

Programmieren in Python

Eine praktische Einführung



Das tuxcademy-Projekt bietet hochwertige frei verfügbare Schulungsunterlagen zu Linux- und Open-Source-Themen – zum Selbststudium, für Schule, Hochschule, Weiterbildung und Beruf.
Besuchen Sie <https://www.tuxcademy.org/>! Für Fragen und Anregungen stehen wir Ihnen gerne zur Verfügung.

Programmieren in Python Eine praktische Einführung

Revision: pyth:e431aefb71298e66:2016-07-18

pyth:4294f89813971178:2016-07-18 1-7

pyth:EfL030ydsekLVrB525xdF7

© 2015 Linup Front GmbH

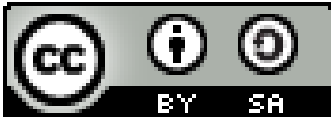
© 2016 tuxcademy (Anselm Lingnau) Mainz

<https://www.tuxcademy.org> · info@tuxcademy.org

Linux-Pinguin »Tux« © Larry Ewing (CC-BY-Lizenz)

Alle in dieser Dokumentation enthaltenen Darstellungen und Informationen wurden nach bestem Wissen erstellt und mit Sorgfalt getestet. Trotzdem sind Fehler nicht völlig auszuschließen. Das tuxcademy-Projekt haftet nach den gesetzlichen Bestimmungen bei Schadensersatzansprüchen, die auf Vorsatz oder grober Fahrlässigkeit beruhen, und, außer bei Vorsatz, nur begrenzt auf den vorhersehbaren, typischerweise eintretenden Schaden. Die Haftung wegen schuldhafter Verletzung des Lebens, des Körpers oder der Gesundheit sowie die zwingende Haftung nach dem Produkthaftungsgesetz bleiben unberührt. Eine Haftung über das Vorgenannte hinaus ist ausgeschlossen.

Die Wiedergabe von Warenbezeichnungen, Gebrauchsnamen, Handelsnamen und ähnlichem in dieser Dokumentation berechtigt auch ohne deren besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne des Warenzeichen- und Markenschutzrechts frei seien und daher beliebig verwendet werden dürften. Alle Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt und sind möglicherweise eingetragene Warenzeichen Dritter.



Diese Dokumentation steht unter der »Creative Commons-BY-SA 4.0 International«-Lizenz. Sie dürfen sie vervielfältigen, verbreiten, und öffentlich zugänglich machen, solange die folgenden Bedingungen erfüllt sind:

Namensnennung Sie müssen darauf hinweisen, dass es sich bei dieser Dokumentation um ein Produkt des tuxcademy-Projekts handelt.

Weitergabe unter gleichen Bedingungen Sie dürfen die Dokumentation bearbeiten, abwandeln, erweitern, übersetzen oder in sonstiger Weise verändern oder darauf aufbauen, solange Sie Ihre Beiträge unter derselben Lizenz zur Verfügung stellen wie das Original.

Mehr Informationen und den rechtsverbindlichen Lizenzvertrag finden Sie unter <http://creativecommons.org/licenses/by-sa/4.0/>

Autor: Anselm Lingnau

Technische Redaktion: Anselm Lingnau (anselm.lingnau@linupfront.de)

Gesetzt in Palatino, Optima und DejaVu Sans Mono



Inhalt

1 Einführung	11
1.1 Python-Hintergrund	12
1.2 Python als Programmiersprache	13
1.3 Ein einfaches Python-Programm	15
1.4 Python interaktiv	18
2 Einfache Datentypen	21
2.1 Numerische Datentypen und Literale	22
2.2 Numerische Ausdrücke	24
2.3 Variable	27
2.4 Zeichenketten	29
2.4.1 Zeichenketten-Literale	29
2.4.2 Zeichenketten-Ausdrücke	31
3 Verzweigungen und Schleifen	33
3.1 Verzweigungen mit if	34
3.1.1 Einstieg	34
3.1.2 Boolesche Ausdrücke	34
3.1.3 else und elif	37
3.2 Schleifen mit while	39
3.2.1 Wiederholungen	39
3.2.2 break und continue	41
4 Strukturierte Datentypen: Folgen	45
4.1 Strukturierte Daten: Tupel	46
4.2 Strukturierte Daten: Listen	51
4.3 Mehr über Zeichenketten	56
4.4 Schleifen mit for und Ranges	62
4.5 List Comprehensions	66
5 Funktionen	69
5.1 Einfache Funktionen	70
5.2 Parameterübergabe	71
5.3 Variable Signaturen	75
5.4 Sichtbarkeitsbereiche	76
6 Strukturierte Datentypen: Dictionaries und Mengen	81
6.1 Dictionaries	82
6.2 Funktionsaufrufe mit beliebigen benannten Parametern	89
6.3 Dictionary Comprehensions	90
6.4 Mengen	92
7 Objektorientierte Programmierung	99
7.1 Klassen, Attribute und Methoden	100
7.2 Konstruktoren und Destruktoren	105
7.3 Vererbung	107

7.4 Operatoren überladen	111
7.5 »Duck Typing«.	115
A Musterlösungen	119
Index	123



Tabellenverzeichnis

2.1	Rückstrich-Kombinationen in Zeichenketten-Literalen	30
3.1	Vergleichsoperatoren in Python	35
7.1	Mathematische Operatoren und die dazugehörigen Methodennamen	113



Abbildungsverzeichnis

1.1	Ein simples Python-Programm	15
-----	---------------------------------------	----

tionen wesentlicher Begriffe sind im Text fett gedruckt und erscheinen ebenfalls am Rand.

Verweise auf Literatur und interessante Web-Seiten erscheinen im Text in der Form »[GPL91]« und werden am Ende jedes Kapitels ausführlich angegeben.

Wir sind bemüht, diese Schulungsunterlage möglichst aktuell, vollständig und fehlerfrei zu gestalten. Trotzdem kann es passieren, dass sich Probleme oder Ungenauigkeiten einschleichen. Wenn Sie etwas bemerken, was Sie für verbesserungsfähig halten, dann lassen Sie es uns wissen, etwa indem Sie eine elektronische Nachricht an

`info@tuxcademy.org`

schicken. (Zur Vereinfachung geben Sie am besten den Titel der Schulungsunterlage, die auf der Rückseite des Titelblatts enthaltene Revisionsnummer sowie die betreffende(n) Seitenzahl(en) an.) Vielen Dank!



1

Einführung

Inhalt

1.1	Python-Hintergrund.	12
1.2	Python als Programmiersprache	13
1.3	Ein einfaches Python-Programm	15
1.4	Python interaktiv	18

Lernziele


- Grundlagen der Programmiersprache Python kennen
- Stärken und Schwächen von Python einschätzen können
- Einfache Python-Programme eintippen und starten können
- Python als interaktiven Interpreter aufrufen können

Vorkenntnisse


- Elementare Linux-Kenntnisse (als Anwender)
- Umgang mit einem (beliebigen) Texteditor
- Erfahrung mit anderen Programmiersprachen ist von Vorteil


1.1 Python-Hintergrund


Die Programmiersprache Python wurde in den späten 1980er Jahren von Guido van Rossum erfunden. Van Rossum war damals beim Zentrum für Mathematik und Informatik der Freien Universität Amsterdam beschäftigt (heute (2015) arbeitet er für die Firma Dropbox). Python wird heute als Open-Source-Softwareprojekt von einem Team von Freiwilligen unter der Ägide der »Python Software Foundation« (einer gemeinnützigen Stiftung) betreut und weiterentwickelt. Guido van Rossum hat immer noch das letzte Wort; sein Titel im Projekt ist »BDFL« oder *benevolent dictator for life* (»wohlwollender Diktator auf Lebenszeit«).

 Python 1.0 wurde im Januar 1994 veröffentlicht, Python 2.0 im Oktober 2000 (die wesentlichen Neuerungen waren *garbage collection* und die Unterstützung von Unicode). Python 3.0 kam im Dezember 2008 heraus. Aktuell (2015) sind die Versionen 2.7.x bzw. 3.4.x.

Python beruht in seinen Grundideen auf einer früheren Programmiersprache namens ABC, die besonders für Unterrichtszwecke gedacht war und an deren Entwicklung Guido van Rossum beteiligt war. Der wesentliche Unterschied zwischen Python und ABC ist, dass Python auch in der wirklichen Welt zu gebrauchen ist – ABC hatte diverse eigentümliche Eigenschaften, die das nicht unbedingt förderten, und die Sprache ist inzwischen auf der Schutthalde der Informatik-Geschichte gelandet.

 Python 3.x hat gegenüber Python 2.x diverse nicht rückwärtskompatible Änderungen – man nahm den Versionsprung zur Gelegenheit, einige »dunkle Ecken« der Programmiersprache aufzuräumen und alte Zöpfe abzuschneiden. Das bedeutet, dass nicht jedes Python-2-Programm unverändert mit Python 3 funktioniert. Es gibt aber Mittel und Wege, Code zu schreiben, der mit beiden Versionen zurechtkommt, und außerdem gibt es automatische Anpassungswerkzeuge, die einen Großteil der (schematischen) Änderungen übernehmen, die für eine Anpassung eines Python-2-Programms an Python 3 nötig sind.

 Python 2 soll noch bis 2020 weiter gewartet werden – allerdings werden grundsätzlich nur noch Fehlerkorrekturen vorgenommen. Die Python-Weiterentwicklung konzentriert sich schon seit einiger Zeit fast völlig auf Python 3.

 Diese Schulungsunterlage beschäftigt sich vorwiegend mit Python 3. Wo sinnvoll und nötig weisen wir auf Eigenheiten von Python 2 hin.

Gemäß Guido van Rossum (1999) sind die Entwicklungsziele von Python:

- Die Sprache soll einfach und intuitiv sein, aber mächtig.
- Eine Open-Source-Implementierung der Sprache soll zur Verfügung stehen, damit jeder zur Weiterentwicklung beitragen kann.
- Der Code soll so leicht verständlich sein wie Englisch.
- Die Sprache soll für Alltagsaufgaben nützlich sein und kurze Entwicklungszeiten fördern.

Inzwischen gehört Python zu den populärsten Programmiersprachen. Auf dem Internet finden Sie unter <http://www.python.org/> die offizielle Web-Seite der »Python Software Foundation«.

1.2 Python als Programmiersprache

Python ist eine moderne Programmiersprache mit vielen nützlichen Eigenschaften. Zu diesen gehören zum Beispiel:

Dynamische Typisierung Variable müssen in Python nicht vor der Verwendung deklariert werden. Trotzdem wird zur Laufzeit überprüft, dass nur zulässige Operationen auf die jeweiligen Daten angewendet werden. Python unterstützt sehr flexible Parameterübergabe an Funktionen sowie Polymorphie, also die Möglichkeit, dieselbe Funktion mit unterschiedlichen Parametersignaturen aufzurufen.

Objektorientierung Python unterstützt objektorientierte Programmierung, also die Definition von Klassen und Objekten mit Methoden und Attributen. Vererbung, auch Mehrfachvererbung, ist möglich, genau wie Metaprogrammierung. Python's objektorientierte Möglichkeiten sind sehr mächtig, aber ihre Verwendung ist nicht zwingend: Kleine Programme und solche, für die ein objektorientiertes Vorgehen nicht sinnvoll ist, können ohne weiteres auch in einem klassischen »imperativen« Stil geschrieben werden.

Sehr leistungsfähige Datentypen Schon »ab Werk« enthält Python diverse interessante und leistungsfähige eingebaute Datentypen – neben numerischen Ganzzahlen und Gleitkommazahlen auch Ganzzahlen (potentiell) beliebiger Länge, komplexe Zahlen oder Dezimalzahlen, mit denen kaufmännische Rechnungen ohne Rundungsfehler möglich sind. Zeichenketten unterstützen Unicode. Es gibt eine Reihe von praktischen »Container-Typen«, darunter Tupel, Listen, Dictionaries und Mengen.

Automatische Speicherverwaltung Python kümmert sich transparent um die Verwaltung von Speicher für Objekte und räumt auch selbsttätig auf – ein Eingreifen des Programmierers ist nur sehr selten erforderlich. Dies erleichtert die Erstellung von Programmen für Anwendungen, wo Sicherheitslücken und Programmabstürze durch Fehler in der Speicherverwaltung nicht erwünscht sind (etwa Internet-Dienste).

