationen über den Fehler bereitstellen. Beim Erstellen von Modulen, welche verschiedene Fehler auslösen können, wird oft eine Basisklasse für Ausnahmen dieses Moduls definiert und alle anderen Ausnahmen für spezielle Fehlerfälle erben dann von dieser Basisklasse:

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        previous -- state at beginning of transition
```

```
        next -- attempted new state
        message -- explanation of why the specific transition is not allowed
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

Meistens gibt man den Ausnahmen Namen, die auf "Error" enden, ähnlich der Namensgebung der Standardausnahmen.

Viele Standardmodule definieren ihre eigenen Ausnahmen, um Fehler zu melden, die in ihren Funktionen auftreten können. Mehr Informationen über Klassen sind in Kapitel *Klassen* zu finden .

Aufräumaktionen festlegen

Die `try`-Anweisung kennt einen weiteren optionalen Block, der für Aufräumaktionen gedacht ist, die in jedem Fall ausgeführt werden sollen. Zum Beispiel:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Auf Wiedersehen, Welt!')
...
Auf Wiedersehen, Welt!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
```

Der *finally-Block* wird immer ausgeführt, bevor die `try`-Anweisung verlassen wird, egal ob eine Ausnahme aufgetreten ist oder nicht. Wenn eine Ausnahme im `try`-Block ausgelöst wurde, die nicht in einem except-Block behandelt wird (oder die in einem except-Block oder else-Block ausgelöst wurde), wird sie nach Ausführung des `finally`-Blocks erneut ausgelöst. Der `finally`-Block wird auch ausgeführt, wenn ein anderer Block der `try`-Anweisung durch eine `break`-, `continue`- or `return`-Anweisung verlassen wurde. Ein etwas komplizierteres Beispiel:

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("Division durch Null!")
...     else:
...         print("Ergebnis ist:", result)
...     finally:
...         print("Führe finally-Block aus")
...
>>> divide(2, 1)
Ergebnis ist: 2.0
Führe finally-Block aus
>>> divide(2, 0)
Division durch Null!
Führe finally-Block aus
>>> divide("2", "1")
Führe finally-Block aus
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Wie Du sehen kannst, wird der `finally`-Block in jedem Fall ausgeführt. Der `TypeError`, der durch die Division zweier Strings ausgelöst wird, wird nicht vom `except`-Block behandelt und wird somit erneut ausgelöst, nachdem der `finally`-Block ausgeführt wurde.

In echten Anwendungen ist der `finally`-Block nützlich, um externe Ressourcen freizugeben (wie Dateien oder Netzwerkverbindungen), unabhängig davon, ob die Ressource erfolgreich benutzt wurde oder nicht.

Vordefinierte Aufräumaktionen

Einige Objekte definieren Standard-Aufräumaktionen, die ausgeführte werden, wenn das Objekt nicht länger gebraucht wird, egal ob die Operation, die das Objekt benutzte, erfolgreich war oder nicht. Schau Dir das folgende Beispiel an, welches versucht, eine Datei zu öffnen und ihren Inhalt auf dem Bildschirm auszugeben.:

```
for line in open("myfile.txt"):
    print(line)
```

Das Problem dieses Codes ist, dass er die Datei, nachdem der Code ausgeführt wurde, für unbestimmte Zeit geöffnet lässt. In einfachen Skripten ist das kein Thema, aber in großen Anwendungen kann es zu einem Problem werden. Die `with`-Anweisung erlaubt es Objekten wie Dateien, auf eine Weise benutzt zu werden, dass sie stets korrekt und sofort aufgeräumt werden.

```
with open("myfile.txt") as f:
    for line in f:
        print(line)
```

Nachdem die Anweisung ausgeführt wurde, wird die Datei f stets geschlossen, selbst wenn ein Problem bei der Ausführung der Zeilen auftrat. Objekte die, wie Dateien, vordefinierte Aufräumaktionen bereitstellen, geben dies in ihrer Dokumentation an.

KAPITEL 9

Klassen

Verglichen mit anderen Programmiersprachen, fügt Pythons Klassenmechanismus Klassen mit einem Minimum an neuer Syntax und Semantik zur Sprache hinzu. Er ist eine Mischung der Klassenmechanismen von C++ und Modula-3. Python Klassen bieten alle Standardeigenschaften von objektorientierter Programmierung: Der Vererbungsmechanismus von Klassen erlaubt mehrere Basisklassen, eine abgeleitete Klasse kann jegliche Methoden seiner Basisklasse(n) überschreiben und eine Methode kann die Methode der Basisklasse mit demselben Namen aufrufen. Objekte können beliebig viele Mengen und Arten von Daten haben. Wie es auch bei Modulen der Fall ist, haben auch Klassen die dynamische Natur von Python inne: Sie werden zur Laufzeit erstellt und können auch nach der Erstellung verändert werden.

In der Terminologie von C++ sind *class members* (inklusive der *data member*) normalerweise in Python *public* (Ausnahmen siehe *Private Variablen*) und alle *member functions* (Methoden) sind *virtual*. Wie in Modula-3 gibt es keine Abkürzung zum referenzieren von Attributen eines Objekts aus dessen Methoden heraus: Die Methode wird mit einem expliziten ersten Argument, welches das Objekt repräsentiert, deklariert, und dann implizit beim Aufruf übergeben wird. Wie in Smalltalk sind Klassen selbst Objekte. Das bietet Semantiken zum importieren und umbenennen. Anders als in C++ und Modula-3 können eingebaute Datentypen vom Benutzer als Basisklassen benutzt, das heisst abgeleitet, werden. Außerdem können die meisten eingebauten Operatoren, wie in C++, mit einer besonderen Syntax (arithmetische Operatoren, Indizierung, usw.) für Instanzen der Klasse neu definiert werden.

(Da es keine allgemein anerkannte Terminologie im Bezug auf Klassen gibt, werde ich zwischendurch auf Smalltalk und C++ Begriffe ausweichen. Ich würde lieber Modula-3 Begriffe benutzen, da seine objektorientierte Semantik näher an Python, als an C++ ist, allerdings erwarte ich, dass wenige Leser davon gehört haben.)

Ein Wort zu Namen und Objekten

Objekte haben Individualität und mehrere Namen (in mehreren Gültigkeitsbereichen) können an dasselbe Objekt gebunden werden. In anderen Sprachen wird dies als *Aliasing* bezeichnet. Das wird meist beim ersten Blick auf Python nicht geschätzt und kann problemlos ignoriert werden, wenn man mit unveränderbaren Datentypen (Zahlen, Zeichenketten, Tupel) arbeitet. Aber Aliasing hat einen möglicherweise überraschenden Effekt auf die Semantik von Pythoncode, der veränderbare Objekte wie Listen, Dictionaries oder die meisten anderen Typen, enthält. Dies kommt normalerweise dem Programm zugute, da sich Aliase in mancher Hinsicht wie Pointer verhalten. Zum Beispiel ist die Übergabe eines Objekts günstig, da von der Implementierung nur ein Pointer übergeben wird. Verändert eine Funktion

ein Objekt, das als Argument übergeben wurde, wird der Aufrufende die Veränderung sehen — dies vermeidet den Bedarf an zwei verschiedenen Übergabemechanismen, wie in Pascal.

Gültigkeitsbereiche und Namensräume in Python

Bevor man Klassen überhaupt einführt, muss man über Pythons Regeln im Bezug auf Gültigkeitsbereiche reden. Klassendefinitionen wenden ein paar nette Kniffe bei Namensräumen an und man muss wissen wie Gültigkeitsbereiche und Namensräume funktionieren, um vollkommen zu verstehen was abläuft. Außerdem ist das Wissen hierüber nützlich für jeden fortgeschrittenen Pythonprogrammierer.

Fangen wir mit ein paar Definitionen an.

Ein *Namensraum* ist eine Zuordnung von Namen zu Objekten. Die meisten Namensräume sind momentan als Dictionaries implementiert, aber das ist normalerweise in keinerlei Hinsicht spürbar (außer bei der Performance) und kann sich in Zukunft ändern. Beispiele für Namensräume sind: Die Menge der eingebauten Namen (die Funktionen wie `abs()` und eingebaute Ausnahmen enthält), die globalen Namen eines Moduls und die lokalen Namen eines Funktionsaufrufs. In gewisser Hinsicht bilden auch die Attribute eines Objektes einen Namensraum. Das Wichtigste, das man über Namensräume wissen muss, ist, dass es absolut keinen Bezug von Namen in verschiedenen Namensräumen zueinander gibt. Zum Beispiel können zwei verschiedene Module eine Funktion namens `maximize` definieren, ohne dass es zu einer Verwechslung kommt, denn Benutzer des Moduls müssen dessen Namen voranstellen.

Nebenbei bemerkt: Ich benutze das Wort *Attribut* für jeden Namen nach einem Punkt — zum Beispiel in dem Ausdruck `z.real` ist `real` ein Attribut des Objekts `z`. Genau genommen sind auch Referenzen zu Namen in Modulen Attributreferenzen: Im Ausdruck `modname.funcname`, ist `modname` ein Modulobjekt und `funcname` ein Attribut dessen. In diesem Fall gibt es eine geradlinige Zuordnung von Modulattributen und globalen Namen, die im Modul definiert sind: Sie teilen sich denselben Namensraum![1]

Attribute können schreibgeschützt oder veränderbar sein. In letzterem Fall ist eine Zuweisung an dieses Attribut möglich. Modulattribute sind veränderbar: Man kann `modname.the_answer = 42` schreiben. Veränderbare Attribute sind gleichzeitig durch die `del`-Anweisung löschbar. Zum Beispiel löscht `del modname.the_answer` das Attribut `the_answer` des Objekts namens `modname`.

Namensräume werden zu verschiedenen Zeitpunkten erzeugt und haben verschiedene Lebenszeiten. Der Namensraum, der die eingebauten Namen enthält, wird beim Start des Interpreters erzeugt und nie gelöscht. Der globale Namensraum für ein Modul wird erzeugt, wenn die Moduldefinition eingelesen wird; normalerweise existieren die Namensräume des Moduls auch solange bis der Interpreter beendet wird. Die Anweisungen, die auf oberster Ebene vom Interpreter aufgerufen werden, entweder von einem Skript oder interaktiv gelesen, werden als Teil des Moduls `__main__` behandelt, sodass sie ihren eigenen globalen Namensraum haben. (Die eingebauten Namen existieren ebenfalls in einem Modul namens `builtins`.)

Der lokale Namensraum einer Funktion wird bei deren Aufruf erstellt und wird gelöscht, wenn sich die Funktion beendet oder eine Ausnahme auslöst, die nicht innerhalb der Funktion behandelt wird. (Eigentlich wäre "vergessen" eine bessere Beschreibung dessen, was passiert.) Natürlich haben auch rekursive Aufrufe ihren jeweiligen lokalen Namensraum.

Ein *Gültigkeitsbereich* (*scope*) ist eine Region eines Python-Programms, in der ein Namensraum direkt verfügbar ist, das heisst es einem unqualifiziertem Namen möglich ist einen Namen in diesem Namensraum zu finden.

Auch wenn Gültigkeitsbereiche statisch ermittelt werden, werden sie dynamisch benutzt. An einem beliebigen Zeitpunkt während der Ausführung, gibt es mindestens drei verschachtelte Gültigkeitsbereiche, deren Namensräume direkt verfügbar sind:

- Der innerste Gültigkeitsbereich, der zuerst durchsucht wird und die lokalen Namen enthält;

[1] Bis auf eine Ausnahme: Modulobjekte haben ein geheimes, schreibgeschütztes Attribut namens `__dict__`, das das Dictionary darstellt, mit dem der Namensraum des Modules implementiert wird; der Name `__dict__` ist ein Attribut, aber kein globaler Name. Offensichtlich ist dessen Benutzung eine Verletzung der Abstraktion der Namensraumimplementation und sollte deshalb auf Verwendungen wie die eines Post-Mortem-Debuggers reduziert werden.

- der Gültigkeitsbereich mit allen umgebenden Namensräumen (enthält auch die globalen Namen des momentanen Moduls), der vom nächsten umgebenden Namensraum aus durchsucht wird, und nicht-lokale, aber auch nicht-globale Namen enthält;
- der vorletzte Gültigkeitsbereich enthält die globalen Namen des aktuellen Moduls;
- der letzte Gültigkeitsbereich (zuletzt durchsuchte) ist der Namensraum, der die eingebauten Namen enthält.

Wird ein Name als `global` deklariert, so gehen alle Referenzen und Zuweisungen direkt an den mittleren Gültigkeitsbereich, der die globalen Namen des Moduls enthält. Um Variablen, die außerhalb des innersten Gültigkeitsbereichs zu finden sind, neu zu binden, kann die `nonlocal`-Anweisung benutzt werden. Falls diese nicht als `nonlocal` deklariert sind, sind diese Variablen schreibgeschützt (ein Versuch in diese Variablen zu schreiben, würde einfach eine *neue* lokale Variable im innersten Gültigkeitsbereich anlegen und die äußere Variable mit demselben Namen unverändert lassen).

Normalerweise referenziert der lokale Gültigkeitsbereich die lokalen Namen der momentanen Funktion. Außerhalb von Funktionen bezieht sich der lokale Gültigkeitsbereich auf denselben Namensraum wie der globale Gültigkeitsbereich: Den Namensraum des Moduls. Klassendefinitionen stellen einen weiteren Namensraum im lokalen Gültigkeitsbereich dar.

Es ist wichtig zu verstehen, dass die Gültigkeitsbereiche am Text ermittelt werden: Der globale Gültigkeitsbereich einer Funktion, die in einem Modul definiert wird, ist der Namensraum des Moduls, ganz egal wo die Funktion aufgerufen wird. Andererseits wird die tatsächliche Suche nach Namen dynamisch zur Laufzeit durchgeführt — jedoch entwickelt sich die Definition der Sprache hin zu einer statischen Namensauflösung zur Kompilierzeit, deshalb sollte man sich nicht auf die dynamische Namensauflösung verlassen! (In der Tat werden lokale Variablen schon statisch ermittelt.)

Eine besondere Eigenart Pythons ist, dass – wenn keine `global`-Anweisung aktiv ist – Zuweisungen an Namen immer im innersten Gültigkeitsbereich abgewickelt werden. Zuweisungen kopieren keine Daten, sondern binden nur Namen an Objekte. Das gleiche gilt für Löschungen: Die Anweisung `del x` entfernt nur die Bindung von `x` aus dem Namensraum des lokalen Gültigkeitsbereichs. In der Tat benutzen alle Operationen, die neue Namen einführen, den lokalen Gültigkeitsbereich: Im Besonderen binden `import`-Anweisungen und Funktionsdefinitionen das Modul beziehungsweise den Funktionsnamen im lokalen Gültigkeitsbereich.

Die `global`-Anweisung kann benutzt werden, um anzuzeigen, dass bestimmte Variablen im globalen Gültigkeitsbereich existieren und hier neu gebunden werden sollen. Die `nonlocal`-Anweisung zeigt an, dass eine bestimmte Variable im umgebenden Gültigkeitsbereich existiert und hier neu gebunden werden soll.

Beispiel zu Gültigkeitsbereichen und Namensräumen

Dies ist ein Beispiel, das zeigt, wie man die verschiedenen Gültigkeitsbereiche und Namensräume referenziert und wie `global` und :keyword‘nonlocal‘ die Variablenbindung beeinflussen:

```
def scope_test():
    def do_local():
        spam = "local spam"
    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"
    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("Nach der lokalen Zuweisung:", spam)
    do_nonlocal()
    print("Nach der nonlocal Zuweisung:", spam)
    do_global()
```

```
    print("Nach der global Zuweisung:", spam)

scope_test()
print("Im globalen Gültigkeitsbereich:", spam)
```

Die Ausgabe des Beispielcodes ist:

```
.. code-block:: none

 Nach der lokalen Zuweisung: test spam
 Nach der nonlocal Zuweisung: nonlocal spam
 Nach der global Zuweisung: nonlocal spam
 Im globalen Gültigkeitsbereich: global spam
```

Beachte, dass die *lokale* Zuweisung (was der Standard ist) die Bindung von *spam* in *scope_test* nicht verändert hat. Die `nonlocal` Zuweisung die Bindung von *spam* in *scope_test* und die `global` Zuweisung die Bindung auf Modulebene verändert hat.

Man kann außerdem sehen, dass es keine vorherige Bindung von *spam* vor der `global` Zuweisung gab.

Eine erste Betrachtung von Klassen

Klassen führen ein kleines bisschen neue Syntax, drei neue Objekttypen und ein wenig neue Semantik ein.

Syntax der Klassendefinition

Die einfachste Form einer Klassendefinition sieht so aus:

```
class ClassName:
    <anweisung-1>
    .
    .
    .
    <anweisung-N>
```

Klassendefinitionen müssen wie Funktionsdefinitionen (`def`-Anweisungen) ausgeführt werden, bevor sie irgendwelche Auswirkungen haben. (Es wäre vorstellbar eine Klassendefinition in einen Zweig einer `if`-Anweisung oder in eine Funktion zu platzieren.)

In der Praxis sind die Anweisungen innerhalb einer Klassendefinition üblicherweise Funktionsdefinitionen, aber andere Anweisungen sind erlaubt und manchmal nützlich — dazu kommen wir später noch. Die Funktionsdefinitionen innerhalb einer Klasse haben normalerweise eine besondere Argumentliste, die von den Aufrufkonventionen für Methoden vorgeschrieben wird — das wird wiederum später erklärt.

Wird eine Klassendefinition betreten, wird ein neuer Namensraum erzeugt und als lokaler Gültigkeitsbereich benutzt — deshalb werden Zuweisungen an lokale Variablen in diesem neuen Namensraum wirksam. Funktionsdefinitionen binden den Namen der neuen Funktion ebenfalls dort.

Wird eine Klassendefinition normal verlassen (indem sie endet), wird ein *Klassenobjekt* erstellt. Dies ist im Grunde eine Verpackung um den Inhalt des Namensraums, der von der Klassendefinition erstellt wurde. Im nächsten Abschnitt lernen wir mehr darüber. Der ursprüngliche lokale Gültigkeitsbereich (der vor dem Betreten der Klassendefinition aktiv war) wird wiederhergestellt und das Klassenobjekt wird in ihm an den Namen, der im Kopf der Klassendefinition angegeben wurde, gebunden (`ClassName` in unserem Beispiel).

Klassenobjekte

Klassenobjekte unterstützen zwei Arten von Operationen: Attributreferenzierungen und Instanziierung.

Attributreferenzierungen benutzen die normale Syntax, die für alle Attributreferenzen in Python benutzt werden: `obj.name`. Gültige Attribute sind alle Namen, die bei der Erzeugung des Klassenobjektes im Namensraum der Klasse waren. Wenn die Klassendefinition also so aussah:

```
class MyClass:
    """A simple example class"""
    i = 12345
    def f(self):
        return 'Hallo Welt'
```

dann sind `MyClass.i` und `MyClass.f` gültige Attributreferenzen, die eine Ganzzahl beziehungsweise ein Funktionsobjekt zurückgeben. Zuweisungen an Klassenattribute sind ebenfalls möglich, sodass man den Wert von `MyClass.i` durch Zuweisung verändern kann. `__doc__` ist ebenfalls ein gültiges Attribut, das den Docstring, der zur Klasse gehört, enthält: `Ä simple example class"`.

Klassen *Instanziierung* benutzt die Funktionsnotation. Tu einfach so, als ob das Klassenobjekt eine parameterlose Funktion wäre, die eine neue Instanz der Klasse zurückgibt. Zum Beispiel (im Fall der obigen Klasse):

```
x = MyClass()
```

Dies erzeugt eine neue *Instanz* der Klasse und weist dieses Objekt der lokalen Variable `x` zu.

Die Instanziierungsoperation ("aufrufen" eines Klassenobjekts) erzeugt ein leeres Objekt. Viele Klassen haben es gerne Instanzobjekte, die auf einen spezifischen Anfangszustand angepasst wurden, zu erstellen. Deshalb kann eine Klasse eine spezielle Methode namens `__init__()`, wie folgt definieren:

```
def __init__(self):
    self.data = []
```

Definiert eine Klasse eine `__init__()`-Methode, ruft die Klasseninstanziierung automatisch `__init__()` für die neu erstellte Klasseninstanz auf. So kann in diesem Beispiel eine neue, initialisierte Instanz durch folgendes bekommen werden:

```
x = MyClass()
```

Natürlich kann die `__init__()`-Methode Argumente haben, um eine größere Flexibilität zu erreichen. In diesem Fall werden die, dem Klasseninstanziierungsoperator übergebenen Argumente an `__init__()` weitergereicht. Zum Beispiel:

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Instanzobjekte

Was können wir jetzt mit den Instanzobjekten tun? Die einzigen Operationen, die Instanzobjekte verstehen, sind Attributreferenzierungen. Es gibt zwei Arten gültiger Attribute: Datenattribute und Methoden.

Datenattribute entsprechen "Instanzvariablen" in Smalltalk und "data members" in C++. Datenattribute müssen nicht deklariert werden; wie lokale Variablen erwachen sie zum Leben, sobald ihnen zum ersten Mal etwas zugewiesen wird. Zum Beispiel wird folgender Code, unter der Annahme, dass `x` die Instanz von `MyClass` ist, die oben erstellt wurde, den Wert `16` ausgeben, ohne Spuren zu hinterlassen:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

Die andere Art von Instanzattribut ist die *Methode*. Eine Methode ist eine Funktion, die zu einem Objekt *gehört*. (In Python existiert der Begriff Methode nicht allein für Klasseninstanzen: Andere Objekttypen können genauso Methoden haben. Zum Beispiel haben Listenobjekte Methoden namens `append()`, `insert()`, `remove()`, `sort()`, und so weiter. Jedoch benutzen wir in der folgenden Diskussion den Begriff Methode ausschliesslich im Sinne von Methoden von Klasseninstanzobjekten, sofern nichts anderes angegeben ist.

Ob ein Attribut eine gültige Methode ist, hängt von der Klasse ab. Per Definition definieren alle Attribute, die ein Funktionsobjekt sind, ein entsprechendes Methodenobjekt für seine Instanz. Deshalb ist in unserem Beispiel `x.f` eine gültige Methodenreferenz, da `MyClass.f` eine Funktion ist, aber `x.i` ist keine, da `MyClass.i` es nicht ist. `x.f` ist aber nicht dasselbe wie `MyClass.f` — es ist ein *Methodenobjekt* und kein Funktionsobjekt.

Methodenobjekte

Üblicherweise wird eine Methode gemäß seiner Bindung aufgerufen:

```
x.f()
```

Im `MyClass` Beispiel wird dies die Zeichenkette `'Hallo Welt'` ausgeben. Jedoch ist es nicht notwendig eine Methode direkt aufzurufen: `x.f` ist ein Methodenobjekt und kann weg gespeichert werden und später wieder aufgerufen werden. Zum Beispiel:

```
xf = x.f
while True:
    print(xf())
```

Das wird bis zum Ende der Zeit `Hallo Welt` ausgeben.

Was passiert genau, wenn eine Methode aufgerufen wird? Du hast vielleicht bemerkt, dass `x.f()` oben ohne Argument aufgerufen wurde, obwohl in der Funktionsdefinition für `f()` ein Argument festgelegt wurde. Was ist mit diesem Argument passiert? Natürlich verursacht Python eine Ausnahme, wenn eine Funktion, die ein Argument benötigt ohne aufgerufen wird — auch wenn das Argument eigentlich gar nicht genutzt wird ...

Tatsächlich, wie du vielleicht schon erraten hast, ist die Besonderheit bei Methoden, dass das Objekt als erstes Argument der Funktion übergeben wird. In unserem Beispiel ist der Aufruf `x.f()` das genaue äquivalent von `MyClass.f(x)`. Im Allgemeinen ist der Aufruf einer Methode mit *n* Argumenten äquivalent zum Aufruf der entsprechenden Funktion mit einer Argumentliste, die durch das Einfügen des Objekts der Methode vor das erste Argument erzeugt wird.

Verstehst du immernoch nicht, wie Methoden funktionieren, hilft vielleicht ein Blick auf die Implementierung, um die Dinge zu klären. Wenn ein Instanzattribut referenziert wird, das kein Datenattribut ist, wird seine Klasse durchsucht. Bezeichnet der Name ein gültiges Klassenattribut, das eine Funktion ist, wird ein Methodenobjekt erzeugt, indem (Zeiger zu) Instanzobjekt und Funktionsobjekt zu einem abstrakten Objekt verschmolzen werden: Dies ist das Methodenobjekt. Wird das Methodenobjekt mit einer Argumentliste aufgerufen, wird es wieder entpackt, eine neue Argumentliste aus dem Instanzobjekt und der ursprünglichen Argumentliste erzeugt und das Funktionsobjekt mit dieser neuen Argumentliste aufgerufen.

Beiläufige Anmerkungen

Datenattribute überschreiben Methodenattribute desselben Namens. Um zufällige Namenskonflikte zu vermeiden, die zu schwer auffindbaren Fehlern in großen Programmen führen, ist es sinnvoll sich auf irgendeine Konvention zu verständigen, die das Risiko solcher Konflikte vermindern. Mögliche Konventionen beinhalten das Großschreiben von Methodennamen, das Voranstellen von kleinen eindeutigen Zeichenketten (vielleicht auch nur ein Unterstrich) bei Datenattributen oder das Benutzen von Verben bei Methodennamen und Nomen bei Datenattributen.

Datenattribute können von Methoden, genauso wie von normalen Benutzern ("clients") eines Objektes referenziert werden. In anderen Worten: Klassen sind nicht benutzbar, um reine abstrakte Datentypen ("abstract data types") zu implementieren. In Wirklichkeit, gibt es in Python keine Möglichkeit um Datenkapselung (*data hiding*) zu erzwingen — alles basiert auf Konventionen. (Auf der anderen Seite kann die Python-Implementierung, in C geschrieben, Implementationsdetails komplett verstecken und den Zugriff auf ein Objekt kontrollieren, wenn das nötig ist; das kann von in C geschriebenen Python-Erweiterungen ebenfalls benutzt werden.)

Clients sollten Datenattribute mit Bedacht nutzen, denn sie könnten Invarianten kaputt machen, die von Methoden verwaltet werden, indem sie auf deren Datenattributen herumtrampeln. Man sollte beachten, dass Clients zu ihrem eigenen Instanzobjekt Datenattribute hinzufügen können, ohne die Gültigkeit der Methoden zu gefährden, sofern Namenskonflikte vermieden werden — auch hier kann eine Bennenungskonvention viele Kopfschmerzen ersparen.

Es gibt keine Abkürzung, um Datenattribute (oder andere Methoden!) innerhalb von Methoden zu referenzieren. Meiner Meinung verhilft das Methoden zu besserer Lesbarkeit: Man läuft keine Gefahr, lokale und Instanzvariablen zu verwechseln, wenn man eine Methode überfliegt.

Oft wird das erste Argument einer Methode `self` genannt. Dies ist nichts anderes als eine Konvention: Der Name `self` hat absolut keine spezielle Bedeutung für Python. Aber beachte: Hälst du dich nicht an die Konvention, kann dein Code schwerer lesbar für andere Python-Programmierer sein und es ist auch vorstellbar, dass ein *Klassenbrowser* (*class browser*) sich auf diese Konvention verlässt.

Jedes Funktionsobjekt, das ein Klassenattribut ist, definiert eine Methode für Instanzen dieser Klasse. Es ist nicht nötig, dass die Funktionsdefinition im Text innerhalb der Klassendefinition ist: Die Zuweisung eines Funktionsobjektes an eine lokale Variable innerhalb der Klasse ist ebenfalls in Ordnung. Zum Beispiel:

```
# Funktionsdefintion außerhalb der Klasse
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'Hallo Welt'
    h = g
```

`f`, `g` und `h` sind jetzt alle Attribute der Klasse `C`, die Funktionsobjekte referenzieren und somit sind sie auch alle Methoden der Instanzen von `C` — `h` ist dabei gleichbedeutend mit `g`. Beachte aber, dass diese Praxis nur dazu dient einen Leser des Programms zu verwirren.

Methoden können auch andere Methoden aufrufen, indem sie das Methodenattribut des Arguments `self` benutzen:

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Methoden können globale Namen genauso wie normale Funktionen referenzieren. Der globale Gültigkeitsbereich der Methode ist das Modul, das die Klassendefinition enthält. (Eine Klasse selbst wird nie als globaler Gültigkeitsbereich benutzt.) Während man selten einen guten Grund dafür hat globale Daten zu benutzen, gibt es viele berechtigte Verwendungen des globalen Gültigkeitsbereichs: Zum einen können Funktionen und Module, die in den globalen Gültigkeitsbereich importiert werden, genauso wie Funktionen und Klassen die darin definiert werden, von der Methode benutzt werden. Normalerweise ist die Klasse, die die Methode enthält, selbst in diesem globalen Gültigkeitsbereich definiert und im nächsten Abschnitt werden wir ein paar gute Gründe entdecken, warum eine Methode die eigene Klasse referenzieren wollte.

Jeder Wert ist ein Objekt und hat deshalb eine *Klasse* (auch *type* genannt). Es wird als `Objekt.__class__` abgelegt.

Vererbung

Natürlich verdient ein Sprachmerkmal nicht den Namen "Klasse", wenn es nicht Vererbung unterstützt. Die Syntax für eine abgeleitete Klassendefinition sieht so aus:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Der Name `BaseClassName` muss innerhalb des Gültigkeitsbereichs, der die abgeleitete Klassendefinition enthält, definiert sein. Anstelle eines Basisklassennamens sind auch andere willkürliche Ausdrücke erlaubt. Dies kann beispielsweise nützlich sein, wenn die Basisklasse in einem anderen Modul definiert ist:

```
class DerivedClassName(modname.BaseClassName):
```

Die Ausführung einer abgeleiteten Klassendefinition läuft genauso wie bei einer Basisklasse ab. Bei der Erzeugung des Klassenobjekts, wird sich der Basisklasse erinnert. Dies wird zum Auflösen der Attributsreferenzen benutzt: Wird ein angefordertes Attribut nicht innerhalb der Klasse gefunden, so wird in der Basisklasse weitergesucht. Diese Regel wird rekursiv angewandt, wenn die Basisklasse selbst von einer anderen Klasse abgeleitet wird.

Es gibt nichts besonderes an der Instanziierung von abgeleiteten Klassen: `DerivedClassName` erzeugt eine neue Instanz der Klasse. Methodenreferenzen werden wie folgt aufgelöst: Das entsprechende Klassenattribut wird durchsucht, falls nötig bis zum Ende der Basisklassenkette hinab und die Methodenreferenz ist gültig, wenn es ein Funktionsobjekt bereithält.

Abgeleitete Klassen können Methoden ihrer Basisklassen überschreiben. Da Methoden keine besonderen Privilegien beim Aufrufen anderer Methoden desselben Objekts haben, kann eine Methode einer Basisklasse, die eine andere Methode, die in derselben Basisklasse definiert wird, aufruft, beim Aufruf einer Methode der abgeleiteten Klasse landen, die sie überschreibt. (Für C++-Programmierer: Alle Methoden in Python sind im Grunde `virtual`.)

Eine überschreibende Methode in einer abgeleiteten Klasse wird in der Tat eher die Methode der Basisklasse mit demselben Namen erweitern, statt einfach nur zu ersetzen. Es gibt einen einfachen Weg die Basisklassenmethode direkt aufzurufen: Einfach `BaseClassName.methodname(self, arguments)` aufrufen. Das ist gelegentlich auch für Clients nützlich. (Beachte, dass dies nur funktioniert, wenn die Basisklasse als `BaseClassName` im globalen Gültigkeitsbereich zugänglich ist.)

Python hat zwei eingebaute Funktionen, die mit Vererbung zusammenarbeiten:

- Man benutzt `isinstance()` um den Typ eines Objekts zu überprüfen: `isinstance(obj, int)` ist nur dann `True`, wenn `obj.__class__` vom Typ `int` oder einer davon abgeleiteten Klasse ist.

- Man benutzt `issubclass()` um Klassenvererbung zu überprüfen: `issubclass(bool, int)` ist `True`, da `bool` eine von `int` abgeleitete Klasse ist. Jedoch ist `issubclass(float, int)` `False`, da `float` keine von `int` abgeleitete Klasse ist.

Mehrfachvererbung

Python unterstützt auch eine Form der Mehrfachvererbung. Eine Klassendefinition mit mehreren Basisklassen sieht so aus:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Für die meisten Zwecke, im einfachsten Fall, kann man sich die Suche nach geerbten Attributen von einer Elternklasse so vorstellen: Zuerst in die Tiefe (*depth-first*), von links nach rechts (*left-to-right*), wobei nicht zweimal in derselben Klasse gesucht wird, wenn sich die Klassenhierarchie dort überlappt. Deshalb wird, wenn ein Attribut nicht in `DerivedClassName` gefunden wird, danach in `Base1` gesucht, dann (rekursiv) in den Basisklassen von `Base1` und wenn es dort nicht gefunden wurde, wird in `Base2` gesucht, und so weiter.

In Wirklichkeit ist es ein wenig komplexer als das, denn die Reihenfolge der Methodenauflösung (*method resolution order - MRO*) wird dynamisch verändert, um zusammenwirkende Aufrufe von `super()` zu unterstützen. Dieser Ansatz wird in manchen anderen Sprachen als *call-next-method* (Aufruf der nächsten Methode) bekannt und ist mächtiger als der `super`-Aufruf, den es in Sprachen mit einfacher Vererbung gibt.

Es ist nötig dynamisch zu ordnen, da alle Fälle von Mehrfachvererbung eine oder mehrere Diamantbeziehungen aufweisen (bei der auf mindestens eine der Elternklassen durch mehrere Pfade von der untersten Klasse aus zugegriffen werden kann). Zum Beispiel erben alle Klassen von `object` und so stellt jeder Fall von Mehrfachvererbung mehrere Wege bereit, um `object` zu erreichen. Um zu verhindern, dass auf die Basisklassen mehr als einmal zugegriffen werden kann, linearisiert der dynamische Algorithmus die Suchreihenfolge, sodass die Ordnung von links nach rechts, die in jeder Klasse festgelegt wird, jede Elternklasse nur einmal aufruft und zwar monoton (in der Bedeutung, dass eine Klasse geerbt werden kann, ohne das die Rangfolge seiner Eltern berührt wird). Zusammengenommen machen diese Eigenschaften es möglich verlässliche und erweiterbare Klassen mit Mehrfachvererbung zu entwerfen. Für Details, siehe http://www.python.org/download/releases/2.3/mro/.

Private Variablen

"Private" Instanzvariablen, die nur innerhalb des Objekts zugänglich sind, gibt es in Python nicht. Jedoch gibt es eine Konvention, die im meisten Python-Code befolgt wird: Ein Name, der mit einem Unterstrich beginnt (z.B. `_spam`) sollte als nicht-öffentlicher Teil der API behandelt werden (egal ob es eine Funktion, eine Methode oder ein Datenattribut ist). Es sollte als Implementierungsdetails behandelt werden, das sich unangekündigt ändern kann.

Da es eine sinnvolle Verwendung für klassen-private Attribute gibt, um Namenskonflikte mit Namen, die von Unterklassen definiert werden zu vermeiden, gibt es eine begrenzte Unterstützung für so einen Mechanismus: *name mangling* (Namensersetzung). Jeder Bezeichner der Form `__spam` (mindestens zwei führende Unterstriche, höchstens ein folgender) wird im Text durch `_classname__spam` ersetzt, wobei `classname` der Name der aktuellen Klasse (ohne eventuelle führende Unterstriche) ist. Die Ersetzung geschieht ohne Rücksicht auf die syntaktische Position des Bezeichners, sofern er innerhalb der Definition der Klasse steht.

Namensersetzung ist hilfreich, um Unterklassen zu ermöglichen Methoden zu überschreiben, ohne dabei Methodenaufrufe innerhalb der Klasse zu stören. Zum Beispiel:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private Kopie der ursprünglichen update() Methode

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # erstellt update() mit neuer Signatur
        # macht aber __init__() nicht kaputt
        for item in zip(keys, values):
            self.items_list.append(item)
```

Beachte, dass die Ersetzungsregeln vor allem dazu gedacht sind, Unfälle zu vermeiden; es ist immernoch möglich auf einen solchen als privat gekennzeichneten Namen von aussen zuzugreifen und ihn auch zu verändern. Das kann in manchen Umständen sogar nützlich sein, beispielsweise in einem Debugger.

Beachte, dass Code, der von `exec()` oder `eval()` ausgeführt wird, den Klassennamen der aufrufenden Klasse nicht als die aktuelle Klasse ansieht. Dies ähnelt dem Effekt der `global`-Anweisung, der ebenfalls sehr beschränkt auf den Code ist, der zusammen byte-kompiliert wird. Die gleiche Begrenzung gilt für `getattr()`, `setattr()` und `delattr()`, sowie den direkten Zugriff auf `__dict__`.

Kleinkram

Manchmal ist es nützlich einen Datentyp zu haben, der sich ähnlich dem `record` in Pascal oder dem “struct” in C verhält und ein Container für ein paar Daten ist. Hier bietet sich eine leere Klassendefinition an:

```
class Employee:
    pass

john = Employee() # Eine leere Arbeitnehmerakte anlegen

# Die Akte ausfüllen
john.name = 'John Doe'
john.dept = 'Computerraum'
john.salary = 1000
```

Einem Stück Python-Code, der einen bestimmten abstrakten Datentyp erwartet, kann stattdessen oft eine Klasse übergeben werden, die die Methoden dieses Datentyps emuliert. Wenn man zum Beispiel eine Funktion hat, die Daten aus einem Dateiobjekt formatiert, kann man eine Klasse mit den Methoden `read()` und `readline()` definieren, die die Daten stattdessen aus einem Zeichenkettenpuffer bekommt, und als Argument übergeben.

Methodenobjekte der Instanz haben auch Attribute: `m.__self__` ist das Instanzobjekt mit der Methode `m()` und `m.__func__` ist das entsprechende Funktionsobjekt der Methode.

Ausnahmen sind auch Klassen

Benutzerdefinierte Ausnahmen werden auch durch Klassen gekennzeichnet. Durch die Nutzung dieses Mechanismus ist es möglich erweiterbare Hierarchien von Ausnahmen zu erstellen.

Es gibt zwei neue (semantisch) gültige Varianten der `raise`-Anweisung:

```
raise Klasse

raise Instanz
```

In der ersten Variante muss `Class` eine Instanz von `type` oder einer davon abgeleiteten Klasse sein und ist eine Abkürzung für:

```
raise Klasse()
```

Die in einem `except`-Satz angegebene Klasse fängt Ausnahmen dann ab, wenn sie Instanzen derselben Klasse sind oder von dieser abgeleitet wurden, nicht jedoch andersrum — der mit einer abgeleiteten Klasse angegebene `except`-Satz fängt nicht die Basisklasse ab. Zum Beispiel gibt der folgende Code B, C, D in dieser Reihenfolge aus:

```
class B(Exception):
   pass
class C(B):
   pass
class D(C):
   pass

for cls in [B, C, D]:
   try:
       raise cls()
   except D:
       print("D")
   except C:
       print("C")
   except B:
       print("B")
```

Beachte, dass B, B, B ausgegeben wird, wenn man die Reihenfolge umdreht, das heisst zuerst `except B`, da der erste zutreffende `except`-Satz ausgelöst wird.

Wenn eine Fehlermeldung wegen einer unbehandelten Ausnahme ausgegeben wird, wird der Name der Klasse, danach ein Doppelpunkt und ein Leerzeichen und schliesslich die Instanz mit Hilfe der eingebauten Funktion `str()` zu einer Zeichenkette umgewandelt ausgegeben.

Iteratoren

Mittlerweile hast du wahrscheinlich bemerkt, dass man über die meisten Containerobjekte mit Hilfe von `for` iterieren kann:

```
for element in [1, 2, 3]:
   print(element)
for element in (1, 2, 3):
   print(element)
for key in {'eins':1, 'zwei':2}:
   print(key)
```

```
for char in "123":
   print(char)
for line in open("meinedatei.txt"):
   print(line)
```

Diese Art des Zugriffs ist klar, präzise und praktisch. Der Gebrauch von Iteratoren durchdringt und vereinheitlicht Python. Hinter den Kulissen ruft die `for`-Anweisung `iter()` für das Containerobjekt auf. Die Funktion gibt ein Iteratorobjekt zurück, das die Methode `__next__()` definiert, die auf die Elemente des Containers nacheinander zugreift. Gibt es keine Elemente mehr, verursacht `__next__()` eine `StopIteration`-Ausnahme, die der `for`-Schleife mitteilt, dass sie sich beenden soll. Man kann auch die `__next__()`-Methode mit Hilfe der eingebauten Funktion `next()` aufrufen. Folgendes Beispiel demonstriert, wie alles funktioniert.

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
   next(it)
StopIteration
```

Kennt man die Mechanismen hinter dem Iterator-Protokoll, ist es einfach das Verhalten von Iteratoren eigenen Klassen hinzuzufügen. Man definiert eine `__iter__()`-Methode, die ein Objekt mit einer `__next__()`-Methode zurückgibt. Definiert die Klasse `__next__()`, kann `__iter__()` einfach `self` zurückgeben:

```
class Reverse:
   """Iterator for looping over a sequence backwards."""
   def __init__(self, data):
       self.data = data
       self.index = len(data)
   def __iter__(self):
       return self
   def __next__(self):
       if self.index == 0:
           raise StopIteration
       self.index = self.index - 1
       return self.data[self.index]
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

Generatoren

Generatoren (*generator*) sind eine einfache aber mächtige Möglichkeit um Iteratoren zu erzeugen. Generatoren werden wie normale Funktionen geschrieben, benutzen aber `yield`, um Daten zurückzugeben. Jedes Mal wenn `next()` aufgerufen wird, fährt der Generator an der Stelle fort, an der er zuletzt verlassen wurde (der Generator merkt sich dabei die Werte aller Variablen und welche Anweisung zuletzt ausgeführt wurde). Das nachfolgende Beispiel zeigt wie einfach die Erstellung von Generatoren ist:

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

Alles, was mit Generatoren möglich ist, kann ebenso (wie im vorigen Abschnitt dargestellt) mit Klassen-basierten Iteratoren, umgesetzt werden. Generatoren erlauben jedoch eine kompaktere Schreibweise, da die Methoden `__iter__()` und `__next__()` automatisch erstellt werden.

Des weiteren werden die lokalen Variablen und der Ausführungsstand automatisch zwischen den Aufrufen gespeichert. Das macht das Schreiben der Funktion einfacher und verständlicher als ein Ansatz, der mit Instanzvariablen wie `self.index` oder `self.data` arbeitet.

Generatoren werfen automatisch `StopIteration`, wenn sie terminieren. Zusammengenommen ermöglichen diese Features die Erstellung von Iteratoren mit einem Aufwand, der nicht größer als die Erstellung einer normalen Funktion ist.

Generator Ausdrücke

Manche einfachen Generatoren können prägnant als Ausdrücke mit Hilfe einer Syntax geschrieben werden, die der von List Comprehensions ähnlich ist, jedoch mit runden, statt eckigen Klammern. Diese Ausdrücke sind für Situationen gedacht, in denen der Generator gleich von der umgebenden Funktion genutzt wird. Generator Ausdrücke sind kompakter, aber auch nicht so flexibel wie ganze Generatordefinitionen und neigen dazu speicherschonender als die entsprechenden List Comprehensions zu sein.

Beispiele:

```
>>> sum(i*i for i in range(10))                 # Summe der Quadrate
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # Skalarprodukt
260

>>> from math import pi, sin
>>> sine_table = {x: sin(x*pi/180) for x in range(0, 91)}

>>> unique_words = set(word for line in page for word in line.split())
```

```
>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

KAPITEL 10

Eine kurze Einführung in die Standardbibliothek

Schnittstelle zum Betriebssystem

Im Modul `os` findet man Dutzende von Funktionen, um mit dem Betriebssystem zu interagieren:

```
>>> import os
>>> os.getcwd()                      # das aktuelle Arbeitsverzeichnis
'C:\\Python33'
>>> os.chdir('/server/accesslogs')   # Arbeitsverzeichnis verändern
>>> os.system('mkdir today') # Kommando in der Systemshell ausführen
>>> 0
```

Dabei sollte unbedingt beachtet werden, `import os` statt `from os import *` zu verwenden, da ansonsten `os.open()` die eingebaute Funktion `open()` überschreibt, die sich vollkommen anders verhält.

Die eingebauten Funktionen `dir()` und `help()` sind gerade beim Gebrauch der interaktiven Python-Shell nützlich, wenn man mit großen Modulen wie `os` arbeitet:

```
>>> import os
>>> dir(os)
<eine Liste aller Funktionen des Moduls>
>>> help(os)
<eine ausführliche Anleitung, erstellt aus den Docstrings des Moduls>
```

Typische Arbeiten mit Dateien und Verzeichnissen erleichtert das Modul `shutil`, das eine einfachere Schnittstelle bereitstellt.:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
>>> shutil.move('/build/executables', 'installdir')
```

Platzhalter in Dateinamen

Mit dem Modul `glob` können Platzhalter bei der Suche nach Datei- oder Verzeichnisnamen verwendet werden:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

Argumente in der Befehlszeile

Die meisten Skripte müssen Argumente aus der Befehlszeile verarbeiten. Diese Argumente werden als Liste im Attribut *argv* des Moduls `sys` gespeichert. Mit folgendem, als `demo.py` gespeicherten Skript:

```
import sys
print sys.argv
```

erhält man die folgende Ausgabe, wenn man `python demo.py eins zwei drei` in der Befehlszeile des Betriebssystems eingibt:

```
['demo.py', 'eins', 'zwei', 'drei']
```

Das Modul `getopt` verarbeitet *sys.argv* nach den üblichen Konventionen der Unixfunktion `getopt()`. Erweiterte und flexiblere Möglichkeiten bietet das Modul `argparse`.

Umleitung von Fehlermeldungen und Programmabbruch

Das Modul `sys` hat darüber hinaus auch Attribute für *stdin*, *stdout* und *stderr*. Letzteres ist vor allem bei der Ausgabe von Warnungen und Fehlermeldungen nützlich, etwa wenn *stdout* umgeleitet worden ist:

```
>>> sys.stderr.write('Warnung, Log-Datei konnte nicht gefunden werden\n')
Warnung, Log-Datei konnte nicht gefunden werden
```

Der direkteste Weg, ein Skript zu beenden, führt über `sys.exit()`.

Muster in Zeichenketten

Das Modul `re` erlaubt die Arbeit mit regulären Ausdrücken (*regular expressions*) für komplexe Zeichenketten-Operationen. Reguläre Ausdrücke eignen sich vor allem für komplizierte Suchen und Änderungen an Zeichenketten:

```
>>> import re
>>> re.findall(r'\bk[a-z]*', 'drei kleine katzen')
['kleine', 'katzen']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'Die Katze im im Hut')
'Die Katze im Hut'
```

Solange allerdings nur einfache Änderungen vorgenommen werden müssen, sollte man eher zu den normalen Methoden der Zeichenketten greifen, da diese einfacher zu lesen und zu verstehen sind:

```
>>> 'Tee für zwo'.replace('zwo', 'zwei')
'Tee für zwei'
```

Mathematik

Das Modul `math` ermöglicht den Zugriff auf Funktionen der zugrundeliegenden C-Bibliothek für Fließkomma-Mathematik:

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

Mit dem Modul `random` lassen sich zufällige Auswahlen treffen:

```
>>> import random
>>> random.choice(['Apfel', 'Birne', 'Banane'])
'Apfel'
>>> random.sample(range(100), 10)   # Stichprobe
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # Zufällige Fließkommazahl
0.17970987693706186
>>> random.randrange(6)    # Zufällige Ganzzahl aus range(6)
4
```

Das SciPy-Projekt hat viele weitere Module für numerische Berechnungen.

Zugriff auf das Internet

Zum Zugriff auf das Internet und für die Arbeit mit Internetprotokollen stehen verschiedene Module bereit. Zwei der einfachsten sind `urllib.request` zum Herunterladen von Daten über URLs und `smtplib` zum Versand von E-Mails:

```
>>> from urllib.request import urlopen
>>> for line in urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...     line = line.decode('utf-8')  # die binären Daten zu Text dekodieren
...     if 'EST' in line or 'EDT' in line:  # Nach Eastern Time suchen
...         print(line)

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Nimm dich in Acht vor den Iden des März!
... """)
>>> server.quit()
```

(Anmerkung: Das zweite Beispiel benötigt einen Mailserver auf `localhost`.)

Datum und Uhrzeit

Das Modul `datetime` stellt Klassen sowohl für einfache als auch kompliziertere Arbeiten mit Datum und Uhrzeit bereit. Während das Rechnen mit Datum und Uhrzeit zwar unterstützt wird, liegt das Hauptaugenmerk der Implementierung auf Attributzugriffen für Ausgabeformatierung und -manipulation. Die Verwendung von Zeitzonen wird ebenfalls unterstützt.

```
>>> # Ein Datum lässt sich leicht aufbauen
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y ist ein %A am %d. Tag des %B.")
'12-02-03. 02 Dec 2003 ist ein Tuesday am 02. Tag des December.'

>>> # Mit dem Datum lässt sich rechnen
>>> geburtstag = date(1964, 7, 31)
>>> alter = jetzt - geburtstag
>>> alter.days
14368
```

Datenkompression

Die üblichen Dateiformate zur Archivierung und Kompression werden direkt in eigenen Modulen unterstützt. Darunter: `zlib`, `gzip`, `bz2`, `zipfile` und `tarfile`.

```
>>> import zlib
>>> s = b'Wenn Fliegen hinter Fliegen fliegen'
>>> len(s)
35
>>> t = zlib.compress(s)
>>> len(t)
31
>>> zlib.decompress(t)
b'Wenn Fliegen hinter Fliegen fliegen'
>>> zlib.crc32(s)
1048664767
```

Performancemessung

Viele Benutzer von Python interessieren sich sehr für die jeweiligen Geschwindigkeitsunterschiede verschiedener Lösungen für ein Problem. Python stellt hier ein Messinstrument zur Verfügung, mit dem diese Fragen beantwortet werden können.

Es könnte etwa verlockend sein, statt Argumente einfach gegeneinander auszutauschen, Tuple und ihr Verhalten beim *Packing/Unpacking* zu verwenden. Das Modul `timeit` zeigt schnell einen eher bescheidenen Geschwindigkeitsvorteil auf:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
```

```
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

Die Zeitmessung mit `timeit` bietet hohe Genauigkeit. Dahingegen lassen sich mit `profile` und `pstats` zeitkritische Bereiche in größeren Abschnitten von Programmcode auffinden.

Qualitätskontrolle

Ein Ansatz, um Software hoher Qualität zu entwickeln, ist es Tests für jede Funktion schreiben, die regelmäßig während des Entwicklungsprozesses ausgeführt werden.

Das Modul `doctest` durchsucht ein Modul nach Tests in seinen Docstrings und führt diese aus. Das Erstellen eines Tests ist sehr einfach, dazu muss lediglich ein typischer Aufruf der Funktion samt seiner Rückgaben in den Docstring der Funktion kopiert werden. Dadurch wird gleichzeitig die Dokumentation verbessert, da Benutzer direkt ein Beispiel mitgeliefert bekommen. Darüber hinaus lässt sich so sicherstellen, dass Code und Dokumentation auch nach Änderungen noch übereinstimmen:

```
def durchschnitt(values):
    """Berechnet das arithmetische Mittel aus einer Liste von Zahlen

    >>> print(durchschnitt([20, 30, 70]))
    40.0
    """
    return sum(values, 0.0) / len(values)

import doctest
doctest.testmod()   # Führt den Test automatisch durch
```

Das Modul `unittest` funktioniert nicht ganz so einfach, dafür lassen sich damit auch umfangreichere Tests erstellen, die dazu gedacht sind, in einer eigenen Datei verwaltet zu werden:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_durchschnitt(self):
        self.assertEqual(durchschnitt([20, 30, 70]), 40.0)
        self.assertEqual(round(durchschnitt([1, 5, 7]), 1), 4.3)
        self.assertRaises(ZeroDivisionError, durchschnitt, [])
        self.assertRaises(TypeError, durchschnitt, 20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

Batteries Included

Bei Python folgt der Philosophie “Batteries Included”. Am besten lässt sich das an den komplexen und robusten Möglichkeiten seiner größeren Pakete sehen. Ein paar Beispiele:

- Die Module `xmlrpc.client` and `xmlrpc.server` erleichtern Remote Procedure Calls (RPC) enorm. Trotz ihrer Namen ist allerdings keine direkte Kenntnis oder Handhabung von XML notwendig.
- Das Modul `email` ist eine Bibliothek zur Arbeit mit E-Mails, inklusive MIME und anderen RFC 2822-basierten Nachrichten. Anders als `smtplib` und `poplib`, mit denen Nachrichten versandt und empfangen werden kön-

nen, ist `email` dafür zuständig, komplizierte Nachrichten (einschließlich Anhänge) zu konstruieren oder zu analysieren. Weiterhin erleichtert es den Umgang mit im Internet verwendeten Encodings und den Headern.

- `xml.dom` und `xml.sax` halten eine robuste Unterstützung für dieses populäre Datenaustausch-Format bereit. Mit `csv` lässt sich in ein allgemein gebräuchliches Datenbankformat schreiben und daraus lesen. Diese Module erleichtern den Austausch von Daten zwischen Python und anderen Werkzeugen enorm.
- Zur Internationalisierung von Anwendungen stehen unter anderem die Module `gettext`, `locale` und `codecs` zur Verfügung.

KAPITEL 11

Eine kurze Einführung in die Standardbibliothek - Teil II

Dieser zweite Teil der Tour beschäftigt sich mit Modulen für Fortgeschrittene. Diese Module sind selten in kleinen Skripten zu finden.

Ausgabeformatierung

Das Modul `reprlib` stellt eine Variante von `repr()` zur Verfügung, die für die verkürzte Anzeige von großen oder tief verschachtelten Containern ausgelegt ist:

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidocious'))
"set(['a', 'c', 'd', 'e', 'f', 'g', ...])"
```

Das Modul `pprint` bietet ausgefeiltere Kontrollmöglichkeiten für die Ausgabe von eingebauten und benutzerdefinierten Objekten, so dass diese im Interpreter gelesen werden können. Falls das Ergebnis länger als eine Zeile ist, fügt der "pretty printer" Zeilenumbrüche und Einrückungen hinzu, um die Datenstruktur klarer zu zeigen:

```
>>> import pprint
>>> t = [[[['schwarz', 'cyan'], 'weiß', ['grün', 'rot']], [['magenta',
...     'gelb'], 'blau']]]
...
>>> pprint.pprint(t, width=30)
[[[['schwarz', 'cyan'],
   'weiß',
   ['grün', 'rot']],
  [['magenta', 'gelb'],
   'blau']]]
```

Das Modul `textwrap` formatiert Textabsätze so, dass sie einer vorgegebenen Bildschirmbreite entsprechen:

```
>>> import textwrap
>>> doc = """Die Methode wrap() verhält sich wie fill(), außer dass sie
... anstatt einer einzigen großen Zeichenkette mit Zeilenumbrüchen, eine
```

```
... Liste aus Zeichenketten zurückgibt um die umgebrochenen Zeilen
... voneinander zu trennen."""
...
>>> print(textwrap.fill(doc, width=40))
Die Methode wrap() verhält sich wie
fill(), außer dass sie anstatt einer
einzigen großen Zeichenkette mit
Zeilenumbrüchen, eine Liste aus
Zeichenketten zurückgibt um die
umgebrochenen Zeilen voneinander zu
trennen.
```

Das Modul `locale` greift auf eine Datenbank mit länderspezifischen Datenformaten zu. Das grouping-Attribut der format-Funktion von locale bietet eine einfache Möglichkeit um Zahlen mit Tausendertrennzeichen zu formatieren:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'de_DE.UTF-8')
'de_DE.UTF-8'
>>> conv = locale.localeconv()          # die Zeichenkonventionen holen
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1.234.567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                      conv['frac_digits'], x), grouping=True)
'€1.234.567,80'
```

Templating

Das Modul `string` enthält die vielseitige Klasse `Template`, die wegen ihrer vereinfachten Syntax zum verändern durch Endbenutzer geeignet ist. Das ermöglicht Anwendern ihre Anwendung anzupassen, ohne die Anwendung zu verändern.

Das Format benutzt Platzhalter, die aus `$` und einem gültigen Python-Bezeichner (alphanumerische Zeichen und Unterstriche) bestehen. Umgibt man einen Platzhalter mit geschweiften Klammern, können andere alphanumerische Zeichen ohne Leerzeichen dazwischen folgen. Schreibt man `$$`, erzeugt man ein einzelnes escaptes `$`:

```
>>> from string import Template
>>> t = Template('Die Bürger von ${village} schicken 10 € für $cause.')
>>> t.substitute(village='Hannover', cause='den Grabenfond')
'Die Bürger von Hannover schicken 10 € für den Grabenfond.'
```

Die Methode `substitute()` verursacht einen `KeyError`, wenn ein Platzhalter nicht von einem Dictionary oder einem Schlüsselwortargument bereitgestellt wird. Bei Serienbrief-artigen Anwendungen können die vom Benutzer bereitgestellten Daten lückenhaft sein und die Methode `safe_substitute()` ist hier deshalb passender — sie lässt Platzhalter unverändert, wenn Daten fehlen:

```
>>> t = Template('Bringe $item $owner zurück.')
>>> d = dict(item='die unbeladene Schwalbe')
>>> t.substitute(d)
Traceback (most recent call last):
  . . .
KeyError: 'owner'
>>> t.safe_substitute(d)
'Bringe die unbeladene Schwalbe $owner zurück.'
```

Unterklassen von Template können einen eigenen Begrenzer angeben. Zum Beispiel könnte ein Umbenennungswerkzeug für einen Fotobrowser das Prozentzeichen als Platzhalter für das aktuelle Datum, die Fotonummer oder das Dateiformat auswählen:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
>>> fmt = input('Umbenennungsschema (%d-Datum %n-Nummer %f-Format):  ')
Umbenennungsschema (%d-Datum %n-Nummer %f-Format):  Ashley_%n%f

>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print('{0} --> {1}'.format(filename, newname))

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

Eine andere Anwendungsmöglichkeit für Templates ist die Trennung von Programmlogik und den Details der Ausgabeformate. Dies ermöglicht es eigene Vorlagen für XML-Dateien, Klartextberichte und HTML Web-Berichte zu ersetzen.

Arbeit mit strukturierten binären Daten

Das Modul `struct` stellt die Funktionen `pack()` und `unpack()` bereit, mit denen strukturierte binäre Daten verarbeitet werden können. Das folgende Beispiel zeigt, wie die Headerinformationen aus einem ZIP-Archiv ausgelesen werden, ohne das `zipfile`-Modul zu benutzen. Die Pack Codes `"H"` und `Ï"` stellen zwei Byte respektive vier Byte lange unsigned Integers dar. Das Zeichen `"<"` bedeutet, dass damit Standardgrößen gemeint sind und in der “Little Endian”-Bytereihenfolge vorliegen:

```
import struct

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3):                      # zeige die ersten 3 Dateiheader
    start += 14
    fields = struct.unpack('<IIIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)

    start += extra_size + comp_size     # skip to the next header
```

Multi-threading

Threading ist eine Methode, um nicht unmittelbar voneinander abhängige Prozesse abzukoppeln. Threads können benutzt werden, um zu verhindern, dass Programme, die während Berechnungen Benutzereingaben akzeptieren, "hängen". Ein ähnlicher Verwendungzweck ist es, einen Thread für I/O und einen anderen für Berechnungen zu benutzen.