Module Python-Programme können in Module aufgeteilt werden, die zusammengehörende Funktionalität einkapseln und über separate Namensräume verfügen. Auf diese Weise ist es bequem möglich, auch große Softwaresysteme in Python zu realisieren. Code-Wiederverwendung wird gefördert. Python enthält Infrastruktur für das automatische Testen von Modulen, und mit dem Sphinx-System ist eine komfortable Dokumentation möglich.

Systemnahe Programmierung Python erlaubt den direkten Zugriff auf Funktionen des unterliegenden Betriebssystems, aber enthält auch komfortable Abstraktionsschichten, um eine portable, betriebssystemübergreifende Programmierung zu ermöglichen. Python-Quellcode sowie -Bytecode ist zwischen verschiedenen Rechnerarchitekturen und Betriebssystemen portabel.

Standardbibliothek und Erweiterungen Python wird mit einer sehr reichhaltigen Standardbibliothek ausgeliefert, die diverse Anwendungsbereiche abdeckt – bis hin zum Netzwerkzugriff, der Unterstützung diverser Internet-Protokolle und -Datenformate und komfortablen Hilfsmitteln zur Programmerstellung wie Testgerüste, Protokollierung, Profilierung und ähnliches. Zahllose Erweiterungen für alle möglichen Zwecke sind im Internet als Open-Source-Code verfügbar.

Erweiterbarkeit Python-Module können in Python oder auch in C/C++ geschrieben werden – letzteres ist aus Effizienzgründen manchmal sinnvoll. Für den Python-Programmierer ist es syntaktisch nicht unterscheidbar,

ob eine Erweiterung in Python oder C realisiert wurde. Es ist auch möglich, Python als Erweiterungssprache für existierende Anwendungen zu verwenden.

Portabilität Die Standardimplementierung von Python ist in portablen ISO-C geschrieben und läuft auf jeder wesentlichen Betriebssystemplattform – außer auf Linux und Unix zum Beispiel auf Windows, OS X, QNX, Vx-Works, Android, ... Auch der Python-Bytecode ist portabel. Neben der C-Implementierung gib es Python-Implementierungen in diversen anderen Programmiersprachen, etwa C#, Java, JavaScript und Python (!).

Freie Verfügbarkeit Die Standardimplementierung von Python ist »frei verfügbar« und kann beliebig (auch kommerziell) eingesetzt werden. Es ist erlaubt, den Code zu ändern und die Änderungen kommerziell zu nutzen und zu vertreiben sowie Python in eigene Produkte zu integrieren und diese ggf. kommerziell zu vertreiben. Die Programmiersprache Python hängt nicht von einem einzelnen Anbieter ab und wird darum vermutlich existieren, solange sich noch jemand dafür interessiert.

Natürlich hat Python – wie jede Programmiersprache – auch ein paar Schwächen, die wir hier nicht verschweigen wollen:

Performance Python wird normalerweise als Interpreter implementiert und ist eine objektorientierte Programmiersprache mit diversen komfortablen Automatismen. Dafür bezahlen Sie als Anwender mit einem gewissen Performance-Verlust gegenüber sorgfältig handoptimiertem C-Code. Ob das tatsächlich ein Problem darstellt, ist natürlich anwendungsabhängig – als Programmierer können Sie in Python wesentlich produktiver sein als in einer Programmiersprache wie C, und das wiegt einen gewissen Effizienzverlust vielleicht auf. Außerdem werden Computer (und damit Ihre Python-Programme) mit der Zeit tendenziell schneller, und es ist einfacher, in einem großen Python-Programm genau die Teile, wo viel Zeit verbraten wird, in eine in C geschriebene und optimierte Erweiterung auszulagern, als ein komplettes großes Programm von Grund auf in C zu schreiben, nur damit ein paar Teile davon schnell ausgeführt werden können.

Dynamische Typisierung Pythons dynamische Typisierung ist eine sehr bequeme Sache, weil Sie nicht viel Zeit damit verbringen müssen, Variable mit den korrekten Datentypen vorzudeklarieren. Allerdings begibt Python sich dadurch der Möglichkeit, allfällige Fehler, die mit Datentypen und Operationen zu tun haben, schon zu erkennen, bevor das Programm ausgeführt wird. Statt dessen riskieren Sie einen Programmabsturz im falschen Moment. Sie können das bis zu einem gewissen Grad mit Disziplin und geschickter Programmierung ausgleichen, aber fairerweise sollten wir sagen, dass für »Hochsicherheitsanwendungen« Python in seiner aktuellen Form möglicherweise nicht die beste Programmiersprache ist.



Eines der interessanteren Themen der aktuellen (2015) Python-Entwicklung ist die Idee, optionale Datentypfestlegungen für Variable und vor allem Funktionsparameter zuzulassen. Damit könnte das Problem der statischen Typüberprüfung (bzw. deren bisheriger Abwesenheit in Python) bis zu einem gewissen Grad adressiert werden. Wir dürfen gespannt sein.

Die kanonische Python-Implementierung beruht auf einem Interpreter. Das heißt, anders als bei vielen anderen Programmiersprachen ist kein expliziter Übersetzungsvorgang nötig, bevor Sie ein Python-Programm starten können – Sie müssen nur den Python-Interpreter mit der Textdatei aufrufen, die das Programm enthält.

```
#!/usr/bin/python3
# hallo.py -- Ein sehr einfaches Python-Programm

print("Hallo Welt!")
```

Bild 1.1: Ein simples Python-Programm



Intern übersetzt Python Programme in einen maschinenunabhängigen Bytecode, der dann ausgeführt wird. Dieser Übersetzungsvorgang ist für den Benutzer transparent. Zur Beschleunigung kann der Python-Interpreter den Bytecode in einer Datei ablegen, damit das betreffende Programm nicht wieder übersetzt werden muss, solange es nicht geändert wurde. Der Python-Interpreter kümmert sich selbst um allfällige Neu-Übersetzungen.

Bytecode

Sie können den Python-Interpreter auch interaktiv aufrufen, um direkt Python-Code einzutippen und auszuführen. Siehe hierzu Abschnitt 1.4.

1.3 Ein einfaches Python-Programm

Python-Programme können Sie mit jedem beliebigen Texteditor anlegen. Voraussetzung ist nur, dass der Editor tatsächliche Textdateien erzeugt und keine Textverarbeitungs-Dokumente (wenn Sie LibreOffice benutzen wollen, müssen Sie also darauf achten, Ihr Programm im richtigen Format abzuspeichern).



Manche Texteditoren sind für die Programmierung in Python besser geeignet als andere. Das Wichtigste, worauf Sie bei der Auswahl eines Editors achten sollten, ist, dass der Editor die Grundzüge der Programmiersprache Python beherrscht und dadurch zum Beispiel in der Lage ist, Programmtext gemäß der Syntax einzufärben und etwa Schlüsselwörter oder Zeichenketten hervorzuheben. Damit werden Fehlerquellen wie zum Beispiel nicht ordentlich beendete Zeichenketten offensichtlicher. Ebenfalls nützlich ist automatische Einrückung, bei der der Editor »weiß«, dass die Zeilen nach einem `if` oder `for` eingerückt werden sollen, und dafür selbständig Vorschläge macht. Spezialisierte Programmierumgebungen haben noch wesentlich weitreichendere Hilfsmittel, die bis zur Refaktorisierung (dem Umschreiben von Programmen zur Vereinfachung) oder dem automatischen Aufrufen von Dokumentation beim Klick auf einen Funktionsnamen gehen.

Nach Konvention schreibt man Python-Code in Textdateien mit der Endung `».py«`.



Das ist nützlich, weil ein hinreichend schlauer Texteditor dann weiß, dass er es mit Python-Code zu tun hat. Spätestens wenn Sie Ihr Programm anderen Leuten zur Verfügung stellen, sollten Sie die Endung aber entfernen, da die verwendete Programmiersprache unter dem Strich ein Implementierungsdetail darstellt, das nicht wichtig ist. Am Ende schreiben Sie Ihr Programm gar doch nochmal neu in C, und dann wäre es höchst doof, wenn die Benutzergemeinde sich daran gewöhnt hätte, es über den Namen `prog.py` aufzurufen.

Ein simples Beispiel für ein Python-Programm sehen Sie in Bild 1.1. Der Code spricht weitgehend für sich, aber wir nehmen diese Gelegenheit, um ein paar grundlegende Dinge zu erwähnen:

- Leerzeilen haben für Python keine Bedeutung. Halten Sie sich also nicht zurück.

- Zeilen mit einem # am Anfang gelten als Kommentarzeilen. Der Kommentar erstreckt sich vom # bis zum Zeilenende und wird von Python ignoriert. Es ist also auch möglich, Zeilen der Form

```
print("Hallo Welt!")    # Freundliche Begrüßung
```

zu haben.



Programmiermethodischer Einschub: Verkneifen Sie sich (in egal welcher Programmiersprache) Zeilen wie

```
i = i + 1    # Erhöhe i um 1
```

So etwas sollten Sie nur schreiben, wenn Sie entweder im ersten Semester Informatik studieren und Ihren Tutor beeindrucken wollen, oder aber als Programmierer pro Byte Quelltext bezahlt werden. Denn es hat keinen praktischen Nährwert, in den Kommentaren den Code wiederzukäuen – guter Code (insbesondere in Python) braucht eigentlich überhaupt keine Kommentare. Wenn Sie etwas kommentieren, dann tun Sie es, weil es unoffensichtlich ist: Zum Beispiel könnte ein bestimmtes Problem in Ihrem Programm eine auf den ersten Blick überkomplizierte Lösung erfordern, weil die offensichtliche simple Lösung einen subtilen Fehler hat. In diesem Fall ist ein Kommentar nötig, damit kein wohlmeinender Kollege daherkommt und denkt, Ihnen einen Gefallen zu tun, indem er Ihren Code »vereinfacht«. Ebenfalls sinnvoll sind »High-Level«-Kommentare, die zum Beispiel für Funktionen angeben, was die Funktion tun soll und wie Parameter und Resultat aussehen müssen. – Das andere Problem mit Kommentaren, die den Code wiederkäuen, ist, dass über kurz oder lang der Code und die Kommentare auseinanderdriften und dann nicht mehr sofort klar ist, wer von beiden recht hat: Haben Sie korrekten Code mit einem falschen Kommentar, oder stimmt der Kommentar und jemand hat sich im Code vertan?

- Python 3 nimmt standardmäßig an, dass Programmcode gemäß UTF-8 codiert ist. Sollten Sie etwas Anderes benutzen (Pfui über Sie!), dann können Sie das Python durch einen speziell formatierten Kommentar am Dateianfang mitteilen. Zum Beispiel:

```
# coding=<Name der Codierung>
# -*- coding: <Name der Codierung> -*-           Emacs versteht das
# vim: set fileencoding=<Name der Codierung> :     Vim versteht das
```

Sie brauchen nur eine dieser Zeilen. Gängige Zeichencodierungen (außer UTF-8) sind zum Beispiel latin-1 oder iso-8859-15.



Genaugenommen muss die erste oder zweite Zeile der Datei auf den regulären Ausdruck »coding[:]=]\s*([\w.]+)« passen. Ist das der Fall, gilt das, was von der ersten Gruppe des Ausdrucks (den runden Klammern) gefunden wurde, als Name der Zeichencodierung. Wenn Python diese Zeichencodierung nicht kennt, dann wird ein Fehler gemeldet. (Wenn das alles für Sie »böhmische Dörfer« sind, dann lassen Sie sich keine grauen Haare wachsen.)



Python 2 geht davon aus, dass Programmcode dem ASCII (128 Zeichen) folgt. Das heißt, sobald Sie irgendwelche Sonderzeichen benutzen, die nicht aus dem traditionellen Zeichenvorrat stammen, müssen Sie einen Codierungs-Kommentar am Dateianfang haben.

Sie können das Beispielprogramm ausprobieren, indem Sie den Python-Interpreter aufrufen und den Namen der Programmdatei als Parameter übergeben:

```
$ python hallo.py
Hallo Welt!
```



Bei diesem Programm ist es noch egal, ob Sie es mit Python 2 oder Python 3 aufrufen. Typische Linux-Distributionen haben heutzutage beide Versionen installiert, und Sie erhalten in der Regel über das Kommando `python2` und `python3` die entsprechende Version. Wenn Sie herausfinden wollen, welche Geschmacksrichtung von Python Sie bekommen, wenn Sie einfach nur `python` sagen, dann versuchen Sie etwas wie

```
$ python -V
Python 2.7.10
```



Wenn Sie die Voreinstellung nicht mögen, dann hindert Sie niemand daran, in Ihrer Shell etwas wie

```
$ alias python=python3
```

zu sagen. Das symbolische Link, das in Ihrem Linux dafür sorgt, dass `python` eigentlich `python2` ist, sollten Sie nur mit großer Vorsicht verbiegen – es könnte sein, dass andere Programme auf Ihrem System unter dem Namen `python` einen Python-2-Interpreter vermuten und dann in Schwierigkeiten geraten.



Die meisten Linux-Distributionen sind dabei, Python 3 zum Standard zu machen. Wie weit dieser Vorgang fortgeschritten ist, müssen Sie im Zweifelsfall gezielt erforschen. (Vielleicht mögen Sie ja auch mithelfen; die meisten Distributionen würden sich freuen.)

Die erste Zeile in unserem Beispiel sorgt dafür, dass Sie unter Linux (oder Unix) das Programm aufrufen können, ohne dass Sie beim Aufruf einen expliziten Python-Interpreter benennen müssen. Dazu müssen Sie die Datei zunächst ausführbar machen:

```
$ chmod +x hallo.py
```

Anschließend können Sie das Programm direkt über seinen Namen aufrufen:

```
$ ./hallo.py
Hallo Welt!
```



Die explizite Verzeichnisangabe ist normalerweise nötig, weil es als ausgesprochen ungeschickt gilt, das aktuelle Verzeichnis nach ausführbaren Programmen zu durchsuchen (Sie machen sich dadurch verwundbar gegenüber »trojanischen Pferden« mit Namen wie `ls`). Wenn Sie Ihre Python-Programme ohne eine Pfadangabe aufrufen wollen, dann machen Sie sich ein Verzeichnis namens `bin` in Ihrem Heimatverzeichnis, kopieren die Python-Programme dort hinein und nehmen es in Ihre `PATH`-Variable auf:

```
$ mkdir $HOME/bin
$ ln -s $(pwd)/hallo.py $HOME/bin/hallo
$ PATH=$HOME/bin:$PATH
$ hallo
Hallo Welt!
```

(Damit das Verzeichnis `bin` dauerhaft im `PATH` landet, müssen Sie natürlich in einer Datei wie `.profile` eine entsprechende Zuweisung einbauen. Aber schauen Sie vorher mal genau nach – vielleicht stellen Sie ja zu Ihrer Genugtuung fest, dass Ihre Distribution das schon übernimmt, wenn ein `bin`-Unterverzeichnis in Ihrem Heimatverzeichnis existiert.)



Verkneifen Sie sich den Programmname `test`. Das gibt in der Regel Konflikte mit dem gleichnamigen eingebauten Shellkommando bzw. dem gleichnamigen Systemkommando, die nur schwer zu diagnostizieren sind, da das Shellkommando `test` keine oder nur kryptische Fehlermeldungen ausgibt. Wir haben Sie gewarnt.

Übungen



1.1 [!2] Wenn Sie es noch nicht gemacht haben: Geben Sie das Beispielprogramm aus Bild 1.1 ein und speichern es als `hallo.py` ab. Rufen Sie `hallo.py` auf und überzeugen Sie sich, dass es funktioniert.



1.2 [!1] Machen Sie `hallo.py` ausführbar und stellen Sie sicher, dass es auch ohne einen auf der Kommandozeile explizit benannten Python-Interpreter läuft.

1.4 Python interaktiv

Wenn Sie den Python-Interpreter aufrufen, ohne den Namen einer Programmdatei zu übergeben, dann bekommen Sie eine interaktive Eingabeaufforderung:

```
$ python3
Python 3.4.3+ (default, Jun 2 2015, 14:09:35)
[GCC 4.9.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

(Die letzte Zeile enthält den Cursor, hier angedeutet durch `>_<`.)

Diese Eingabeaufforderung verrät Ihnen freundlicherweise die Versionsnummer und das Übersetzungsdatum dieses Interpreters. Die drei Größerzeichen (`>>><`) sind Pythons Einladung an Sie, Code einzutippen, der dann auch direkt ausgeführt wird, bevor Sie erneut nach einem Kommando gefragt werden:

```
>>> print("Hallo Welt!")
Hallo Welt!
>>> _
```

Im Rahmen dieses Kurses werden wir den interaktiven Modus von Python sehr intensiv für Experimente nutzen. In der Unterlage geben wir die letzte Zeile (`>>><`) aber aus Platzgründen nicht mehr an.

Sie können den interaktiven Modus von Python verlassen, indem Sie auf der Standardeingabe das Dateiende signalisieren. Drücken Sie dazu die Taste `Strg`, halten Sie sie fest und drücken Sie gleichzeitig die Taste `d`.

Übungen



1.3 [!1] Rufen Sie Python im interaktiven Modus auf. Was passiert, wenn Sie bei der Eingabeaufforderung `help`, `copyright`, `credits` oder `license` angeben?

Zusammenfassung

- Die Programmiersprache Python wurde Ende der 1980er Jahre von Guido van Rossum erfunden und wird heute als Open-Source-Projekt von der »Python Software Foundation« weiterentwickelt.
- Python ist eine moderne Programmiersprache mit vielen netten Eigenschaften und wenigen Schwächen.
- Python-Programme stehen in normalen Textdateien. Üblich ist die Endung » .py«.
- Ohne Parameter aufgerufen startet der Python-Interpreter im interaktiven Modus.



2

Einfache Datentypen

Inhalt

2.1	Numerische Datentypen und Literale	22
2.2	Numerische Ausdrücke	24
2.3	Variable	27
2.4	Zeichenketten	29
2.4.1	Zeichenketten-Literale	29
2.4.2	Zeichenketten-Ausdrücke.	31

Lernziele

- Mit ganzen Zahlen und Gleitkommazahlen umgehen können
- Variable benutzen können
- Einfache Operationen für Zeichenketten beherrschen

Vorkenntnisse


- Interaktiver Gebrauch von Python (Kapitel 1)
- Erfahrung mit anderen Programmiersprachen ist von Vorteil


2.1 Numerische Datentypen und Literale

Python unterscheidet drei verschiedene grundlegende Arten von Zahlen: Ganze Zahlen, Gleitkommazahlen und komplexe Zahlen.

Ganze Zahlen Ganze Zahlen sind sozusagen das Brot- und Buttergeschäft der Computerei. In Python können Sie mit ihnen ziemlich intuitiv umgehen. Ganzzahlige Konstanten im Programmtext (vulgo »Literale«) sind einfach Folgen von Ziffern, die dann als Dezimalzahlen interpretiert werden:


```
1
4711
3141926535
```


negative ganze Zahlen  Natürlich dürfen Sie auch negative ganze Zahlen benutzen, indem Sie ein Minuszeichen links vor die Zifferfolge stellen. Das Minuszeichen ist aber strenggenommen nicht Teil des Literals, sondern ein mathematischer Operator.

führende Nullen  Verkniefen Sie sich führende Nullen – aus Gründen, die wir gleich noch sehen werden. In Python 3 sind sie sowieso verboten.


Andere Zahlensysteme Wenn Sie maschinennahe Programmierung betreiben, ist es oft nützlich, Zahlen in einem anderen Zahlensystem als dem sonst üblichen Dezimalsystem anzugeben. Python unterstützt das Binär-, das Oktal- und das Hexadezimalsystem:


```
0b1111011          123 als binäres Literal (Basis 2)
0o173              123 als oktales Literal (Basis 8)
0x7b               123 als hexadezimaler Literal (Basis 16)
```

 Die Präfixe `0b`, `0o` und `0x` dürfen Sie auch als `0B`, `0O` bzw. `0X` schreiben. Zumindest von `0O` würden wir Ihnen allerdings dringend abraten, da das große `O` einer Null – je nach Schrifttype – unangenehm ähnlich sieht. (Auch `0B` finden wir nicht unbedingt prall.)

 Python 2 unterstützt für oktale Literale auch die althergebrachte, von der Programmiersprache C geerbte Schreibweise mit führender Null (`0173` in unserem Beispiel). Wir finden das nicht offensichtlich genug und empfehlen darum, lieber `0o173` zu benutzen¹.

Lange ganze Zahlen Ganzahlige Literale (egal in welchem Zahlensystem), die größer sind als die größte direkt darstellbare positive ganze Zahl – auf 32-Bit-Rechnern gemeinhin `2.147.483.647` –, werden als »lange ganze Zahlen« betrachtet. Lange ganze Zahlen sind nur durch den verfügbaren Speicher begrenzt.

 »Einfache« ganze Zahlen decken mindestens den Bereich von `-2.147.483.648` bis `2.147.483.647` ab – wenn Ihr Computer eine »natürliche Wortlänge« von mehr als 32 Bit hat, auch gerne mehr, aber nie weniger.

 Python 3 macht keinen Unterschied zwischen »einfachen« und langen ganzen Zahlen. Wenn der Zahlenbereich für »einfache« ganze Zahlen überschritten wird, rechnet Python 3 automatisch mit langen ganzen Zahlen weiter.

¹Das war übrigens der Grund für die Warnung vor führenden Nullen weiter oben. Wenn Sie dezimale Literale aus Gründen der Optik mit führenden Nullen versehen, können Sie interessante Überraschungen erleben. Wir drücken Ihnen die Daumen, dass Sie in Ihren Konstanten eine gelegentliche Ziffer 8 oder 9 haben, damit Ihnen Ihr Programm frühzeitig um die Ohren fliegt, anstatt später merkwürdige Ergebnisse zu liefern.



Sie können in Python 2 ein ganzzahliges Literal ausdrücklich zu einer langen ganzen Zahl erklären, indem Sie an die Ziffernfolge ein L anhängen:

1L	Lange ganze Zahl
----	------------------

(1 wäre auch erlaubt, aber das ist zu leicht mit einer 1 zu verwechseln. Lassen Sie's also sein.) In Python 3 ist das nicht möglich, weil unnötig.



Das Rechnen mit langen ganzen Zahlen ist deutlich langsamer als das Rechnen mit »einfachen« ganzen Zahlen.

Gleitkommazahlen Neben ganzen Zahlen unterstützt Python auch Gleitkommazahlen, salopp gesagt also Zahlen mit einem Nachkommaanteil. Das »Komma« wird nach US-amerikanischer Manier aber als Punkt geschrieben: Komma als Punkt

1.5

Bei Bedarf können Sie den Teil der Zahl vor oder nach dem Dezimalpunkt auch weglassen (dann wird »0« angenommen) – aber nicht beide.

.5	Dasselbe wie 0.5
1.	Dasselbe wie 1.0

Außerdem können Sie einen »Exponenten« angeben. Wenn f die »Mantisse« (die Ziffern ohne den Exponenten) darstellt und E den Exponenten, dann repräsentiert das Literal den Wert $f \cdot 10^E$: Exponent

1e100	10^{100} – ein Googol
1e+100	Dasselbe nochmal
1e-100	10^{-100} – eine sehr kleine Zahl



Statt eines kleinen e können Sie auch ein großes E verwenden, um den Exponenten einzuleiten.

Auch wenn ganzzahlige und Gleitkomma-Literale im Programmtext ziemlich ähnlich aussehen, stehen sie doch für sehr unterschiedliche Daten: Normale Ganzzahlen werden im Computer genau dargestellt (mit den eingebauten Ganzzahlen des Prozessors, falls sie nicht zu groß sind, sonst mit Python-internen Datenstrukturen), Gleitkommazahlen dagegen entsprechen den eingebauten Gleitkommazahlen »doppelter Genauigkeit«.




Was die genauen Eigenschaften der Gleitkommazahlen Ihres Rechners anbetrifft, sind Sie dessen C-Implementierung ausgeliefert (vor allem wenn es um gültige Zahlenbereiche oder das Verhalten bei einer Bereichsüberschreitung geht). Gängige Prozessoren unterstützen heutzutage Gleitkomma-Arithmetik gemäß IEEE 754; für doppelte Genauigkeit (eine Gleitkommazahl nimmt 8 Byte ein) haben Sie 52 Bit für die Mantisse, 11 Bit für den Exponent und 1 Bit für das Vorzeichen zur Verfügung. Das erste Bit der Mantisse ist nach Definition immer 1, so dass Sie unter dem Strich eine 53 Bit-Mantisse erhalten². Sie dürfen also mit 15–17 signifikanten Dezimalstellen rechnen, wobei der dezimale Exponent zwischen –308 und 308 liegen kann. Eigenschaften der Gleitkommazahlen



Vielleicht werden Sie sich fragen, warum Python nicht (auch) Gleitkommazahlen »einfacher Genauigkeit« anbietet (so wie zum Beispiel die Programmiersprache C). Die Antwort darauf lautet, dass Zahlen intern nicht als Zahlen, sondern als Python-Objekte dargestellt werden und dass der Zusatzaufwand, um aus einer Gleitkommazahl ein Python-Objekt zu machen, so groß ist, dass die Extramühe, zwei verschiedene Gleitkomma-Formate zu unterstützen, die Ersparnis von 4 Byte pro Zahl nicht wert wäre. (RAM ist heute ja ziemlich billig.)


²Spezialfälle wie »denormalisierte Werte« lassen wir hier mal außer Acht.

Vorbehalte  Auch bei Python gelten die üblichen Vorbehalte im Umgang mit computerisierten Gleitkommazahlen. Zum Beispiel sollten Sie, wenn Sie zehnmal den Wert 0.1 addieren, nicht damit rechnen, dass exakt 1.0 herauskommt. Überhaupt ist es in der Regel keine gute Idee, zwei Gleitkommazahlen auf Gleichheit zu testen. Dazu sagen wir später mehr.

Komplexe Zahlen Elektrotechniker und andere Ingenieure werden mit Genugtuung feststellen, dass Python auch ohne Zusatzbibliotheken mit komplexen Zahlen umgehen kann. Imaginäre Literale bestehen einfach aus Gleitkomma- oder ganzzahligen Literalen mit einem angehängten »j« (oder »J«):

1j	oder auch $\sqrt{-1}$
----	-----------------------

Wenn Sie eine komplexe Zahl mit einem von Null verschiedenen Realteil hinschreiben wollen, müssen Sie den Real- und den Imaginärteil addieren (und das erklären wir im nächsten Abschnitt).

 Intern sind komplexe Zahlen einfach Paare von zwei Gleitkommazahlen. Das heißt, alles, was wir weiter oben über die Darstellung, gültigen Wertebereiche usw. von Gleitkommazahlen gesagt haben, gilt sinngemäß auch für komplexe Zahlen.

2.2 Numerische Ausdrücke

»Rechnen« ist eine der naheliegendsten Anwendungen für Programmiersprachen wie Python – und in der Tat gibt der Python-Interpreter im interaktiven Modus einen sehr bequemen »Taschenrechner« ab (obwohl Sie Python auf einem Android-Handy laufen lassen müssen, damit es wirklich ein *Taschen*-Rechner wird):