Dieser Code zeigt wie das `threading` Modul benutzt werden kann um Prozesse im Hintergrund ablaufen zu lassen, während das Hauptprogramm parallel dazu weiterläuft:

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile
    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Zippen im Hintergrund abgeschlossen:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('Das Hauptprogramm läuft inzwischen weiter.')

background.join()    # Warten bis sich der Thread beendet.
print('Das Hauptprogramm hat auf die Beendigung des Hintergrund-Prozesses
      gewartet.')
```

Das Hauptproblem von Programmen mit mehreren Threads ist die Koordination der Zugriffe auf gemeinsame Daten oder andere Ressourcen. Dafür bietet das threading Modul einige Synchronisationsmethoden wie Locks, Events, Condition Variables und Semaphoren an.

Der beste Weg ist es aber, allen Zugriff auf Ressourcen in einem Thread zu koordinieren. Das `queue` Modul wird benutzt, um die Anfragen von den anderen Threads in dieses zu bekommen. Programme die `Queue` Objekte als Kommunikation zwischen ihren Threads nutzen sind einfacher zu entwickeln, lesbarer und stabiler.

Logging

Das Modul `logging` ermöglicht ein ausführliches und flexibles Erstellen von Logfiles. Im einfachsten Fall werden Logs in eine Datei geschrieben oder an `sys.stderr` geschickt:

```
import logging
logging.debug('Debugging Information')
logging.info('Information')
logging.warning('Warnung:Datei %s nicht gefunden', 'server.conf')
logging.error('Fehler')
logging.critical('Kritischer Fehler!')
```

Die Ausgabe von Meldungen der Stufen *info* und *debug* wird standardmäßig unterdrückt; übrige Meldungen werden an `sys.stderr` geschickt. Darüber hinaus können Meldungen auch per E-Mail, über Datenpakete (UDP), Sockets (TCP) oder an einen HTTP-Server ausgeliefert werden. Filter können weiterhin entscheiden, worüber Meldungen ausgegeben werden - je nach Priorität: `DEBUG`, `INFO`, `WARNING`, `ERROR` und `CRITICAL`.

Das Logging-system kann entweder direkt mittels Python konfiguriert werden oder seine Konfiguration aus einer vom Benutzer definierbaren Konfigurationsdatei lesen, ohne dass dabei das Programm selbst geändert werden muss.

Weak References

Python bietet automatische Speicherverwaltung (Zählen von Referenzen für die meisten Objekte und *garbage collection*). Der Speicher wird kurz nachdem die letzte Referenz auf ein Objekt aufgelöst worden ist freigegeben.

Für die meisten Anwendungen funktioniert dieser Ansatz gut, gelegentlich kann es allerdings auch nötig werden, Objekte nur so lange vorzuhalten, wie sie an anderer Stelle noch verwendet werden. Das allein führt allerdings bereits dazu, dass eine Referenz auf das Objekt erstellt wird, die es permanent macht. Mit dem Modul `weakref` können Objekte vorgehalten werden, ohne eine Referenz zu erstellen. Wird das Objekt nicht länger gebraucht, wird es automatisch aus einer Tabelle mit so genannten *schwachen Referenzen* gelöscht und eine Rückruf-Funktion für *weakref*-Objekte wird aufgerufen. Dieser Mechanismus wird etwa verwendet, um Objekte zwischenzuspeichern, deren Erstellung besonders aufwändig ist:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...             self.value = value
...     def __repr__(self):
...             return str(self.value)
...
>>> a = A(10)                   # Eine Referenz erstellen
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a            # Erstellt keine Referenz
>>> d['primary']                # Klappt, falls Objekt noch vorhanden
10
>>> del a                       # Einzige Referenz löschen
>>> gc.collect()                # Garbage collector aufrufen
0
>>> d['primary']                # Eintrag wurde automatisch gelöscht
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                # Eintrag wurde automatisch gelöscht
  File "C:/python33/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

Werkzeuge zum Arbeiten mit Listen

Viele Datenstrukturen können mit dem eingebauten Listentyp dargestellt werden. Jedoch gibt es manchmal Bedarf für eine alternative Implementierung mit anderen Abstrichen was Leistung angeht.

Das Modul `array` stellt die Klasse `array` bereit, die sich wie eine Liste verhält, jedoch nur homogene Daten aufnimmt und diese kompakter speichert. Das folgende Beispiel zeigt ein `array` von Nummern, die als vorzeichenlose binäre Nummern der Länge 2 Byte (Typcode `"H"`) gespeichert werden, anstatt der bei Listen üblichen 16 Byte pro Python-Ganzzahlobjekt:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
```

```
>>> a[1:3]
array('H', [10, 700])
```

Das Modul `collections` stellt die Klasse `deque` bereit, das sich wie eine Liste verhält, aber an das schneller angehängt und schneller Werte von der linken Seite "gepopt" werden können, jedoch langsamer Werte in der Mitte nachschlägt. Sie ist gut dazu geeignet Schlangen (Queues) und Baumsuchen, die zuerst in der Breite suchen (breadth first tree searches):

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print("Handling", d.popleft())
Handling task1

unsearched = deque([starting_node])
def breadth_first_search(unsearched):
   node = unsearched.popleft()
   for m in gen_moves(node):
       if is_goal(m):
           return m
       unsearched.append(m)
```

Zusätzlich zu alternativen Implementierungen von Listen bietet die Bibliothek auch andere Werkzeuge wie das `bisect`-Modul an, das Funktionen zum verändern von sortierten Listen enthält:

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

Das `heapq`-Modul stellt Funktionen bereit, um Heaps auf der Basis von normalen Listen zu implementieren. Der niedrigste Wert wird immer an der Position Null gehalten. Das ist nützlich für Anwendungen, die wiederholt auf das kleinste Element zugreifen, aber nicht die komplette Liste sortieren wollen:

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                      # in Heapreihenfolge neu ordnen
>>> heappush(data, -5)                 # neuen Eintrag hinzufügen
>>> [heappop(data) for i in range(3)]  # die drei kleinsten Einträge holen
[-5, 0, 1]
```

Dezimale Fließkomma-Arithmetik

Das Modul `decimal` bietet den `Decimal`-Datentyp für dezimale Fließkomma-Arithmetik. Verglichen mit der eingebauten `float`- Implementierung von binären Fließkomma-Zahlen ist die Klasse besonders hilfreich für

- Finanzanwendungen und andere Gebiete, die eine exakte dezimale Repräsentation,
- Kontrolle über die Präzision,
- Kontrolle über die Rundung, um gesetzliche oder regulative Anforderungen zu erfüllen,
- das Tracking von signifikanten Dezimalstellen

erfordern, oder

- für Anwendungen bei denen der Benutzer erwartet, dass die Resultate den händischen Berechnungen entsprechen.

Die Berechnung einer 5% Steuer auf eine 70 Cent Telefonrechnung ergibt unterschiedliche Ergebnisse in dezimaler und binärer Fließkomma-Repräsentation. Der Unterschied wird signifikant, wenn die Ergebnisse auf den nächsten Cent gerundet werden:

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(.70 * 1.05, 2)
0.73
```

Das `Decimal` Ergebnis behält die Null am Ende, automatisch vierstellige Signifikanz aus den Faktoren mit zweistelliger Signifikanz folgernd. `Decimal` bildet die händische Mathematik nach und vermeidet Probleme, die auftreten, wenn binäre Fließkomma-Repräsentation dezimale Mengen nicht exakt repräsentieren können.

Die exakte Darstellung ermöglicht es der Klasse `Decimal` Modulo Berechnungen und Vergleiche auf Gleichheit durchzuführen, bei denen die binäre Fließkomma-Repräsentation untauglich ist:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> sum([0.1]*10) == 1.0
False
```

Das `decimal`-Modul ermöglicht Arithmetik mit so viel Genauigkeit, wie benötigt wird:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

KAPITEL 12

Wie geht's weiter?

Das Lesen dieses Tutorials hat hoffentlich dein Interesse verstärkt Python einzusetzen und bist nun bestimmt voller Eifer, "echte" Probleme mit Python zu lösen. Wo kann man sich nun weiter informieren?

Dieses Tutorial ist Teil der Python-Dokumentation. Einige andere Dokumente davon sind:

- The Python Standard Library: Dieses Handbuch enthält eine vollständige (aber trotzdem kompakte) Referenz zu den Typen, Funktionen und Modulen, die die Standardbibliothek enthält und die enthält *eine Menge* zusätzlichen Quelltextes. So gibt es beispielsweise Module zum Lesen von Unix-Mailboxen, Herunterladen von Dokumenten über HTTP, Erzeugen von Zufallszahlen, Verarbeiten von Kommandozeilen-Optionen, Schreiben von CGI-Programmen, Komprimieren von Daten und für viele weitere Dinge. Das Überfliegen der Referenz zur Standardbibliothek gibt einen Überblick über die verfügbaren Module.
- Installing Python Modules erklärt wie man externe, von anderen Python-Programmierern geschriebene, Module installiert.
- The Python Language Reference: Gibt eine detaillierte Beschreibung der Syntax und Semantik von Python. Dies ist eher schwer zu lesen, aber als komplette Einführung in die Sprache als solche sinnvoll.

Weitere Quellen zu Python:

- http://www.python.org: Die eigentliche Internet-Präsenz von Python. Hier findet man den Quelltext, die Dokumentation und Verweise zu weiteren Webseiten, die einen Bezug zu Python haben. Die Webseite wird aus einigen Regionen der Welt, wie beispielsweise Europa, Japan, und Australien gespiegelt; ein Spiegelserver ist unter Umständen schneller als die Hauptseite — abhängig von der geografischen Lage.
- http://docs.python.org: Schneller Zugriff auf die Python-Dokumentation.
- http://pypi.python.org: Der *Python Package Index* (PyPI), früher auch als Cheese Shop bezeichnet, ist ein Index für Python-Module, die von anderen Benutzern zum Download bereitgestellt wurden. Sobald du Quellcode veröffentlichst, kannst du ihn hier registrieren, damit andere Benutzer diesen leichter finden können.
- http://aspn.activestate.com/ASPN/Python/Cookbook/: Das Python Cookbook ist eine umfangreiche Sammlung von Code-Beispielen, größeren Modulen und nützlichen Scripts. Besonders bemerkenswerte Beiträge wurden in einem Buch mit dem Titel Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3) veröffentlicht.
- http://scipy.org: Das "Scientific Python" Projekt enthält Module für schnelle Array-Berechnungen und -Manipulationen und zusätzlich eine Menge Pakete für Dinge wie Lineare Algebra, Fourier-Transformationen,

nicht-lineare Solver, Verteilung von Zufallszahlen, statistische Analyse und so weiter

Bei Fragen und Problemen zu Python, kann man sich an das deutschsprachige Python-Forum auf http://www.python-forum.de/, die englischsprachige Newsgroup *comp.lang.python* oder die Mailingliste python-list@python.org wenden. Es existiert ein Gateway zwischen der Newsgroup und der Mailingliste, so dass Nachrichten, die an die eine Adresse geschickt werden, automatisch zu der anderen weitergeleitet werden. In der Newsgroup gibt es jeden Tag um die 120 Beiträge (zu Spitzenzeiten einige Hundert) an Fragen (und Antworten), Vorschlägen für neue Features und Ankündigungen von neuen Modulen. Vor dem Posten sollte man sich jedoch die Frequently Asked Questions (auch FAQ genannt) durchlesen oder in das Verzeichnis `Misc/` der Python Quelltext-Distribution schauen. Archive für die Mailingliste findet man unter http://mail.python.org/pipermail/. Die FAQ beantwortet viele Fragen, die immer wieder aufkommen.

KAPITEL 13

Interaktive Eingabe-Bearbeitung und Ersetzung des Verlaufs

Einige Versionen des Python-Interpreters unterstützen die Bearbeitung der aktuellen Zeile und die Ersetzung des Verlaufs ähnlich der Möglichkeiten die die Korn-Shell oder die GNU Bash-Shell bieten. Dies wird durch die Benutzung der GNU Readline-Bibliothek implementiert, die eine Bearbeitung im Emacs- oder vi-Stil ermöglicht. Da diese Bibliothek eine eigene Dokumentation hat, wird sie hier nicht neu geschrieben, aber die Grundlagen lassen sich einfach erklären. Die hier beschriebenen Möglichkeiten sind in der Unix- und Cygwin-Version des Interpreters optional vorhanden.

Dieses Kapitel beschäftigt sich weder mit den Bearbeitungsmöglichkeiten, die Mark Hammonds PythonWin-Paket bereitstellt, noch mit denen der auf Tk basierten Entwicklungsumgebung IDLE, die mit Python verteilt wird. Der Kommandozeilen-Rückruf, den es innerhalb der DOS-Boxen unter NT und ein paar anderen DOS- und Windows-Versionen gibt, ist auch eine ganz andere Angelegenheit.

Zeilenbearbeitung

Falls unterstützt, ist die Bearbeitung der Eingabezeile immer dann aktiv, wenn der Interpreter eine primäre oder sekundäre Eingabeaufforderung ausgibt. Die aktuelle Zeile kann mit den üblichen Emacs-Steuerzeichen bearbeitet werden. Die wichtigsten davon sind: `Strg-A` bewegt den Cursor zum Anfang der Zeile, `Strg-E` zum Ende. `Strg-B` bewegt ihn ein Zeichen nach links, `Strg-F` nach rechts. `Strg-K` löscht ("killt") den Rest der Zeile rechts vom Cursor, `Strg-Y` fügt die zuletzt gelöschten Zeichen wieder ein ("yankt"). `Strg-_` macht die zuletzt getätigte Änderung rückgängig und kann wiederholt werden, um sie mehrmals auszuführen.

Ersetzung des Verlaufs

Die Ersetzung des Verlaufs funktioniert so: Alle eingetippten, nicht-leeren Zeilen werden in einem Verlaufspuffer gespeichert. Wird eine neue Eingabeaufforderung ausgegeben, ist man am Ende dieses Puffers platziert. `Strg-P` bewegt einen eine Zeile im Puffer zurück, `Strg-N` eine Zeile vorwärts. Jede Zeile im Verlaufspuffer kann bearbeitet werden; ein Sternchen vor der Eingabeaufforderung markiert eine Zeile als geändert. Drückt man die `Enter`-Taste, wird die aktuelle Zeile an den Interpreter übergeben. `Strg-R` startet eine inkrementelle Rückwärtssuche, `Strg-S` eine Vorwärtssuche.

Tastenkombinationen

Die Tastenkombinationen und ein paar andere Parameter der Readline-Bibliothek können angepasst werden, indem man Befehle in eine Initialisierungsdatei namens `~/.inputrc` schreibt. Tastenkombinationen haben die Form

```
Tastenname: Funktionsname
```

oder

```
"Zeichenkette": Funktionsname
```

und Optionen können so verändert werden:

```
set Optionsname Wert
```

Zum Beispiel:

```
#Ich bevorzuge den Bearbeitungsstil von vi:
set editing-mode vi

#Auf einer einzelnen Zeile bearbeiten:
set horizontal-scroll-mode On

#Ein paar Tastenkombinationen verändern:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

Beachte, dass in Python die Standardkombination für `Tab` das Einfügen eines `Tab`-Zeichens ist, anstatt dem Readline-Standard, die Funktion zum vervollständigen von Dateinamen. Bestehst du aber darauf, kannst du das mit

```
Tab: complete
```

in deiner `~/.inputrc` überschreiben. (Aber natürlich erschwert das das Schreiben von eingerückten Fortsetzungszeilen, wenn man es gewöhnt ist, `Tab` dafür zu benutzen.)

Automatische Vervollständigung von Variablen- und Modulnamen ist optional verfügbar. Um sie im Interaktiven Modus des Interpreters zu aktivieren, füge folgendes in deine Startup-Datei[#]_ ein:

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

Dies bindet die `Tab`-Taste an die Vervollständigungsfunktion, tippt man sie also zweimal bekommt man Vorschläge zur Vervollständigung; die Funktion durchsucht die lokalen Variablen und die Namen in verfügbaren Module. Für Ausdrücke mit Punkten, wie `string.a`, wird sie den Ausdruck bis zum letzen `'.'` auswerten und dann Vervollständigungen aus den Attributen des sich ergebenden Objektes vorschlagen. Beachte, dass dies von der Anwendung definierten Code ausführen könnte, wenn ein Objekt mit einer `__getattr__``()`-Methode Teil des Ausdrucks ist.

Eine leistungsfähigere Startup-Datei könnte wie das Beispiel aussehen. Beachte, dass sie die Namen löscht, sobald sie nicht mehr benötigt werden; dies wird getan, da die Startup-Datei im selben Namensraum wie die interaktiven Befehle ausgeführt wird und das Entfernen der Namen Nebeneffekte in der interaktiven Umgebung vermeidet. Du könntest es nützlich finden manche der importierten Module, wie `os`, das in den meisten Interpreter-Sitzungen gebraucht wird, zu behalten.

```
# Add auto-completion and a stored history file of commands to your Python
# interactive interpreter. Requires Python 2.0+, readline. Autocomplete is
# bound to the Esc key by default (you can change it - see readline docs).
```

```
#
# Store the file in ~/.pystartup, and set an environment variable to point
# to it:  "export PYTHONSTARTUP=~/.pystartup" in bash.

import atexit
import os
import readline
import rlcompleter

historyPath = os.path.expanduser("~/.pyhistory")

def save_history(historyPath=historyPath):
   import readline
   readline.write_history_file(historyPath)

if os.path.exists(historyPath):
   readline.read_history_file(historyPath)

atexit.register(save_history)
del os, atexit, readline, rlcompleter, save_history, historyPath
```

Alternativen zum Interaktiven Interpreter

Diese Möglichkeiten sind ein enormer Schritt vorwärts verglichen mit früheren Versionen des Interpreters, aber ein paar Wünsche sind noch offen: Es wäre nett, wenn die richtige Einrückung bei Fortsetzungszeilen vorgeschlagen würde (der Parser weiss, ob eine Einrückung benötigt wird). Der Vervollständigungsmechanismus könnte die Symbolstabelle des Interpreters nutzen. Und ein Befehl zum Überprüfen von passenden Klammern, Anführungszeichen, usw. (oder sie sogar vorschlägt) wäre auch nützlich.

Ein alternativer erweiterter interaktiver Interpreter, der schon seit einer Weile verfügbar ist, ist IPython. Er bietet Tab Completion, Erkundung von Objekten und eine fortschrittliche Verwaltung der Befehls-Chronik. Er kann komplett angepasst werden und auch in andere Anwendungen eingebettet werden. Eine weitere ähnlich fortschrittliche interaktive Umgebung ist bpython.

KAPITEL 14

Fließkomma-Arithmetik: Probleme und Einschränkungen

Fließkommazahlen werden in der Hardware des Computers als Brüche mit der Basis 2 (binär) dargestellt. Beispielsweise hat folgende Dezimalzahl

```
0.125
```

den Wert 1/10 + 2/100 + 5/1000; die entsprechende Binärzahl

```
0.001
```

hat den Wert 0/2 + 0/4 + 1/8. Diese beiden Brüche haben einen identischen Wert, der einzige wirkliche Unterschied ist, dass der erste als Bruch zur Basis 10 und der zweite als Bruch zur Basis 2 geschrieben wurde.

Leider können die meisten Dezimalbrüche nicht exakt als Binärbruch dargestellt werden. Eine Konsequenz daraus ist, dass im Allgemeinen die als dezimale Fließkommazahlen eingegebenen Werte nur durch die binären Fließkommazahlen angenähert werden können, die eigentlich von dem Computer gespeichert werden.

Das Problem ist zunächst einfacher im Dezimalsystem zu verstehen. Nehmen wir beispielsweise den Bruch 1/3. Man kann ihn in Dezimaldarstellung folgendermaßen annähern

```
0.3
```

oder, besser

```
0.33
```

oder noch besser,

```
0.333
```

und so weiter. Egal wie viele Stellen man schreibt, dass Resultat wird niemals exakt 1/3, aber es wird sich stetig 1/3 annähern.

Äquivalent kann, egal wie viele Stellen mit der Basis 2 man verwendet, die Dezimalzahl 0.1 niemals exakt als Binärbruch dargestellt werden. Im Binärsystem ist 1/10 die periodische Binärzahl:

```
0.0001100110011001100110011001100110011001100110011...
```

Hält man nach einer endlichen Zahl Bits an, erhält man eine Annäherung. In den meisten Rechnern werden heute Fließkommazahlen als Binärbrüche, mit dem Zähler in den ersten 53 Bits (beginnend mit dem höchstwertigsten Bit), gefolgt von dem Nenner als Potenz von Zwei, dargestellt. Im Fall von 1/10 lautet der Binärbruch `3602879701896397 / 2 ** 55`, was in etwa, aber nicht exakt, dem echten Wert von 1/10 entspricht.

Viele Benutzer sind sich durch die angezeigten Werte der Problematik nicht bewusst. Python zeigt nur eine dezimale Annäherung an den echten Dezimalwert an, der im Rechner gespeichert wird. Wenn Python den echten Dezimalwert zur gespeicherten binären Annäherung an 0,1 anzeigen würde, müsste es folgendes anzeigen

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

Das sind mehr Stellen als den meisten Leuten lieb ist, deshalb hält Python die Anzahl der Stellen überschaubar, indem es stattdessen einen gerundeten Wert anzeigt

```
>>> 1 / 10
0.1
```

Zur Erinnerung - auch wenn der angezeigte Wert wie der exakte Wert von 1/10 aussieht - ist der eigentlich gespeicherte Wert, der nächstgelegene Binärbruch.

Interessanterweise gibt es viele verschiedene Dezimalzahlen welche die selbe beste Approximation durch einen Binärbruch haben. Beispielsweise werden `0.1`, `0.10000000000000001` und `0.1000000000000000055511151231257827021181583404541015625` alle mit `3602879701896397 / 2 ** 55` angenähert. Da all diese Dezimalwerte die selbe Approximation haben, könnte jeder von ihnen angezeigt werden und immer noch die Bedingung `eval(repr(x)) == x` erfüllen.

In der Vergangenheit wählten der Python-Prompt und die eingebaute Funktion `repr()` diejenige mit 17 signifikanten Stellen: `0.10000000000000001`. Seit Python 3.1 ist Python (auf den meisten Systemen) dazu fähig, die kürzeste Darstellung zu wählen und einfach `0.1` anzuzeigen.

Das Verhalten liegt in der Natur der Fließkomma-Darstellung im Binärsystem: es ist kein Fehler in Python und auch kein Fehler in deiner Software. Man sieht dieses Problem in allen Sprachen, die die Fließkomma-Darstellung der Hardware unterstützen (obwohl manche Sprachen den Unterschied nicht standardmäßig oder in allen Anzeigemodi *anzeigen*).

Für eine schönere Ausgabe kann man String Formatierung nutzen, um die Anzahl der ausgegebenen signifikanten Ziffern zu beschränken.

```
>>> format(math.pi, '.12g')  # 12 signifikante Stellen angeben
'3.14159265359'
```

```
>>> format(math.pi, '.2f')   # 2 Nachkommastellen angeben
'3.14'
```

Es ist wichtig sich zu verinnerlichen, dass dies in Wahrheit eine Illusion ist - man rundet einfach die *Darstellung* des echten Maschinenwertes.

Diese Illusion erzeugt unter Umständen eine weitere. Da beispielsweise 0.1 nicht exakt 1/10 ist, ist die Summe von dreimal 0.1 nicht exakt 0.3:

```
>>> .1 + .1 + .1 == .3
False
```

Da 0.1 außerdem nicht näher an den exakten Wert von 1/10 und 0.3 heranreichen kann hilft auch vorheriges Runden mit `round()` nichts:

```
>>> round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1)
False
```

Obwohl die Zahlen nicht besser an ihren gedachten exakten Wert angenähert werden können, kann die Funktion `round()` nützlich für das nachträgliche Runden, so dass die ungenauen Ergebnisse vergleichbar zueinander werden:

```
>>> round(.1 + .1 + .1, 1) == round(.3, 1)
True
```

Binäre Fließkommaarithmetik sorgt noch für einige Überraschungen wie diese. Das Problem mit "0.1" ist im Abschnitt "Darstellungsfehler" weiter unten detailliert beschrieben. Dazu sei auf The Perils of Floating Point für eine umfassendere Liste von üblichen Problemen verwiesen.

Wie schon dort gegen Ende des Textes gesagt wird: "Es gibt keine einfachen Antworten." Trotzdem sollte man nicht zögerlich bei dem Einsatz von Fließkommazahlen sein! Die Fehler in Python-Fließkommaoperationen sind Folgefehler der Fließkomma-Hardware und liegt auf den meisten Maschinen in einem Bereich der geringer als 1 zu 2**53 pro Operation ist. Das ist mehr als ausreichend für die meisten Anwendungen, aber man muss sich in Erinnerung halten das es sich nicht um Dezimal-Arithmetik handelt und dass jede Operation mit einer Fließkommazahl einen neuen Rundungsfehler enthalten kann.

Von einigen pathologischen Fällen abgesehen, erhält man in den meisten existierenden Fällen, für die gängigsten Anwendungen von Fließkommazahlen das erwartete Ergebnis, wenn man einfach die Anzeige des Ergebnisses auf die Zahl der Dezimalstellen rundet, die man erwartet. `str()` genügt meist, für eine feinere Kontrolle kann man sich `str.format()` mit den Formatierungsoptionen in Format String Syntax anschauen.

Für Anwendungsfälle, die eine exakte dezimale Darstellung benötigen, kann das Modul `decimal` verwendet werden, welches Dezimal-Arithmetik implementiert, die für Buchhaltung und andere Anwendungen, die eine hohe Präzision erfordern, geeignet ist.

Eine andere Form exakter Arithmetik wird von dem Modul `fractions` bereitgestellt, welche eine Arithmetik implementiert, die auf rationalen Zahlen basiert (so dass Zahlen wie 1/3 exakt abgebildet werden können).

Wenn man im größeren Umfang mit Fließkommazahlen zu tun hat, sollte man einen Blick auf Numerical Python und die vielen weitere Pakete für mathematische und statistische Operationen die vom SciPy-Projekt bereitgestellt werden anschauen.

Python verfügt außerdem über ein Werkzeug für die seltenen Fälle, in denen man *wirklich* den exakten Wert des floats wissen will. Die Methode `float.as_integer_ratio()` gibt den Wert der Fließkommazahl als Bruch zurück:

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

Da dieser Bruch exakt ist, kann er benutzt werden, um ohne Verluste den originalen Wert wiederherzustellen:

```
>>> x == 3537115888337719 / 1125899906842624
True
```

Die Metode `float.hex()` stellt die Fließkommazahl hexadezimal (Basis 16) dar und gibt ebenfalls den exakten im Rechner gespeicherten Wert zuück:

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

Diese präzise hexadezimale Darstellung kann benutzt werden, um den originalen Wert exakt wiederherzustellen:

```
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

Da diese Darstellung exakt ist, kann sie genutzt werden, um Daten zwischen verschiedenen Versionen von Python (plattformunabhängig) und zwischen verschiedenen anderen Sprachen, die dieses Format unterstützen (wie z.B. Java und C99), auszutauschen.

Ein weiteres hilfreiches Werkzeug ist die Funktion `math.fsum()`, welche den Genauigkeitsverlust beim Summieren verringert. Sie registriert die "verlorenen Ziffern" als Werte, die zu einer Summe addiert werden. Dies kann die Gesamtgenauigkeit dahingehend beeinflussen, dass die Fehler sich nicht zu einer Größe summieren, die das Endergebnis beeinflusst:

```
>>> sum([0.1] * 10) == 1.0
False
>>> math.fsum([0.1] * 10) == 1.0
True
```

Darstellungsfehler

Dieser Abschnitt erklärt das "0.1" Beispiel im Detail und zeigt wie man selbstständig eine exakte Analyse dieser Fälle durchführen kann. Ein grundlegendes Verständnis der binären Fließkomma-Darstellung wird vorausgesetzt.

Der Begriff *Darstellungsfehler* verweist auf den Umstand das manche (die meisten sogar) Dezimalbrüche nicht exakt als Binärbrüche (Basis 2) dargestellt werden können. Dies ist der Hauptgrund warum Python (oder Perl, C, C++, Java, Fortran, und viele andere) oft nicht das erwartete Ergebnis anzeigen.

Warum ist das so? 1/10 ist nicht exakt als Binärbruch darstellbar. Fast alle heutigen Rechner (November 2000) benutzen die IEEE-754 Fließkommaarithmetik und wie fast alle Plattformen, bildet Python floats als IEEE-754 "double precision" ab. IEEE-754 doubles sind auf 53 Bits genau, so dass sich der Computer bemüht, 0.1 mit einem Bruch der Form *J*/2***N* bestmöglich anzunähern, wobei *J* eine 53 Bit breite Ganzzahl ist. Schreibt man:

```
1 / 10 ~= J / (2**N)
```

als

```
J ~= 2**N / 10
```

und erinnert sich daran das *J* genau 53 Bit breit ist (d. h. `>= 2**52` und `< 2**53`), ergibt sich als bester Wert für *N* 56:

```
>>> 2**52 <=  2**56 // 10  < 2**53
True
```

Das heißt, 56 ist der einzige Wert für *N*, wenn *J* auf 53 Bits beschränkt ist. Der bestmögliche Wert für *J* ist dann der gerundete Quotient:

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

Da der Rest mehr als die Hälfte von 10 beträgt, wird die beste Annäherung durch Aufrunden ermittelt:

```
>>> q+1
7205759403792794
```

Aus diesem Grund ist die bestmögliche Approximation von 1/10 als "IEEE-754 double precision":

```
7205759403792794 / 2 ** 56
```

Kürzt man Zähler und Nenner mit 2, ergibt sich folgender Bruch:

```
3602879701896397 / 2 ** 55
```

Man beachte, dass, da aufgerundet wurde, dieser Wert in Wahrheit etwas größer ist als 1/10; hätte man nicht aufgerundet, wäre der Bruch ein wenig kleiner als 1/10. Aber in keinen Fall wäre er *exakt* 1/10!