```
>>> 1+2*3
7
>>> (1+2)*3
9
```

Grundrechenarten Wie Sie sehen, kommt Python mit den gängigen Grundrechenarten zurecht (multipliziert wird, wie in anderen Programmiersprachen, mit dem Sternchen) und beachtet auch die gängigen mathematischen Gepflogenheiten: Punktrechnung geht vor Strichrechnung, und mit Klammern können Sie angeben, was zuerst ausgewertet werden soll.

Natürlich kann Python auch subtrahieren:

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

Negative Zahlen sind auch kein Problem – setzen Sie einfach ein Minuszeichen direkt vor die Zahl.

Division Interessant ist das Thema »Division«:

>>> 10/5	2.0	Python 3
	2	Python 2
>>> 11/5	2.2	Python 3
2 Python 2		
>>> 11./5	2.2	Immer

In Python 3 liefert eine Division immer ein Gleitkommareultat, unabhängig vom Typ der Operanden. In Python 2 hängt das Resultat der Division davon ab, ob die Operanden ganze Zahlen oder Gleitkommazahlen sind: Handelt es sich um ganze Zahlen, ist das Ergebnis wieder ganzzahlig; ist mindestens einer der Operanden eine Gleitkommazahl, kommt ein Gleitkommaergebnis heraus.



Division durch Null ist natürlich verboten. Das signalisiert Python Ihnen durch eine »Exception«:

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

Exceptions führen in der Regel zum Programmabbruch; nur im interaktiven Python-Interpreter sind sie nicht weiter schlimm, weil es mit der nächsten Eingabeaufforderung weitergeht. Grundsätzlich können Sie in Ihren Programmen Exceptions abfangen, um Fehlersituationen auszubügeln. Das würde hier aber etwas zu weit führen.



Genaugenommen ist das Ergebnis einer Division von zwei ganzen Zahlen in Python 2 die größte ganze Zahl, die kleiner oder gleich dem »echten« Ergebnis ist. Zum Beispiel:

```
>>> -10/5
-2
>>> -11/5.
-2.2
>>> -11/5
-3
```

Die größte ganze Zahl $\leq -2,2$

Python 3 hat für den (gängigen) Fall, dass ein ganzzahliges Ergebnis gewünscht wird, den speziellen Divisionsoperator `//`:

```
>>> 11//5
2
>>> -11//5
-3
```

`//` benimmt sich wie der Divisionsoperator `/` von Python 2.



Wenn Sie Python 2 benutzen, sich aber schon an das Divisionsverhalten von Python 3 gewöhnen wollen, können Sie mit dem Kommando

```
from __future__ import division
```

am Anfang einer Datei das Python-3-Verhalten einschalten. Das funktioniert ab Python 2.2.

Der Operator `%`, auch bekannt als »Modulo-Operator«, liefert den Rest bei ganzzahliger Division: Modulo-Operator

```
>>> 7 % 3
1
>>> 7. % 3.
1.0
>>> -7 % 3
2
```

Huch.



Das letzte Beispiel ist möglicherweise etwas verwirrend. Um es zu verstehen, müssen Sie wissen, dass in Python die (ganzzahlige) Division und die Modulo-Operation wie folgt zusammengehören:

$$x = (x/y) * y + (x \% y)$$

Damit die Gleichung für $x = -7$ und $y = 3$ gilt, muss $x \% y$ den Wert 2 haben.

Potenzen Mit dem Operator `**` können Sie Potenzen bilden:

```
>>> 2**3
8
>>> 2**-3
0.125
```

Assoziativität Alle diese Operatoren sind linksassoziativ, das heißt, Ausdrücke wie »1+2+3« oder »4-5-6« werden respektive wie »(1+2)+3« oder »(4-5)-6« interpretiert. Die Ausnahme ist der Potenzoperator `**`, der rechtsassoziativ ist – »2**3**4« ist dasselbe wie »2**(3**4)«.



Auch wenn es dem Menschen auf der Straße willkürlich vorkommen mag: Das ist so Usus in der Mathematik.

mathematische Funktionen Python unterstützt auch einige nützliche mathematische Funktionen, die Sie auf Zahlen anwenden können:

```
>>> abs(-5)                                Betrag
5
>>> int(3.14)                               Umwandeln in ganze Zahl
3
>>> int(-3.14)                             int rundet immer in Richtung Null
-3
>>> max(1,7,5)                             Maximum
7
>>> min(5,2,1)                             Minimum
1
>>> round(1.123)                           Runden
1.0
>>> round(1.234, 1)                       Runden auf bestimmte Stellenzahl
1.2
```



Der Unterschied zwischen `round()` und `int()` ist, dass `round()` eine Gleitkommazahl liefert und `int()` eine Ganzzahl – auch wenn Sie mit `round()` auf »null Nachkommastellen« runden. Wenn zwei mögliche Rundungsergebnisse gleich weit von der zu rundenden Zahl entfernt sind, rundet `round()` von Null weg; `round(-0.5)` ist also `-1.`, wenn Sie auf »null Nachkommastellen« runden³.

math Für größere Geschütze der mathematischen Art müssen Sie auf das `math`-Modul zurückgreifen, ungefähr so:

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sin(math.pi/2)
1.0
>>> math.log(10)                            Basis e
2.302585092994046
```

³... sagt die Theorie. Im wirklichen Leben ist Python durchaus für eine Überraschung gut – `round(3.55,1)` zum Beispiel ergibt `3.5`, nicht, wie man eigentlich erwarten würde, `3.6`. Das liegt daran, dass eine Python-Gleitkommazahl einen Wert wie `3.55` nicht exakt darstellen kann und die binäre Darstellung näher an `3,5` als an `3,6` liegt.

(Eine genauere Beschreibung des Inhalts von `math` können Sie der Python-Dokumentation entnehmen.)

Übungen



2.1 [!1] Was ist der Unterschied zwischen `1e100` und `10**100`?



2.2 [1] Verwenden Sie den Satz des Pythagoras, um die Länge der Hypotenuse a eines rechtwinkligen Dreiecks zu bestimmen, dessen Katheten b und c jeweils die Länge 6 und 8 haben. (Ob Nanometer oder Lichtjahre, bleibt hier Ihrer Fantasie überlassen.) *Tipp:* $a^2 = b^2 + c^2$, und die Quadratwurzelfunktion heißt `math.sqrt`.

2.3 Variable

Die allermeisten Programmiersprachen sind sich einig, dass es eine gute Idee ist, Objekten (wie Zahlenwerten) einen Namen geben zu können – das macht Programme übersichtlicher, leichter lesbar und besser wartbar. Es ist allemal vernünftiger, etwas zu sagen wie

```
>>> pi = 3.141592654
>>> u = 2*pi*5           Umfang eines Kreises mit Radius 5
>>> a = pi*5*5         Fläche eines Kreises mit Radius 5
```

als etwas wie

```
>>> u = 2*3.141592654*5
>>> a = 3.141592654*5*5
```

(Im ersten Beispiel haben Sie auch weniger Arbeit, wenn der Wert von π sich mal ändert.)



Weil Sie aufgepasst haben, wissen Sie natürlich noch, dass Python im Modul `math` die vordefinierte Variante `pi` bereithält. Sie müssen sich also in diesem Fall gar keinen Stress machen, und mit

```
from math import pi
```

am Anfang Ihres Programms müssen Sie sogar nicht mal `math.pi` tippen wie im vorigen Abschnitt.

Eigentlich haben Sie das Wichtigste schon gesehen: Wenn Sie einem Literal oder dem Ergebnis eines mathematischen Ausdrucks einen Namen geben wollen, dann schreiben Sie einfach den Namen hin, ein Gleichheitszeichen (mit oder ohne Leerzeichen drumherum, das ist egal) und dann den Literal oder Ausdruck – eben wie in

```
>>> drei=3
```

Namen in Python dürfen Buchstaben, Ziffern und die Unterstreichung enthalten, das erste Zeichen eines Namens darf aber keine Ziffer sein. Groß- und Kleinschreibung sind bedeutsam. Die Länge eines Namens ist nicht beschränkt (außer durch Ihre persönliche Bequemlichkeit). Namen



»Buchstabe« steht in Python 2 für die Zeichen `A...Z` bzw. `a...z` – keine Umlaute oder sonstigen fremdländischen Zeichen. Python 3 ist da wesentlich liberaler – es erlaubt praktisch alles, was wie ein Buchstabe aussieht. Es hindert Sie also niemand daran, in Frankreich Variablennamen mit Akzenten

oder in Russland kyrillische Variablenamen zu verwenden – von chinesischen, japanischen, koreanischen oder arabischen Zeichen mal ganz zu schweigen. (Ob Sie Ihren Kollegen damit uneingeschränkt einen Gefallen tun, steht allerdings auf einem anderen Blatt.)



Die Details stehen im Abschnitt 2.3 des Python-Sprachhandbuchs. Unter <http://www.dcl.hpi.uni-potsdam.de/home/loewis/table-3131.html> finden Sie eine (nicht normative) Liste der Zeichen aus Unicode 4.1, die für Python-Namen in Frage kommen.

Schlüsselwörter Einige ansonsten plausibel wirkende Namen werden von Python als **Schlüsselwörter** für interne Zwecke verwendet und sind als Namen von Variablen nicht erlaubt. In Python 3 sind das

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



In Python 2 sind `exec` und `print` Schlüsselwörter und `True`, `False` und `None` nicht. `nonlocal` ist in Python 2 auch kein Schlüsselwort. Machen Sie sich aber keinen zu großen Kopf.

Namen mit Sonderbedeutung Außerdem haben einige Klassen von Namen eine besondere Bedeutung. Diese erkennt man an den Unterstreichungen am Anfang oder Ende:

`_IRGENDWAS` Namen, die mit einer Unterstreichung anfangen, werden von einem Modul nicht automatisch exportiert.



Der Name `»_«` bezieht sich im interaktiven Modus implizit auf den zuletzt berechneten Wert. Das macht das Leben bequemer:

```
>>> 1+2*3
7
>>> 4*_+5
33
```

In Programmen hat `_` keine Sonderbedeutung.



`_` wird in Programmen oft verwendet, um Zeichenketten zu kennzeichnen, die im Zuge der »Internationalisierung« bzw. »Lokalisierung« eines Programms übersetzt werden sollen. Man schreibt also

```
print _("Hallo Welt")
```

und sorgt dafür, dass `_` für eine Funktion steht, die die Übersetzung der Zeichenkette in einer geeigneten Datenbank nachschlägt. Vorgeschrieben ist das aber nicht.

`__IRGENDWAS__` Namen mit je zwei Unterstreichungen am Anfang und am Ende haben eine »magische« Sonderbedeutung, die vom Python-Interpreter festgelegt wird. Sie können damit zum Beispiel Konstruktoren und Destruktoren für Objekte implementieren oder Operatoren überladen.

`__IRGENDWAS` Namen mit zwei Unterstreichung am Anfang (aber keinen zwei Unterstreichungen am Ende) gelten als »klassenprivat«. Wenn Sie objektorientiert mit Klassen und Unterklassen arbeiten, können Sie so vermeiden, dass interne Attribute verschiedener Klassen miteinander kollidieren. Wir schauen uns das im Kapitel über objektorientierte Programmierung genauer an.

2.4 Zeichenketten

2.4.1 Zeichenketten-Literale

Python kann nicht nur mit Zahlen umgehen, sondern auch mit Zeichenketten (das sollte Sie nicht mehr wirklich überraschen). Schauen wir uns zunächst an, wie Zeichenketten-Literale aussehen:

```
>>> "Hallo"
'Hallo'
>>> 'Hallo'
'Hallo'
```

Zeichenketten-Literale werden entweder mit einfachen oder mit doppelten Anführungszeichen »eingerahmt«. Die Anführungszeichen sind selber nicht Teil des Literals. Die betreffenden Anführungszeichen dürfen im Literal selbst nicht vorkommen:

```
>>> "Er sagte "Huch" und fiel in Ohnmacht"
SyntaxError: invalid syntax
>>> 'Er sagte "Huch" und fiel in Ohnmacht'
'Er sagte "Huch" und fiel in Ohnmacht'
>>> "Er sagte 'Huch' und fiel in Ohnmacht"
"Er sagte 'Huch' und fiel in Ohnmacht"
```

Besser.
Auch gut.

Sie können die Anführungszeichen im Literal auch hinter einem Rückstrich (*backslash*) verstecken:

```
>>> "Er sagte \"Huch\" und fiel in Ohnmacht"
'Er sagte "Huch" und fiel in Ohnmacht'
```

Alternativ können Sie »lange« Zeichenketten benutzen. Die beginnen mit drei Anführungszeichen (statt einem) und enden mit drei weiteren zu denen am Anfang passenden Anführungszeichen. In ihrem Innern dürfen nicht nur (einzeln oder paarweise) dieselben Anführungszeichen stehen, sondern auch Zeilenumbrüche:

```
>>> """Er sagte "Huch"
... und fiel in Ohnmacht"""
'Er sagte "Huch"\nund fiel in Ohnmacht'
```

Im vorigen Beispiel können Sie auch sehen, wie Sie einen Zeilenumbruch in einer literalen Zeichenkette unterbringen: mit der Zeichenkombination »\n«. Allgemein gibt es eine ganze Reihe von Sonderzeichen, die über solche Rückstrich-Kombinationen zu erreichen sind; betrachten Sie Tabelle 2.1.

Python 2 und Python 3 unterscheiden sich darin, wie sie mit Unicode umgehen. In Python 3 sind alle Zeichenketten-Literale automatisch Unicode-Zeichenketten. Python 3 geht auch davon aus, dass Programmdateien gemäß UTF-8 codiert sind; eine explizite Angabe der Zeichencodierung brauchen Sie also nur, wenn das nicht der Fall ist⁴. Zum Ausgleich unterstützt Python 3 sogenannte **Byte-Strings**, die durch ein vorgestelltes **b** (oder **B**) kenntlich gemacht werden:

```
b"Hallo Welt!"
```

Byte-Strings bestehen in Python 3, wie der Name sagt, aus Bytes, die nicht weiter interpretiert werden. ASCII-Zeichen stehen zum größten Teil für sich (die üblichen Sequenzen wie in Tabelle 2.1 gelten, außer \u, \U und \N), aber wenn Sie ein Byte mit einem Wert größer 127 darstellen wollen, müssen Sie \x oder die Oktalschreibweise benutzen.

⁴Für UTF-8-codierte Dateien schadet sie aber auch nichts.

Tabelle 2.1: Rückstrich-Kombinationen in Zeichenketten-Literalen

Kombination	Bedeutung
<code>\\</code>	Rückstrich (<code>\</code>)
<code>\'</code>	Einfaches Anführungszeichen (<code>'</code>)
<code>\"</code>	Doppeltes Anführungszeichen (<code>"</code>)
<code>\a</code>	Signalton (ASCII BEL)
<code>\b</code>	Rückschritt (ASCII BS)
<code>\f</code>	Seitenvorschub (ASCII FF)
<code>\n</code>	Zeilenschritt (ASCII LF)
<code>\r</code>	Wagenrücklauf (ASCII CR)
<code>\t</code>	Horizontaler Tabulator (ASCII TAB)
<code>\v</code>	Vertikaler Tabulator (ASCII VT)
<code>\ooo</code>	Zeichen mit dem oktalen numerischen Code »ooo«
<code>\xhh</code>	Zeichen mit dem hexadezimalen numerischen Code »hh«
<code>\uxxxx</code>	Zeichen mit dem 16-Bit-Hexadezimalcode »xxxx«*
<code>\uxxxxxxx</code>	Zeichen mit dem 32-Bit-Hexadezimalcode »xxxxxxx«*
<code>\N{Name}</code>	Zeichen mit dem Namen <i>Name</i> *

* = Nur in Unicode-Literalen (Python 2) bzw. Zeichenketten-Literalen (Python 3)

Python 2 trennt zwischen ASCII- und Unicode-Strings, und der Standard ist ASCII. Das heißt, es wird davon ausgegangen, dass ein Literal wie

```
"Hallo Welt"
```

nur ASCII-Zeichen enthält. Sollen Zeichen vorkommen, die nicht dem Standard-7-Bit-ASCII entstammen, müssen Sie das voranmelden:

```
u"Gr\u00f6ezi Welt"
```

Das `u` vor dem öffnenden Anführungszeichen weist Python darauf hin, dass es sich hier um einen Unicode-Literal handelt; fehlte es, dann würde das `»\u00fc«` eine Fehlermeldung auslösen.



Statt `u` dürfen Sie auch `U` schreiben, um einen Unicode-Literal einzuleiten. Das ist allerdings ziemlich unüblich.



Vorgesetzte `us` (oder `Us`) werden von Python 3 nur noch aus Kompatibilitätsgründen unterstützt⁵. Standardmäßig sind ja alle Zeichenketten Unicode.



Dass in Python 2 in einem Zeichenketten-Literal ein Unicode-Zeichen stehen darf, heißt noch lange nicht, dass Sie es auch direkt eintippen dürfen: Etwas wie

```
u"Grüezi Welt"
```

im Programmtext würde Python 2 anmeckern, da es nicht auf Antrieb weiß, wie das Zeichen `»ü«` zu interpretieren ist (es gibt ja grundsätzlich diverse Möglichkeiten – ISO-Latin-1 oder UTF-8 könnten Ihnen in den Sinn kommen).



Dieses Problem können Sie beheben, indem Sie Python mitteilen, welche Zeichencodierung in einer Datei benutzt wird. Dazu müssen Sie in die erste oder zweite Zeile der Datei etwas schreiben wie

⁵Vor Python 3.3 ging das gar nicht – aber dann wurde es wieder eingeführt, um das Erstellen von Programmen zu erleichtern, die sowohl mit Python 2 als auch Python 3 funktionieren sollen.

```
# -*- coding: utf-8 -*-           Emacs-freundlich
# vim:fileencoding=utf-8         Vi(m)-freundlich
```

(Python sucht nach `coding`, gefolgt von einem Doppelpunkt oder Gleichheitszeichen, möglicherweise gefolgt von Freiplatz, gefolgt vom Namen einer Zeichencodierung.)



Alternativ können Sie auch das Unicode-Byteorder-Zeichen – die Bytes `\xef\xbb\xbf` – an den Anfang Ihrer Datei setzen (Windows Notepad macht das für Sie), damit Python Ihre Datei als UTF-8 erkennt.

In Python können Sie mehrere Zeichenketten-Literale direkt nebeneinanderstellen. Diese gelten dann als ein großes Literal, das die Zeichenkette repräsentiert, die Sie erhalten, wenn Sie die Werte der einzelnen Literale lückenlos aneinanderhängen. Die einzelnen Literale dürfen zum Beispiel unterschiedliche Anführungszeichen-Konventionen verwenden:

```
>>> "Hallo" 'Welt'
'HalloWelt'
```

Auch eine Mischung aus Unicode- und einfachen Literalen ist erlaubt:

```
>>> u"Gr\u00f6ezi" 'Welt'
u'Gr\xfcenziWelt'
>>> print(u"Gr\u00f6ezi" 'Welt')
GrüeziWelt
```

2.4.2 Zeichenketten-Ausdrücke

Auch Zeichenketten können Sie in Python gezielt manipulieren, etwa indem Sie sie aneinanderhängen:

```
>>> hw = "Hallo Welt"
>>> print(hw + 'enbummler')
Hallo Weltenbummler
```

»Multiplikation« einer Zeichenkette mit einer Zahl ist auch erlaubt:

```
>>> "Bla" * 3           So herum ...
'BlaBlaBla'
>>> 3 * "Bla"         ... oder so herum
'BlaBlaBla'
```

Die Länge einer Zeichenkette ermitteln Sie mit der Funktion `len()`:

Länge

```
>>> len("Hallo")
5
```

Zeichenketten sind in Python eine Form von »Folgen« (*sequence types*). Zu diesen zählen auch Listen und Tupel, die wir später kennenlernen werden. Es gibt diverse nützliche Operationen, die Sie auf Listen und Tupel anwenden können und die auch mit Zeichenketten funktionieren – wir kommen darauf zurück.