Der Rechner bekommt also nie 1/10 zu *sehen*: Was er sieht, ist der exakte oben dargestellte Bruch, die beste "IEEE-754 double" Approximation, die es gibt:

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

Wenn dieser Bruch mit 10**55 multipliziert wird, kann man sich diesen Wert bis auf 55 Dezimalstellen anzeigen lassen:

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
1000000000000000055511151231257827021181583404541015625
```

was bedeutet, dass der exakte Wert der im Rechner gespeichert würde, in etwa dem Dezimalwert 0.1000000000000000055511151231257827021181583404541015625 entspricht. Anstatt den ganzen Dezimalwert anzuzeigen runden viele Sprachen (inklusive älterer Versionen von Python) das Ergebnis auf 17 signifikante Stellen:

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

Die Module `fractions` und `decimal` vereinfachen diese Rechnungen:

```
>>> from decimal import Decimal
>>> from fractions import Fraction

>>> Fraction.from_float(0.1)
Fraction(3602879701896397, 36028797018963968)

>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)

>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')

>>> format(Decimal.from_float(0.1), '.17')
'0.10000000000000001'
```

KAPITEL 15

Zur Übersetzung

Dies ist die Übersetzung des offiziellen Python-Tutorials der Version 3.3.

KAPITEL 16

Autoren der Übersetzung

- Florian Heinle <launchpad@planet-tiax.de>
- Florian Mayer <flormayer@aim.com>
- HerrHagen
- Johannes Schönberger
- Lutz Horn <lutz.horn@fastmail.fm>
- Michael Markert <markert.michael@gmail.com>
- Stefan Matthias Aust <eibaan@googlemail.com>

KAPITEL 17

Möglichkeiten der Mitarbeit

Dieses Tutorial wird über ein bei http://bitbucket.org gehostetes Mercurial-Repository übersetzt und betreut.

Details zur Mitarbeit finden sich im Wiki des Projektes.

Um Fehler oder Verbesserungen zu melden kann der Issue-Tracker benutzt werden.

Desweiteren gibt es den IRC-Channel #py-tutorial-de auf freenode.

KAPITEL 18

Glossar

>>> Der Standard-Prompt der interaktiven Python-Shell. Man begegnet ihm oft in Code-Beispielen die interaktiv im Interpreter ausgeführt werden können.

... Der Standard-Prompt der interaktiven Shell, wenn man Code für einen eingerückten Code-Block oder innerhalb eines Paars passender Klammern (rund, eckig oder geschweift) eingibt.

2to3 Ein Werkzeug, das versucht Python 2.x Code zu Python 3.x Code zu konvertieren, indem es die meisten Inkompabilitäten, die durch das Parsen der Quellen und das traversieren des Parse-Baumes erkannt werden können, behandelt.

2to3 ist in der Standardbibliothek als `lib2to3` verfügbar. Ein eigenständiger Einstiegspunkt ist als `Tools/scripts/2to3` bereitgestellt. Siehe 2to3 - Automated Python 2 to 3 code translation.

abstract base class Abstrakte Basisklassen (Abstract Base Classes - kurz ABCs) ergänzen *duck-typing*, indem sie das definieren von Schnittstellen ermöglichen, wo andere Techniken wie `hasattr()` umständlich wären. Python kommt mit vielen eingebauten ABCs für Datenstrukturen (im `collections`-Modul), Zahlen (im `numbers`-Modul) und Ströme (im `io`-Modul). Man kann eigene ABC mit dem `abc`-Modul erzeugen.

argument Ein Wert, der einer Funktion oder Methode übergeben wird und einer benannten lokalen Variable im Funktionsrumpf zugewiesen wird. Eine Funktion oder Methode kann sowohl Positions-, als auch Schlüsselwort-Argumente in ihrer Definition haben. Positions- und Schlüsselwort-Argumente können von variabler Länge sein: `*` akzeptiert (falls in der Funktionsdefinition) oder übergibt (im Funktionsaufruf) mehrere Positionsargumente in einer Liste, während `**` dasselbe für Schlüsselwort-Argumente in einem Dictionary leistet.

Jeder Ausdruck kann innerhalb der Argumentliste benutzt werden und der ausgewertete Wert wird an die lokale Variable übergeben.

attribute Ein Wert, der mit einem Objekt assoziiert ist, wird von einem Namen mittels Punkt-Ausdruck referenziert. Zum Beispiel, hätte ein Objekt *o* ein Attribut *a*, so würde es als *o.a* referenziert.

BDFL Benevolent Dictator For Life (Wohlwollender Diktator auf Lebenszeit), auch bekannt als Guido van Rossum, der Schöpfer von Python.

bytecode Die interne Darstellung eines Python-Programmes im Interpreter, Python-Programme werden zu ihm kompiliert. Der Bytecode wird in `.pyc`- und `.pyo`-Dateien gespeichert, so dass das Ausführen derselben Datei beim zweiten Mal schneller ist (da die erneute Kompilierung zu Bytecode vermieden werden kann). Diese

"Zwischensprache" (*intermediate language*) ist dazu gedacht auf einer *virtual machine* (Virtuellen Maschine) ausgeführt zu werden, die den Maschinencode zum jeweiligen Bytecode ausführt.

class Eine Vorlage für die Erstellung benutzerdefinierter Objekte. Klassendefinitionen enthalten normalerweise Methodendefinitionen, die auf den Exemplaren der Klasse agieren.

coercion Die implizite Konvertierung eines Exemplares eines Typs in einen anderen während einer Operation, die zwei Argumente desselben Typs erfordert. Zum Beispiel konvertiert `int(3.15)` die Fliesskomma-Zahl zu der Ganzzahl `3`, aber in `3 + 4.5` ist jedes Argument von einem verschiedenen Typ (eines *int*, eines *float*) und beide müssen zum selben Typ konvertiert werden oder es wird ein `TypeError` erzeugt. Ohne Coercion müssten alle Argumente, selbst von kompatiblen Typen, zum selben Typ vom Programmierer normalisiert werden, z.B. `float(3) + 4.5` statt nur `3 + 4.5`.

complex number Eine Erweiterung zum bekannten reellen Zahlensystem, in dem alle Zahlen als eine Summe eines reellen Anteils und eines imaginären Anteils ausgedrückt werden. Imaginäre Zahlen sind echte Vielfache der imaginären Einheit (der Wurzel von `-1`), oft in der Mathematik oft als *i* oder im Ingenieurwesen als *j* geschrieben. Python hat eingebaute Unterstützung für Komplexe Zahlen, die mit folgender Notation geschrieben werden: Der imaginäre Anteil wird mit dem Suffix *j* geschrieben, z.B. `3+1j`. Um Zugang zu den komplexen Äquivalenten des `math`-Modules zu bekommen, benutzt man `cmath`. Der Gebrauch von Komplexen Zahlen ist ein recht fortgeschrittenes mathematisches Werkzeug. Kennt man keine Notwendigkeit sie zu benutzen, ist es fast sicher, dass man sie getrost ignorieren kann.

context manager Ein Objekt, das die Umgebung, der man in einer `with`-Anweisung begegnet, kontrolliert, indem es die Methoden `__enter__()` und `__exit__()` definiert. Siehe **PEP 343**.

CPython Die kanonische Implementierung der Pyton Programmiersprache. Der Term "CPython" wird in Kontexten benutzt, in denen es nötig ist diese Implementierung von anderen wie Jython oder IronPython zu unterscheiden.

decorator Eine Funktion, die eine andere Funktion zurückgibt, normalerweise als Funktionstransformation durch die `@wrapper`-Syntax benutzt. Häufige Beispiele für Dekoratoren sind `classmethod()` und `staticmethod()`.

Die Dekorator-Syntax ist nur Syntaktischer Zucker (syntactic sugar). Die beiden folgenden Definitionen sind semantisch äquivalent:

```
def f(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

Dasselbe Konzept existiert für Klassen, ist jedoch dort weniger gebräuchlich. Siehe die Dokumentation für Funktionsdefinitionen und Klassendefinitionen für mehr über Dekoratoren.

descriptor Jedes Objekt, das die Methoden `__get__()`, `__set__()` oder `__delete__()` definiert. Wenn ein Klassenattribut ein Deskriptor ist, wird sein spezielles Bindeverhalten beim Attributs-Lookup ausgelöst. Wenn man *a.b* für das Abfragen (*get*), Setzen (*set*) oder Löschen (*delete*) eines Attributs benutzt, wird nach einem Objekt namens *b* im Klassendictionary von *a* gesucht, ist *b* aber ein Deskriptor, wird die jeweilige Deskriptor-Methode aufgerufen. Das Verstehen von Deskriptoren ist wichtig für ein tiefes Verständnis von Python, da sie die Basis für viele Features einschliesslich Funtionen, Methoden, Properties, Klassenmethoden, statische Methoden und Referenzen zu Super-Klassen bilden.

Für mehr Informationen zu den Deskriptor-Methoden, siehe Implementing Descriptors.

dictionary Ein assoziatives Array, wo beliebige Schlüssel auf Werte abgebildet werden. Die Benutzung von `dict` kommt der von `list` sehr nahe, aber ein Schlüssel kann jedes Objekt sein, das eine `__hash__()`-Methode hat, nicht nur Ganzzahlen. Trägt den Namen *hash* in Perl.

docstring Ein Stringliteral, das als erster Ausdruck in einer Klasse, Funktion oder einem Modul vorkommt. Während es beim Ausführen der Suite ignoriert wird, erkennt der Compiler es und weist es dem `__doc__`-Attribut der umgebenden Klasse, Funktion oder Modul zu. Da es durch Introspektion verfügbar ist, ist es der kanonische Ort für Dokumentation des Objekts.

duck-typing Ein pythonischer Programmierstil, der den Typ eines Objektes anhand seiner Methoden- oder Attributssignatur bestimmt, statt durch die explizite Zuordnung zu einem Typ-Objekt. ("Sieht es wie eines Ente aus und quakt es wie eine Ente, dann muss es eine Ente sein.") Durch die Hervorhebung von Schnittstellen statt spezifischer Typen, verbessert ein gut-durchdachter Code seine Flexibilität, indem er polymorphe Substitution zulässt. Duck-typing vermeidet Tests mittels `type()` oder `isinstance()`. (Beachte jedoch, dass duck-typing durch Abstrakte Basis Klassen ergänzt werden kann.) Stattdessen benutzt es Tests mit `hasattr()` oder *EAFP*-Programmierung.

EAFP "Easier to ask for forgiveness than permission." (Leichter um Vergebung zu bitten, als um Erlaubnis.) Dieser geläufige Python-Programmierstil setzt die Existenz von validen Schlüsseln oder Attributen voraus und fängt Ausnahmen ab, wenn die Voraussetzung nicht erfüllt wurde. Für diesen sauberen und schnellen Stil ist die Präsenz vieler `try`- und `except`-Anweisungen charakteristisch. Diese Technik hebt sich von dem *LBYL*-Stil ab, der in vielen anderen Sprachen wie beispielsweise C geläufig ist.

expression Ein Stück Syntax, die zu einem Wert evaluiert werden kann. Mit anderen Worten ist ein Ausdruck eine Anhäufung von Ausdruckselementen wie Literale, Namen, Attributszugriffe, Operatoren oder Funktionsaufrufen, die alle einen Wert zurückgeben. Im Unterschied zu vielen anderen Sprachen, sind nicht alle Sprachkonstrukte Ausdrücke. Es gibt ebenfalls Anweisungen (*statement*), die nicht als als Ausdruck benutzt werden können, wie etwa `if`. Zuweisungen sind ebenfalls Anweisungen, keine Ausdrücke.

extension module Ein Modul, das in C oder C++ geschrieben ist und mit Pythons C API mit dem Kern und dem Benutzer-Code zusammenarbeitet.

finder Ein Objekt, das versucht den *loader* für ein Modul zu finden. Es muss eine Methode namens `find_module()` implementieren.

Siehe **PEP 302** für Details und `importlib.abc.Finder` für eine *abstract base class*.

floor division Mathematische Division die jeden Rest verwirft. Der Operator für Ganzzahl-Division ist `//`. Zum Beispiel evaluatiert der Ausdruck `11//4` zu `2` im Gegensatz zu `2.75`, die von Fliesskomma-Division zurückgegeben wird.

function Eine Serie von Anweisungen, die einen Wert zum Aufrufenden zurückgeben. Ihr können ebenfalls null oder mehr Argumente übergeben werden, die in der Ausführung des Rumpfs benutzt werden können.

Siehe auch *argument* und *method*.

__future__ Ein Pseudo-Modul, das Programmierern ermöglicht neue Sprach-Features zu aktivieren, die nicht kompatibel mit dem aktuellen Interpreter sind.

Durch den Import des `__future__`-Moduls und dem Auswerten seiner Variablen, kann man sehen, wann ein Feature zuerst der Sprache hinzugefügt wurde und wann es das Standard-Verhalten wird:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection Der Prozess des Freigebens nicht mehr benötigten Speichers. Pythons garbage collection erfolgt mittels des Zählens von Referenzen (*reference counting*) und einem zyklischen Garbage Collectors, der imstande ist Referenzzyklen zu entdecken und aufzubrechen.

generator Eine Funktion die einen Iterator zurückgibt. Sie sieht aus wie eine normale Funktion, mit der Ausnahme, dass Werte zum Aufrufenden mittels einer `yield`-Anweisung statt mit einer `return`-Anweisung zurückgegeben werden. Generator-Funktionen enthalten oft eine oder mehrere Schleifen (`for` oder `while`), die dem Aufrufenden Elemente liefern (`yield` en). Die Ausführung der Funktion wird nach dem `yield` unterbrochen

(während das Ergebnis zurückgegeben wird) und wird dort wiederaufgenommen, wenn das nächste Element durch den Aufruf der `__next__()`-Methode des zurückgegebenen Iterators angefordert wird.

generator expression Ein Ausdruck, der einen Generator zurückgibt. Er sieht wie ein normaler Ausdruck aus, gefolgt von einem `for`-Ausdruck, der eine Schleifenvariable - hier *range* - definiert und einem optionalem `if`-Ausdruck. Der kombinierte Ausdruck generiert Werte für eine umgebende Funktion:

```
>>> sum(i*i for i in range(10))         # summe der quadrate von 1,2, ..., 10
285
```

GIL Siehe *global interpreter lock*.

global interpreter lock multi-processor machines. Efforts have been made in the past to create a "free-threaded" interpreter (one which locks shared data at a much finer granularity), but so far none have been successful because performance suffered in the common single-processor case.

Das Lock, das von Python-Threads benutzt wird, um sicherzustellen, dass nur ein Thread gleichzeitig in der Virtuellen Maschine (*virtual machine*) von *CPython* ausgeführt wird. Den ganzen Interpreter zu locken, macht es dem Interpreter einfacher multi-threaded zu sein, auf Kosten eines Großteils der Parallelität, die von Multi-Prozessor Maschinen bereitgestellt wird. In der Vergangenheit gab es viele Bestrebungen einen "free-threaded" Interpreter (der den Zugriff auf geteilte Daten in einer feineren Granularität blockt) zu erschaffen, jedoch war noch keiner erfolgreich, da alle Performance-Einbußen im häufigen Fall des Einzel-Prozessors.

hashable Ein Objekt ist *hashbar*, wenn es einen Hashwert hat, der sich niemals während seiner Existenz ändert (es braucht eine `__hash__()`-Methode) und mit anderen Objekten verglichen werden kann (es braucht eine `__eq__()`-Methode). Hashbare Objekte, die sich gleichen, müssen denselben Hashwert haben.

Hashbarkeit macht ein Objekt als Dictionary-Schlüssel und als Mengen-Mitglied benutzbar, da diese Datenstrukturen intern den Hashwert benutzen.

Alle von Pythons eingebauten, unveränderbaren Objekte sind hashbar, während keiner der veränderbaren Container (wie Listen oder Dictionaries) es ist. Objekte, die Exemplare von benutzerdefinierten Klassen sind, sind standardmäßig hashbar; sie vergleichen auf ungleich und ihr Hashwert ist ihre `id()`.