Folgen

Zusammenfassung

- Python unterstützt ganze Zahlen (»normale« und »lange«), Gleitkommazahlen und komplexe Zahlen.
- Python unterstützt die gängigen Grundrechenarten mit den üblichen Vorrangregeln (die Division ist ein bisschen trickreich).
- Diverse nützliche mathematische Funktionen sind entweder standardmäßig im Python-Interpreter realisiert oder über das Modul `math` zugänglich.
- Namen in Python dürfen Buchstaben, Ziffern und den Unterstrich enthalten (aber nicht mit einer Ziffer anfangen). Einige Schlüsselwörter sind tabu.
- Namen, die mit einem Unterstrich anfangen, haben eine Sonderbedeutung.
- Zeichenketten-Literale werden durch einfache oder doppelte Anführungszeichen begrenzt. »Lange« Literale dürfen Zeilenumbrüche enthalten.
- Zeichenketten unterstützen Aneinanderhängen, »Multiplikation« und Längenbestimmung.
- Zeichenketten in Python sind eine Art von Folgen



3

Verzweigungen und Schleifen

Inhalt

3.1	Verzweigungen mit <code>if</code>	34
3.1.1	Einstieg	34
3.1.2	Boolesche Ausdrücke	34
3.1.3	<code>else</code> und <code>elif</code>	37
3.2	Schleifen mit <code>while</code>	39
3.2.1	Wiederholungen	39
3.2.2	<code>break</code> und <code>continue</code>	41

Lernziele

- Boolesche Ausdrücke in Python verstehen
- Verzweigungen mit `if` verwenden können
- Schleifen mit `while` verwenden können

Vorkenntnisse

- Einfache Datentypen in Python (Kapitel 2)
- Erfahrung mit anderen Programmiersprachen ist von Vorteil

3.1 Verzweigungen mit if

3.1.1 Einstieg

Programmiersprachen werden eigentlich erst in dem Moment interessant, wo sie es erlauben, dass bestimmte Programmteile manchmal ausgeführt werden und manchmal nicht¹. Da Python eine interessante Programmiersprache ist, enttäuscht es uns in dieser Beziehung nicht. Das `if`-Kommando wertet eine Bedingung aus und führt ein Programmstück nur dann aus, wenn die Bedingung »wahr« ist:

```
if i > 10:
    print "i ist größer als 10!"
    j = 2*i+1
```

(Beachten Sie den Doppelpunkt am Zeilenende hinter der Bedingung.)

zusammengesetztes Kommando

Kommandofolge

Die `if`-Konstruktion ist ein Beispiel für ein **zusammengesetztes Kommando** in Python. Es besteht aus einem »Kopf« – dem `if`, der Bedingung und dem Doppelpunkt – gefolgt von einer **Kommandofolge** (engl. *Suite*). Eine Kommandofolge wiederum ist entweder eine Anzahl von gleich weit eingerückten Kommandos auf eigenen Zeilen unter dem Kopf (wie im Beispiel oben) oder aber eine durch Semikolons getrennte Anzahl von Kommandos auf derselben Zeile wie der Kopf, unmittelbar hinter dem Doppelpunkt:

```
if i > 10: print "i ist größer als 10!"; j = 2*i+1
```



Letzteres ist hin und wieder nützlich, aber nicht uneingeschränkt förderlich für die Lesbarkeit Ihres Codes. Verwenden Sie es sparsam und mit Vorbedacht.

Einrückung

Pythons Philosophie, Einrückung zur Darstellung der Programmstruktur zu verwenden, ist für viele Programmierer zunächst ungewohnt und abschreckend, aber hat durchaus Vorteile, weil sie Fehler verhindern kann.



Andere Programmiersprachen verwenden geschweifte Klammern oder Schlüsselwörter wie `BEGIN` und `END` – aber Sie sind trotzdem gehalten, Ihre Programme ordentlich einzurücken. Das ist eine potentielle Problemquelle, weil einerseits nicht garantiert ist, dass die Einrückung der tatsächlichen Programmstruktur gemäß der Klammern oder Schlüsselwörter entspricht, andererseits wir Programmierer uns gerne vom groben optischen Eindruck der Einrückung irreführen lassen, selbst wenn im Programm eigentlich etwas ganz Anderes steht. In Python ist beides dasselbe, so dass es keine Verwirrung geben kann.



Wie üblich taugt so etwas als Quelle hitziger Meinungsverschiedenheiten. Wir haben dazu nichts zu sagen außer dass Python eben so ist; wenn Sie die geschweiften Klammern wollen, dann müssen Sie halt in Perl oder PHP programmieren.

3.1.2 Boolesche Ausdrücke

Wahr und falsch

Der Ausdruck hinter dem `if` muss »wahr« sein, damit die vom `if` abhängige Kommandofolge ausgeführt wird. Als »wahr« betrachten wir in diesem Zusammenhang alles, was nicht »falsch« ist. »Falsch« wiederum sind Werte, die numerisch Null sind (egal, welche Typen – ganzzahlig, Gleitkomma oder komplex), und einige andere Werte, die Sie noch nicht gesehen haben.

¹Das andere, was man haben möchte, ist, bestimmte Programmteile in Abhängigkeit von Bedingungen wiederholen zu können ... aber dazu kommen wir später in diesem Kapitel.

Tabelle 3.1: Vergleichsoperatoren in Python

Operator	Bedeutung
==	Gleichheit
!=	Ungleichheit
<>	Ungleichheit (Python 2, verpönt)
<	Kleiner als
<=	Kleiner als oder gleich
>	Größer als
>=	Größer als oder gleich
is	Objektidentität
is not	Objekt-Nichtidentität



Ohne jetzt allzusehr vorgreifen zu wollen: Die anderen Möglichkeiten sind die vordefinierten Werte `False` und `None`, leere Zeichenketten und leere Container (wie Tupel, Listen, Dictionaries, Mengen usw.). Alles andere gilt als »wahr«.



`True` und `False` sind im Grunde nur vornehme Namen für die numerischen Werte 1 und 0. Python unterstützt einen besonderen numerischen Datentyp für Boolesche Werte². `False` und `True` sind die einzigen Booleschen Objekte in Python und benehmen sich für alle praktischen Zwecke wie 0 und 1, werden aber respektive als die Zeichenketten "False" und "True" ausgegeben.

Während etwas wie »1+2« absolut als »Boolescher« Ausdruck in einem `if` in Frage kommt, bieten **Vergleichsoperatoren** sich vielleicht eher an. Python unterstützt die gängigen Vergleichsoperatoren, die Sie vielleicht von anderen Programmiersprachen her kennen (siehe Tabelle 3.1).

Vergleichsoperatoren

Vergleichsoperatoren liefern immer einen der Werte `False` oder `True`. Beide Operanden müssen denselben Typ haben. Numerische Operanden werden notfalls umgewandelt – etwa von ganzen in Gleitkomma-Zahlen –, bis sie vergleichbar werden:

```
>>> 1 < 0
False
>>> 2 > 1.5
True
```

Zeichenketten werden »lexikografisch« verglichen, also Zeichen für Zeichen, wobei die Codepunkt-Nummern der Zeichen zählen (»@« ist zum Beispiel 64 und »a« ist 97). Python betrachtet dazu das erste Zeichen der beiden Zeichenketten; ist dies bei beiden gleich, fährt es mit dem zweiten Zeichen fort usw., bis es entweder eine Position erreicht, wo die beiden Zeichenketten sich unterscheiden, oder das Ende einer der Zeichenketten. Im ersteren Fall entscheiden die Zeichen an dieser Position über das Ergebnis, im letzteren gilt die kürzere Zeichenkette als »kleiner«, und wenn sie gleich lang sind (also beide auf einmal zu Ende), dann sind sie gleich.

Zeichenketten vergleichen



Komplexe Zahlen können gar nicht verglichen werden. In früheren Versionen von Python ging das, aber es ist mathematisch gesehen Unfug.



Python 2 erlaubt tatsächlich Vergleiche zwischen Objekten verschiedenen Typs. Allerdings passiert nicht unbedingt das, was man vermuten würde. In Vergleichen zwischen numerischen Typen und anderen (etwa Zeichenketten) gilt ein Objekt numerischen Typs immer als »kleiner«:

²Der britische Mathematiker, Philosoph und Logiker George Boole (1815–1864) gilt als Begründer der mathematischen Logik. Er war auch ziemlich clever, was Differentialgleichungen und Wahrscheinlichkeitsrechnung angeht.

```
>>> 4 < "3"
True
```

Nanu!?

Nichtnumerische Typen werden untereinander verglichen, indem die Namen ihrer Typen als Zeichenketten verglichen werden. Die Werte werden dafür überhaupt nicht angeschaut:

```
>>> True > "bla"
False
```

(True ist vom Typ `bool` und "bla" vom Typ `str`, der Vergleich ist also unter dem Strich eigentlich »"bool" > "str"«, und das ist – wie oben besprochen – False.)³



In Python 3 sind Vergleiche zwischen Objekten unterschiedlichen Typs einfach nicht erlaubt⁴:

```
>>> 4 < "3"
TypeError: unorderable types: int() < str()
```

Logische Operatoren Boolesche Werte können mit den **logischen Operatoren** `and`, `or` und `not` verknüpft werden. Der Operator `not` ist am einfachsten: Er liefert `False`, wenn sein Operand »wahr« ist, und sonst `True`. (Beachten Sie, dass Python nicht darauf besteht, dass der Operand ein *Boolescher* Wert im engeren Sinne ist.)

```
>>> not True
False
>>> not False
True
>>> not (1 > 0)
False
>>> not 4711
False
```

Operator `and` Der Operator `and` liefert genau dann ein »wahres« Ergebnis, wenn beide Operanden »wahr« sind:

```
>>> True and False
False
>>> True and True
True
```

Es ist wichtig zu wissen, dass `and` sich dafür die minimal mögliche Mühe macht: Zuerst wird der linke Operand ausgewertet, und wenn dieser ein »falsches« Ergebnis liefert, wird der rechte Operand völlig ignoriert, da sein Wert für das Gesamtergebnis nicht mehr relevant ist (es ist so oder so »falsch«). Dies ist oft nützlich, etwa in Konstruktionen wie

```
if n != 0 and total/n > 5:
    <<<<<<
```

³Man könnte sich fragen, was Guido van Rossum Potentes geraucht hatte, bevor er sich diese Merkwürdigkeit überlegt hat. Die Motivation dahinter, das zu erlauben, war, dass es möglich sein sollte, »heterogene Listen«, also Listen von Objekten unterschiedlichen Typs, zu sortieren. Später kamen komplexe Zahlen dazu, und heterogene Listen wurden sachte verpönt.

⁴Auch heterogene Listen können in Python 3 nicht mehr sortiert werden.

wo die Division nur stattfinden soll, wenn der Divisor von Null verschieden ist⁵. Logischerweise gilt das auch analog für Verkettungen der Form

```
a and b and c and d
```

– die Auswertung endet mit dem ersten Operanden, der »falsch« ist, bzw. wenn auch d »wahr« ist, mit dem Ende der Kette.

Der Operator `or` liefert ein »wahres« Ergebnis, wenn mindestens einer seiner Operanden »wahr« ist. Er funktioniert ganz genauso wie `and`, bis darauf, dass die Auswertung mit dem ersten Operanden endet, der sich als »wahr« herausstellt. Operator `or`



Tatsächlich liefert `or` nicht `True` oder `False` als Ergebnis, sondern den Wert des letzten ausgewerteten Operanden. Das ist sehr bequem, erlaubt es doch die Vergabe von »Standardwerten« für unbesetzte Variable:

```
>>> a = 0
>>> a or 123
123
```

(`and` macht dasselbe, aber da drängt die Anwendung sich nicht so auf.)

`not` hat Vorrang vor `and` und beide haben Vorrang vor `or`.

Hier ist noch eine kleine nützliche Ungewöhnlichkeit (oder ungewöhnliche Nützlichkeit): Python gestattet es, Vergleiche aneinanderzureihen: »`0 < a < 10`« ist äquivalent zu »`0 < a and a < 10`«, bis darauf, dass der Wert von `a` nur einmal bestimmt werden muss (damit sich das wirklich lohnt, könnte dort statt `a` ein komplizierter Funktionsaufruf stehen, der eine Datenbankrecherche macht oder etwas aus dem Internet herunterlädt). Auch hier wird links angefangen, weitergemacht, bis das Ergebnis feststeht, und allfällige irrelevante Ausdrücke am rechten Ende der Kette ignoriert. Vergleiche aneinanderreihen



Die Vergleiche müssen nicht notwendigerweise etwas miteinander zu tun haben: »`0 < a == b < c`« ist auch erlaubt (und äquivalent zu »`0 < a and a == b and b < c`«). Insbesondere erlaubt eine Vergleichskette wie »`a < b > c`« keine Aussagen darüber, ob »`a < c`« wahr ist oder nicht. Sie sollten das nicht überstrapazieren.