IDLE Eine IDE (Integrated Development Environment) für Python. IDLE ist eine einfache Editor- und Interpreter-Umgebung, die in der Standard-Distribution von Python enthalten ist. Gut für Anfänger geeignet und dient auch als Beispiel-Code für alle, die eine moderat komplexe, Multi-Plattform GUI Anwendung erstellen wollen.

immutable Ein Objekt mit einem festen Wert. Zu den unveränderbaren (*immutable*) Objekten zählen Zahlen, Strings und Tupel. Solche Objekte könnnen nicht verändert werden. Ein neues Objekte muss erzeugt werden, wenn ein verschiedener Wert gespeichert werden muss. Sie spielen eine wichtige Rolle an Stellen, bei denen ein konstanter Hashwert benötigt wird, zum Beispiel als Schlüssel in einem Dictionary.

importer Ein Objekt, das sowohl Module fundet und lädt; zugleich ein *finder*- und *loader*-Objekt.

interactive Python hat einen interaktiven Interpreter. Das bedeutet, dass man Anweisungen und Ausdrücke in den Interpreter-Prompt eingeben kann, die sofort ausgeführt werden und deren Ergebnis man sehen kann. Man startet einfach `python` ohne Argumente (möglicherweise indem man es im Hauptmenü des Computers auswählt). Es ist ein mächtiger Weg, um neue Ideen zu testen oder Module und Pakete zu untersuchen (`help(x)` ist hilfreich).

interpreted Python ist eine interpretierte Sprache, im Gegensatz zu einer kompilierten, obwohl die Unterscheidung aufgrund des Bytecode-Compilers verschwommen ist. Das heisst, dass Quelldateien direkt ausgeführt werden können ohne explizit eine ausführbare Datei zu erstellen, die dann ausgeführt wird. Interpretierte Sprachen haben typischerweise einen kürzeren Entwicklungs/Debug-Zyklus als kompilierte, jedoch laufen deren Programme generell etwas langsamer. Siehe auch *interactive*.

iterable Ein Container-Objekt, das dazu imstande ist seine Mitglieder nacheinander zurückzugeben. Beispiele von Iterables sind alle Sequenztypen (wie etwa `list`, `str` und `tuple`) und einige nicht-Sequenztypen wie `dict` und `file` und Objekte, die man mit `__iter__()`- oder `__getitem__()`-Methoden definiert. Iterables können in `for`-Schleifen und vielen anderen Stellen verwendet werden, wo eine Sequenz benötigt wird

(`zip()`, `map()`, etc.). Wird ein Iterable als Argument der eingebauten Funktion `iter()` übergeben, gibt sie einen Iterator für dieses Objekt zurück. Dieser Iterator ist gut, für einen Durchlauf über die Menge der Werte. Nutzt man Iterables, ist es meist nicht nötig `iter()` aufzurufen oder sich mit Iterator-Objekten direkt zu befassen. Die `for`-Anweisung erledigt das automatisch, indem sie eine temporäre unbenannte Variable erstellt, um den Iterator für die Laufzeit der Schleife zu halten. Siehe auch *iterator*, *sequence* und *generator*.

iterator Ein Objekt, das einen Datenstrom repräsentiert. Wiederholte Aufrufe der `__next__()`-Methode des Iterators oder die Übergabe an die eingebaute Funktion `next()` geben die aufeinanderfolgenden Elemente im Datenstrom zurück. Sind keine Daten mehr vorhanden, wird eine `StopIteration`-Ausnahme ausgelöst. An dieser Stelle ist das Iterator-Objekt erschöpft und alle weiteren Aufrufe verursachen nur weitere `StopIteration`. Iteratoren müssen ebenfallse eine `__iter__()`-Methode haben, die den Iterator selbst zurückgibt, sodass jeder Iterator selbst ein Iterable ist und in den meisten Fällen benutzt werden kann, wo andere Iterables akzeptiert werden. Eine wichtige Ausnahme ist Code, der mehrere Iterationen versucht. Ein Container-Objekt (wie etwa `list`) erzeugt jedes Mal einen neuen Iterator, wenn man es der `iter()`-Funktion übergibt oder in einer `for`-Schleife benutzt. Versucht man dies mit einem Iterator, wird nur dasselbe erschöpfte Iterator-Objekt zurückgeben, das schon im vorangegangenen Durchlauf benutzt wurde und es so wie einen leeren Container erscheinen lässt.

Mehr Informationen können bei Iterator Types gefunden werden.

keyword argument Argumente, denen ein `variable_name=` im Aufruf vorausgeht. Der Variablenname bestimmt den lokalen Namen der Funktion, dem der Wert zugewiesen wird. `**` wird benutzt um ein Dictionary von Schlüsselwort-Argumenten zu übergeben oder zu akzeptieren. Siehe *argument*.

lambda Eine anonyme inline Funktion, die nur aus einem einzelnen Ausdruck (*expression*) besteht, der ausgewertet wird, wenn die Funktion aufgerufen wird. Die Syntax, um eine lambda-Funktion zu erstellen ist `lambda [arguments]: expression`.

LBYL "Look before you leap." ("Schau bevor du springst.") Dieser Programmierstil testet explizit auf Vorbedingungen bevor Aufrufe oder Lookups getätigt werden. Dieser Stil steht dem *EAFP* Ansatz gegenüber und die Präsenz vieler `if`-Anweisungen ist charakteristisch für ihn.

list Eine eingebaute Python *sequence*. Trotz des Namens ist sie ähnlicher zu Arrays in anderen Sprachen als zu Verknüpften Listen (*linked lists*), da der Elementzugriff in O(1) ist.

list comprehension Ein kompakter Weg, um alle oder Teile der Elemente in einer Sequenz verarbeitet und eine Liste der Ergebnisse zurückgibt. `result = ["0x%02x"% x for x in range(256) if x % 2 == 0]` generiert eine Liste von Strings, die die geraden Hex-Zahlen (0x..) im Bereich von 0 bis 255 enthält. Der `if`-Abschnitt ist optional. Wird er ausgelassen, werden alle Elemente von `range(256)` verarbeitet.

loader Ein Objekt, das ein Modul lädt. Es muss eine Methode namens `load_module()` definieren. Ein *loader* wird typischerweise von einem *finder* zurückgegebenen. Siehe **PEP 302** für Details und `importlib.abc.Loader` für eine *abstract base class*.

mapping Ein Container-Objekt (wie etwa `dict`), das beliebige Schlüssel-Lookups mittels der speziellen Methode `__getitem()` unterstützt.

metaclass Die Klasse einer Klasse. Klassendefinitionen erstellen einen Klassennamen, ein Klassendictionary und eine Liste der Basisklassen. Eine Metaklasse ist dafür verantwortlich diese drei Argumente entgegen zunehmen und Klassen zu erzeugen. Die meisten Objektorientierten Programmiersprachen bieten eine Standard-Implementierung. Was Python speziell macht, ist dass es möglich ist eigene Metaklassen zu erstellen. Die meisten Benutzer benötigen dieses Werkzeug nicht, kommt das Bedürfnis aber auf, können Metaklassen mächtige und elegante Lösungen bieten. Sie wurden schon benutzt um Attributszugriffe zu loggen, Thread-Sicherheit hinzuzufügen, Objekterzeugung zu verfolgen, Singletons zu implementieren und für viele andere Aufgaben.

Mehr Informationen können in Customizing class creation gefunden werden.

method Eine Funktion, die innerhalb eines Klassenkörpers definiert wurde. Wird es als Attribut eines Exemplares dieser Klasse aufgerufen, bekommt die Methode das Exemplar-Objekt als ihr erstes Argument (*argument*) (das normalerweise `self` genannt wird). Siehe *function* und *nested scope*.

mutable Veränderliche (*mutable*) Objekte können ihren Wert ändern, aber ihre `id()` behalten. Siehe auch *immutable*

named tuple Jede Tupel-ähnliche Klasse, deren indizierbaren Elemente auch über benannte Attribute zugänglich sind (zum Beispiel gibt `time.localtime()` ein Tupel-ähnliches Objekt zurück, wo das Jahr sowohl durch einen Index, wie `t[0]`, als auch durch ein benanntes Attribut wie `t.tm_year` zugänglich ist).

Ein benanntes Tupel kann ein eingebauter Typ wie etwa `time.struct_time` sein oder es kann mit einer regulären Klassendefinition erstellt werden. Ein voll funktionierendes benanntes Tupel kann auch mit der Factory-Funktion `collections.namedtuple()` erstellt werden. Der zweite Ansatz bietet automatische extra Features wie eine selbst-dokumentierende Repräsentation wie `Employee(name='jones', title='programmer')`.

namespace Der Ort, an dem eine Variable gespeichert wird. Namensräume sind als Dictionaries implementiert. Es gibt lokale, globale und eingebaute Namensräume, wie auch verschachtelte Namensräume in Objekten (in Methoden). Namensräume unterstützen Modularität, indem sie Namenskonflikten vorbeugen. Zum Beispiel werden die Funktionen `builtins.open()` und `os.open()` anhand ihres Namensraumes unterschieden. Namensräume unterstützen außerdem die Lesbarkeit und Wartbarkeit indem sie klar machen, welches Modul eine Funktion implementiert. Zum Beispiel machen `random.seed()` oder `itertools.izip()` es klar, dass diese Funktionen in den Modulen `random` beziehungsweise `itertools` implementiert werden.

nested scope Die Fähigkeit eine Variable in einer umgebenden Definition zu referenzieren. Zum Beispiel, kann eine Funktion, die in einer anderen Funktion definiert wird auf die Variablen in der äußeren Funktion zugreifen. Beachte, dass verschachtelte Gültigkeitsbereiche nur bei Referenzierungen, nicht bei Zuweisungen, die immer in den innersten Gültigkeitsbereich schreiben, funktionieren. Im Gegensatz dazu lesen und schreiben lokale Variablen in den innersten Gültigkeitsbereich. Gleichfalls, lesen und schreiben globale Variablen in den globalen Namensraum.

new-style class Alter Name für die Sorte von Klassen, die nun für alle Klassenobjekte benutzt wird. In früheren Versionen von Python, konnten nur new-style Klassen Pythons neuere vielseitige Features wie `__slots__`, Deskriptoren, Properties und `__getattribute__()` benutzen.

object Alle Daten mit Zustand (Attribute oder Wert) und definiertem Verhalten (Methoden). Ebenfalls die ultimative Basisklasse von jeder *new-style class*.

positional argument Die Argumente, die lokalen Namen innerhalb einer Funktion oder Methode zugewiesen werden, die von der Reihenfolge in der sie im Aufruf angegeben werden festgelegt sind. `*` wird benutzt um entweder mehrere Positionsargumente entgegenzunehmen (wenn es in der Definition vorkommt) oder um mehrere Argumente als eine Liste einer Funktion zu übergeben. Siehe *argument*.

Python 3000 Spitzname für die Reihe der Veröffentlichungen in Python 3.x (geprägt vor langer Zeit, als die Veröffentlichung von Version 3 etwas in ferner Zukunft war.) Dies wird auch als "Py3k" abgekürzt.

Pythonic Eine Idee oder Stück von Code, der den häufigsten Idiomen der Python-Sprache eng folgt, statt Konzepte zu verwenden, die häufig in anderen Sprachen vorkommen. Zum Beispiel ist es ein häufiges Idiom in Python über alle Elemente eines Iterable mithilfe einer `for`-Anweisung zu iterieren. Viele andere Sprachen haben nicht diese Art von Konstrukt, sodass Leute, die mit Python nicht vertraut sind manchmal einen numerischen Zähler benutzen:

```
for i in range(len(food)):
    print(food[i])
```

Im Gegensatz zum sauberen, pythonischen Weg:

```
for piece in food:
    print(piece)
```

reference count Die Anzahl von Referenzen zu einem Objekt. Fällt der Referenzzähler eines Objekts auf null, wird es deallociert. Das Referenzzählen ist generell nicht sichtbar für Python-Code, ist jedoch ein Schlüsselelement

der *CPython* Implementierung. Das Modul `sys` definiert eine `getrefcount()`-Funktion, die Programmierer aufrufen können, um den Referenzzähler für ein bestimmtes Objekt zu bekommen.

__slots__ Eine Deklaration innerhalb einer Klasse, die Speicher spart, indem der Platz für Instanzattribute vorher deklariert wird und Exemplardictionaries eliminiert werden. Auch wenn sie populär sind, ist es trickreich die Technik richtig anzuwenden und sollte am besten für seltene Fälle aufgehoben werden, wenn es große Zahlen von Exemplaren in einer speicherkritischen Anwendung gibt.

sequence Ein Iterable (*iterable*), das effizienten Elementzugriff mit Ganzzahlindizes durch die spezielle Methode `__getitem__()` bietet und eine `__len__()`-Methode definiert, die die Länge der Sequenz zurückgibt. Manche eingebauten Sequenztypen sind `list`, `str`, `tuple` und `bytes`. Beachte, dass `dict` ebenfalls `__getitem__()` und `__len__()` definiert, aber eher als Mapping (*mapping*), denn als Sequenz angesehen, da die Lookups durch beliebige unveränderbare (*immutable*) Schlüssel möglich sind, nicht nur durch Ganzzahlen.

slice Ein Objekt, das normalerweise einen Abschnitt einer Sequenz (*sequence*) enthält. Ein Slice wird mittels der Subskript-Notation, `[]` mit Doppelpunkten zwischen Nummern, wenn mehrere gegeben werden, wie in `variable_name[1:3:5]`. Die Notation mit eckigen Klammern (Subskript-Notation) benutzt `slice`-Objekte intern.

special method Eine Methode die implizit von Python aufgerufen wird, um eine bestimmte Operation auf einem Typ auszuführen, wie etwa Addition. Solche Methoden haben Namen mit führenden wie abschliessenden doppelten Unterstrichen. Spezielle Methoden sind bei Special method names dokumentiert.

statement Eine Anweisung ist Teil einer Suite (ein "Block" von Code). Eine Anweisung ist entweder ein Ausdruck (*expression*) oder eine von mehreren Konstrukten mit einem Schlüsselwort, wie etwa `if`, `while` oder `for`.

triple-quoted string Ein String, der von entweder drei Anführungszeichen (") oder Apostrophen (') umgeben ist. Während sie keine Funktionalität bieten, die nicht bei einfach-quotierten Strings verfügbar wären, sind sie aus mehreren Gründen nützlich. Sie erlauben das Einbeziehen von unmaskierten Anführungszeichen und Apostrophen innerhalb eines Strings und sie können mehrere Zeilen umfassen ohne das Fortsetzungszeichen benutzen zu müssen, was sie besonders nützlich beim Schreiben von Docstrings macht.

type Der Typ eines Python-Objektes legt fest, welche Art von Objekt es ist; jedes Objekt hat einen Typ. Der Typ eines Objektes ist als dessen `__class__`-Attribut zugänglich oder kann mit `type(obj)` bestimmt werden.

view Die Objekte, die von `dict.keys()`, `dict.values()` und `dict.items()` zurückgegeben werden, werden Dictionary-Views genannt. Sie sind Lazy Sequenzen, die Veränderungen im zugrundeliegenden Dictionary bemerken. Um einen Dictionary-View zu zwingen eine volle Liste zu werden, benutzt man `list(dictview)`. Siehe Dictionary view objects.

virtual machine Ein Computer, der komplett in Software definiert ist. Pythons Virtuelle Maschine führt den *bytecode* aus, den der Bytecode-Compiler erzeugt.

Zen of Python Aufzählung von Pythons Design Prizipien und Philosophien, die hilfreich beim verstehen und benutzen der Sprache sind. Gibt man "`import this`" am interaktiven Prompt ein, kann man die Aufzählung einsehen.

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

Z