3.1.3 else und elif

Das `if`-Kommando ist nicht nur dazu zu gebrauchen, eine Kommandofolge auszuführen, wenn eine Bedingung erfüllt ist. Sie können auch alternativ eine Kommandofolge ausführen, wenn die Bedingung »wahr« ist, und eine andere, wenn sie »falsch« ist. Dazu dient `else`:

```
if n % 2 == 0:
    print("n ist gerade")
else:
    print("n ist ungerade")
```

(Wichtig: Auch hinter dem `else` steht ein Doppelpunkt.) In der `if...else`-Konstruktion wird die Kommandofolge nach dem `if` ausgeführt, wenn die Bedingung »wahr« ist, die Kommandofolge nach dem `else` sonst.



Die Regel, dass eine Kommandofolge direkt hinter dem Doppelpunkt stehen darf, gilt natürlich auch hier. Etwas wie

⁵Wenn Sie C oder eine andere Sprache aus demselben Dunstkreis kennen (Perl, Tcl, Shell, ...), dann ist dieses Verhalten Ihnen bereits geläufig. In anderen Programmiersprachen – Pascal oder FORTRAN drängen sich auf – bräuchte man dafür zwei ineinander verschachtelte IFs, weil ansonsten nicht garantiert ist, dass nicht möglicherweise der zweite Operand zuerst ausgewertet wird.

```
if n % 2 == 0: print ("n ist gerade")
else: print ("n ist ungerade")
```

ist absolut erlaubt, selbst wenn es nicht gerade den Gipfel des lesbaren Codes darstellt. Auch Mischformen wie

```
if n % 2 == 0:
    print ("n ist gerade")
else: print ("n ist ungerade")
```

sind denkbar, unter denselben Vorbehalten.

ifs verschachteln Natürlich können Sie ifs (und elses) auch verschachteln. Sie müssen nur die Einrückung konsequent einhalten. Zum Beispiel:

```
if x <= 0:
    print("x ist Null oder negativ")
else:
    if x < 1000:
        print("x ist kleiner als 1000")
    else:
        if x < 1000000:
            print("x ist kleiner als 1000000")
        else:
            print("x ist gigantomanisch riesengroß!")
```

Das funktioniert und ist – mit einem bisschen Wohlwollen – sogar halbwegs übersichtlich, aber wie Sie sehen, rutscht Ihr Code immer weiter nach rechts. Bildschirme sind heute ziemlich breit, aber 100% genial ist das nicht. Deswegen gibt es die Möglichkeit, das Ganze mit Hilfe von elif etwas kompakter hinzuschreiben:

```
if x <= 0:
    print("x ist Null oder negativ")
elif x < 1000:
    print("x ist kleiner als 1000")
elif x < 1000000:
    print("x ist kleiner als 1000000")
else:
    print("x ist gigantomanisch riesengroß!")
```

Hier werden die Bedingungen nach dem if bzw. den elifs nacheinander geprüft, bis sich eine als »wahr« herausstellt. Die Kommandofolge, die dieser Bedingung folgt, wird ausgeführt. Ist keine der Bedingungen »wahr«, dann wird die Kommandofolge hinter dem else ausgeführt.



Natürlich muss es nicht zwingend ein else geben. Gibt es keins, dann passiert in dem Fall, dass keine der if/elif-Bedingungen »wahr« war, eben einfach nichts.

In unserem Beispiel involvieren alle Bedingungen die Variable x. Auch das ist natürlich nicht vorgeschrieben; die Bedingungen können völlig beliebig sein und müssen überhaupt nichts miteinander zu tun haben. Allerdings ergibt es sich oftmals einfach so.

switch-Anweisung



Die CASE- oder switch-Anweisung, die viele Programmiersprachen für Fallunterscheidungen gemäß dem Wert einer Variablen unterstützen, werden Sie in Python vergeblich suchen. Sie können sie aber durch eine if-elif-Kette simulieren oder eine Auswahl von anderen Techniken verwenden, die auf Sprachelementen beruhen, die wir hier noch nicht besprochen haben.



Nur zur Illustration und ohne Rücksicht auf etwaige unaufgelöste Vorwärtsreferenzen: Schlagen Sie zum Beispiel einen gewünschten Wert in einem literalen Dictionary nach.

```
choice = 'Mortadella'
print({ 'Gelbwurst': 1.25,
        'Mortadella': 1.99,
        'Leberwurst': 0.99,
        'Speck': 1.10 }[choice])
```

Hier gibt es leider keine direkte Möglichkeit, einen Standardwert vorzugeben, aber das können Sie über die get()-Methode erreichen:

```
options = { 'Gelbwurst': 1.25,
            'Mortadella': 1.99,
            'Leberwurst': 0.99,
            'Speck': 1.10 }
choice = 'Mortadella'
print options.get(choice, 'Schlechte Auswahl'))
```

Oder Sie schalten einen in-Test vor:

```
if choice in branch:
    print(branch['choice'])
else:
    print('Schlechte Auswahl')
```

(Mehr über Dictionaries steht im Kapitel 6.)

3.2 Schleifen mit while

3.2.1 Wiederholungen

Nachdem Sie jetzt gelernt haben, wie Fallunterscheidungen mit if funktionieren, ist der nächste Schritt, Ihnen zu zeigen, wie Schleifen funktionieren. Die einfachste Form von Schleife, die in Python zur Verfügung steht, ist der Fallunterscheidung prinzipiell gar nicht unähnlich:

```
i = 1
while i <= 5:
    print i
    i = i + 1
```

Dieses Programm produziert die Ausgabe

```
1
2
3
4
5
```

Wie die if-Konstruktion besteht die while-Schleife aus einem Kopf – dem Schlüsselwort while, einer Bedingung und einem Doppelpunkt – und einer Kommandofolge.



Auch hier dürften Sie grundsätzlich

```
while i <= 5: print i; i = i + 1
```

schreiben. Aber tun Sie sich und uns einen Gefallen und lassen Sie's sein.

Der einzige Unterschied zwischen `if` und `while` besteht darin, dass bei `if` nach der Ausführung der Kommandofolge mit dem Rest des Programms weitergemacht wird. Bei `while` dagegen wird zum Anfang der Schleife zurückgesprungen und die Bedingung nochmals ausgewertet. Das wiederholt sich, bis die Auswertung der Bedingung »falsch« ergibt – und dann wird (wie bei `if`) die Kommandofolge übersprungen und mit dem Rest des Programms fortgefahren.



Wenn die Bedingung schon bei der allerersten Auswertung »falsch« ergibt, dann wird die Kommandofolge einfach übersprungen und mit dem Rest des Programms weitergemacht. Man spricht von einer »abweisenden« Schleife.



Viele Programmiersprachen haben »nichtabweisende« Schleifen, die auf jeden Fall einmal ausgeführt werden, bevor die Bedingung geprüft wird – »do {...} while (...);« in C (und Verwandten), »REPEAT ...UNTIL ...« in Pascal und so weiter. In Python möchte man unnötigen Ballast so weit wie möglich vermeiden und hat sich das gespart.

Lassen Sie sich übrigens nicht von dem Ausdruck

```
i = i + 1
```

verwirren. Mathematisch gesehen ist der natürlich Unsinn, aber wenn Sie sich erinnern, dass der Python-Operator `=` nicht dazu dient, eine mathematische Gleichung aufzustellen, sondern eine Abkürzung ist für »Bestimme den Wert des Ausdrucks auf der rechten Seite und schreibe ihn in die Variable auf der linken Seite«, dann ergibt es schon mehr Sinn. Unter dem Strich erhöht der Ausdruck den in der Variablen `i` gespeicherten Wert um 1.

Kombinierte Zuweisung



Dafür gibt es übrigens eine Kurzschreibweise:

```
i = i + 1
```

ist dasselbe wie

```
i += 1
```



Die Kurzschreibweise gibt es auch für die anderen Operatoren: `-=`, `*=`, `/=`, `//=`, `%=` und `**=`. Die Bedeutung ist jeweils sinngemäß dieselbe.



Diese »erweiterten Zuweisungen« funktionieren nicht nur für Zahlen, sondern zum Beispiel auch für Zeichenketten:

```
>>> s = "123"
>>> s += "X"
>>> s
'123X'
>>> s *= 3
>>> s
'123X123X123X'
```

Die einzige Vorbedingung ist, dass die zugrundeliegenden Operatoren wie `+` oder `*` für den betreffenden Datentyp definiert sind.



Ganz identisch sind » $x = x + y$ « und » $x += y$ « übrigen nicht. Zum einen könnte es sein, dass bei der Bestimmung von x irgendwelche Seiteneffekte ausgelöst werden. Bei $=$ passiert das dann zweimal und bei $+=$ nur einmal (das ist Absicht). Zum anderen ist es sehr wahrscheinlich, dass bei $=$ das Objekt, auf das x zeigt, verworfen und durch ein bei der Auswertung der rechten Seite neu angelegtes Objekt ersetzt wird, während bei $+=$, soweit irgend möglich, das Objekt, auf das x zeigt, direkt manipuliert wird. (Wenn wir genauer über Zuweisung geredet haben, wird das hoffentlich klarer.)



Zeichenketten sind in Python »unveränderbare« Objekte. Das heißt, auch bei » $s += "x"$ « muss ein neues Objekt generiert werden, das dann an s zugewiesen wird. Der Gipfel der Effizienz ist das natürlich nicht. (Wobei einige Implementierungen von Python es sich da erlauben, ein paar Abkürzungen zu nehmen, um den Performanceverlust zu mildern.)

Übungen



3.1 [!1] Schreiben Sie ein Programm, das von 10 aus rückwärts zählt und bei 0 aufhört.



3.2 [2] Schreiben Sie ein Programm, das die Zahlen 1 bis 50 ausgibt (eine pro Zeile), mit den folgenden Ausnahmen:

- Ist eine Zahl durch 5 teilbar, wird statt der Zahl das Wort »Hopp« ausgegeben.
- Ist eine Zahl durch 7 teilbar, wird statt der Zahl das Wort »Hipp« ausgegeben.
- Ist eine Zahl durch 5 *und* 7 teilbar, wird statt der Zahl die Phrase »Hipp Hopp« ausgegeben.



3.3 [3] Betrachten Sie eine beliebige natürliche Zahl n . Wenn n gerade ist, dividieren Sie n durch 2. Wenn n ungerade ist, berechnen Sie $3n + 1$. Wiederholen Sie dieses Vorgehen beliebig oft. Wahrscheinlich werden Sie früher oder später bei 1 ankommen. – Die *Collatz-Vermutung* (aufgestellt 1937 vom deutschen Mathematiker Lothar Collatz, 1910–1990) besagt, dass man ausgehend von irgendeiner natürlichen Zahl *immer* bei 1 ankommt. Schreiben Sie ein Programm, das für die Zahlen von 2 bis 100 die Collatz-Vermutung überprüft.⁶



3.4 [2] Ändern Sie Ihr Programm aus Übung 3.3 so, dass es für jede getestete Zahl die Anzahl der Schritte bestimmt, die nötig sind, um 1 zu erreichen. Was ist die größte Anzahl von Schritten, die für Zahlen bis 100 vorkommt, und bei welcher Zahl ist sie nötig? Wie sieht es aus für Zahlen bis 1000?

3.2.2 break und continue

Hin und wieder kommt es vor, dass eine Schleife vorzeitig abgebrochen werden muss (zum Beispiel weil eine Fehlersituation aufgetreten ist). In Python dient dazu das `break`-Kommando. Es beendet die innerste umgebende Schleife:

break-Kommando

```
s0 = 'Habe nun, ach! Philosophie, Juristerei und Medizin'
s1 = ''
i = 0
while i < len(s0):
    s1 += s0[i]           # i-tes Zeichen von s0
    if s0[i] == '!': break
```

⁶Aktuell (2015) ist nicht bekannt, ob die Collatz-Vermutung tatsächlich gilt. Überprüft wurde sie für alle Zahlen bis in die Gegend von $2 \cdot 10^{18}$ (Stand Februar 2015), aber das beweist natürlich überhaupt nichts.

```
i += 1
print s1
```

liefert die Ausgabe

```
Habe nun, ach!
```

weil das `if` in der Schleife die Schleife mit `break` beendet, wenn das aktuell betrachtete Zeichen ein Ausrufungszeichen ist.



Sie können mit `break` in Python (wie in C) immer nur die innerste Schleife beenden⁷. Wenn Sie ein `break` haben wollen, das auch aus umschließenden Schleifen herauspringen kann, müssen Sie die Bourne-Again Shell (Bash) benutzen oder Perl (dort heißt es `last`).

while und else



`while`-Schleifen in Python können einen `else`-Zweig haben (wir hatten ja gesagt, dass `if` und `while` sich nicht besonders voneinander unterscheiden). Wie bei `if` wird die Kommandofolge hinter `else` ausgeführt, wenn die kontrollierende Bedingung »falsch« liefert.



Sie werden sich möglicherweise – und mit einiger Berechtigung – fragen, was das soll. Schließlich ist

```
i = 1
while i <= 5:
    print i
    i += 1
else:
    print("Fertig!")
```

genau dasselbe wie

```
i = 1
while i <= 5:
    print i
    i += 1
print("Fertig!")
```

und das ist auch noch kürzer. Allerdings müssen Sie dazu wissen, dass das Kommando `break` *komplett* aus der Schleife herausspringt und dabei auch ein allfälliges `else` übergeht. Das heißt, das `else` einer `while`-Schleife wird nur dann ausgeführt, wenn die Schleife »ordentlich« beendet wurde, also über die Bedingung hinter dem `while` und nicht irgendein `break` innerhalb der Schleife. Betrachten Sie das Beispiel

```
start = 10
end = 15
k = start
while k <= end:
    if k % 7 == 0:
        print k, "ist durch 7 teilbar!"
        break
    k += 1
else:
    print "Keine Zahl zwischen", start, "und", end
    print "ist durch 7 teilbar!"
```

Das `else` erspart Ihnen an dieser Stelle eine weitere und vielleicht nicht immer auf den ersten Blick einleuchtende Fallunterscheidung.

`continue` Zusätzlich zum `break` gibt es auch noch das Kommando `continue`. Mit `continue`

⁷OK, OK, in C gibt es notfalls immer noch `goto` ...

können Sie aus der Mitte einer Schleife heraus wieder an deren Anfang springen und den nächsten Durchlauf beginnen – wobei bei `while` dafür zuallererst die Bedingung getestet wird.

Das Beispiel dafür benutzt ein paar Sachen, die Sie noch nicht kennen: Sie können Daten von der Standardeingabe lesen, indem Sie das Modul `sys` importieren: Standardeingabe lesen

```
>>> import sys
```

Anschließend liest die Methode `sys.stdin.readline()` die nächste Zeile von der Standardeingabe ein:

```
>>> s = sys.stdin.readline()
foobar
>>> s
'foobar\n'
```



»Methoden« sind ein Konzept aus der objektorientierten Programmierung, auf das wir später noch zurückkommen. Falls es Sie interessiert: `sys.stdin` ist ein Dateiojekt, das für die Standardeingabe steht, und die Methode `readline()` ist für Dateiobjekte erklärt. Mit dieser Methode bringen Sie das Dateiojekt dazu, die nächste Zeile zu holen und als Ergebnis zurückzuliefern.

Methoden



Die gelesene Zeile endet immer mit einem Zeilentrenner (`»\n«`)⁸. Nur am Dateiende ist das nicht der Fall, dann liefert die Funktion die leere Zeichenkette zurück.

Ob eine Zeichenkette nur aus Ziffern besteht, können Sie mit der Methode `isdigit()` prüfen. (Zeichenketten sind auch Objekte.)

`isdigit()`

```
>>> "123".isdigit()
True
>>> "abc".isdigit()
False
```

Der Zeilentrenner an gerade frisch gelesenen Zeilen ist dabei möglicherweise störend. Ihn können Sie mit der Methode `strip()` loswerden:

`strip()`

```
>>> "abc\n".strip()
'abc'
```



Genaugenommen entfernt `strip()` beliebigen Freiplatz – außer Zeilentrennern also auch Leerzeichen, Tabulatorzeichen, Wagenrücklaufzeichen, Formularvorschübe und so weiter – von *beiden* Enden einer Zeichenkette. Also:

```
>>> " \t \n xyz 123 \f\r ".strip()
'xyz 123'
```

(Freiplatz im Inneren einer Zeichenkette bleibt unberührt).

Alle diese Nützlichkeiten können wir jetzt verwenden, um eine Zahl von der Standardeingabe einzulesen und sicherzustellen, dass es sich um eine Zahl handelt:

```
number = None
while number is None:
    line = sys.stdin.readline()
```

⁸Jedenfalls allerallermeistens. Wenn die letzte Zeile in einer Datei keinen Zeilentrenner am Ende hat, dann hat sie auch keinen, wenn sie von Python gelesen wurde. Damit das passiert, müssen Sie sich aber schon ein bisschen anstrengen.

```

if line == '':
    print "Dateiende!"
    break
line = line.strip()
if not line.isdigit():
    print "Bitte geben Sie eine Zahl ein!"
    continue
number = int(line)
else:
    print number

```

Wir benutzen hier `None` als Startwert für die Variable `number`. `None` ist ein spezieller Wert, der für »kein Wert« steht.



Normalerweise könnte man statt »`while number is None`« auch nur »`while not number`« schreiben. Allerdings könnte es sein, dass der Benutzer die Zahl 0 eingibt – und wie wir in Abschnitt 3.1.2 gelernt haben, gilt der numerische Wert 0 als »falsch«, so dass »`while not number`« einen weiteren Schleifendurchlauf bewirken würde. »`while number is None`« prüft hingegen auf Identität mit dem Objekt `None` (das es nur einmal gibt).



Objektidentität

Der Vergleichsoperator `is` ist eine verschärfte Form des Operators `==` – während `==` auf Wertgleichheit testet (wobei Sie theoretisch sogar bestimmen können, wie das genau funktionieren soll), testet `is` auf Objektidentität. »`a is b`« ist genau dann »wahr«, wenn `a` und `b` für dasselbe Python-Objekt stehen. (Statt »`not a is b`« können – und sollten – Sie »`a is not b`« schreiben; `is not` verhält sich zu `is` wie `!=` zu `==`.)

Übungen



3.5 [!3] Schreiben Sie ein einfaches Zahlen-Ratespiel: Der Computer erzeugt eine Zufallszahl zwischen 1 und 100 und fordert den Benutzer auf, diese Zahl zu erraten. Dazu gibt der Benutzer eine vermutete Zahl ein, und der Computer antwortet mit »Zu groß« oder »Zu klein«, bevor er nach einer neuen Vermutung fragt. Das Spiel endet, wenn die Zufallszahl erraten wurde oder der Benutzer eine leere Eingabe macht. (*Tipp:* Nach dem Kommando »`import random`« liefert die Funktion `random.randint(a, b)` eine ganzzahlige Zufallszahl n mit $a \leq n \leq b$.)

Zusammenfassung

- Python nutzt Einrückung zur Darstellung der Programmstruktur.
- In Python gibt es einen Booleschen Datentyp, der aber bis auf die Werte `True` und `False` dem ganzzahligen Datentyp entspricht.
- Python unterstützt diverse Vergleichsoperatoren, die Boolesche Ergebnisse liefern.
- Die logischen Operatoren sind `not`, `and` und `or`.
- Python erlaubt die bedingte Ausführung von Programmteilen mit `if` und `else`.
- Ketten von Vergleichen lassen sich oft mit `elif` vereinfachen.
- Python unterstützt keine CASE- bzw. `switch`-Verzweigung.



4

Strukturierte Datentypen: Folgen

Inhalt

4.1	Strukturierte Daten: Tupel.	46
4.2	Strukturierte Daten: Listen	51
4.3	Mehr über Zeichenketten	56
4.4	Schleifen mit for und Ranges.	62
4.5	List Comprehensions	66

Lernziele

- Tupel und Listen einsetzen können
- Mehr über Zeichenketten erfahren
- Schleifen mit for verwenden können
- *List comprehensions* einsetzen können

Vorkenntnisse

- Einfache Datentypen in Python (Kapitel 2)
- Verzweigungen und Schleifen in Python (Kapitel 3)
- Erfahrung mit anderen Programmiersprachen ist von Vorteil

4.1 Strukturierte Daten: Tupel

Pythons einfache Datentypen sind bequem und nützlich, aber reichen nicht für alle Anwendungen aus. Wenn wir zum Beispiel ein Programm schreiben wollen, das sich mit Punkten im dreidimensionalen Raum befasst, dann ist es nützlich, die x -, y - und z -Koordinaten eines Punkts gemeinsam behandeln zu können. Schließlich ist eine Zuweisung wie

```
p1 = p0
```

wesentlich bequemer als

```
p1_x = p0_x; p1_y = p0_y; p1_z = p0_z
```

Tupel Python erlaubt es, mehrere Datenobjekte zu einem »Tupel« zusammenzufassen. Einen Punkt im Raum könnten Sie zum Beispiel darstellen wie

```
p0 = 1, 2, 3
```

x -, y - und z -Koordinate

Das heißt, bei Tupel-Literalen werden die einzelnen Komponenten (nennen wir sie »Elemente«) durch Kommas getrennt. Oft schreibt man Tupel in Klammern, um deutlicher zu machen, dass die Komponenten zusammengehören:

```
p0 = (1, 2, 3)
```

Tupel mit einem einzigen Element



Wenn Sie aufgepasst haben, werden Sie sich jetzt sicherlich fragen, ob es Tupel mit einem einzigen Element geben kann und wenn ja, wie Python solche Tupel von mathematischen Ausdrücken unterscheidet, die in Klammern stehen. Ist »(4711)« ein literales Tupel mit dem einzigen Element »4711«, oder ist es einfach die literale Ganzzahl 4711 in Klammern? Die Antwort darauf ist: Letzteres. Um ein Tupel mit einem einzigen Element hinzuschreiben, müssen Sie ein Komma hinter das Element setzen, so wie »(4711,)« (»4711,« würde es auch schon tun.)



Allgemein gesagt: Ein Komma hinter dem letzten Element macht nichts und wird ignoriert. Dies hilft nicht nur bei der Unterscheidung zwischen ein-elementigen Tupeln und mathematischen Ausdrücken, sondern erleichtert zum Beispiel auch das Schreiben von Programmen, die Python-Code generieren.

Leere Tupel



Als nächstes werden Sie vermutlich darüber nachgrübeln, ob es auch Tupel ganz *ohne* Elemente geben kann? Natürlich – das leere Tupel ist einfach ein Paar von Klammern ohne Inhalt:

```
empty = ()
```

(Hier dürfen die Klammern selbstverständlich nicht fehlen.)

tuple()



Statt über die runden Klammern können Sie Tupel auch mit der Funktion `tuple()` konstruieren (genauer gesagt ist `tuple()` der Aufruf des Konstruktors für die `tuple`-Klasse, aber objektorientierte Programmierung haben wir noch nicht besprochen). Für literale Tupel im Programmtext ist das vermutlich etwas überkandidelt, aber mit `tuple()` können Sie Objekte anderer Datentypen in Tupel umwandeln, was manchmal nützlich ist. Der Unterschied ist, dass `tuple()` einen einzigen Parameter erwartet, der zum Beispiel eine Liste oder Zeichenkette sein kann (einzelne Zahlen sind nicht zulässig).

Elemente eines Tupels

Die Elemente eines Tupels müssen nicht zwingend einfache Datenobjekte sein, sondern sind ziemlich beliebig. Es hindert Sie zum Beispiel niemand daran, ein Tupel als Element eines anderen Tupels zu benutzen:

```
square = ((0,0), (1,0), (1,1), (0,1))
```

Generell müssen nicht alle Elemente eines Tupels vom selben Typ sein:

```
tage = ((1, 1, "Neujahr"), (18, 4, "Velociraptor Awareness Day"),
        (1, 5, "Maifeiertag"), (25, 5, "Towel Day"))
```

(Hier sehen Sie auch, dass Sie innerhalb eines Tupels die Quelltextzeilen beliebig umbrechen dürfen.)

Sie können Tupel als Ganzes an andere Variable zuweisen (Duh) oder komponentenweise verteilen: Tupel-Zuweisung

```
>>> p = (1, 2, 3)
>>> (a, b, c) = p
>>> b
2
```



Die Klammern auf der linken Seite sind hier auch nicht nötig – Sachen wie

```
>>> a, b, c = p
>>> a, b, c = 1, 2, 3
```

würden auch funktionieren.

In jedem Fall muss die linke Seite der Zuweisung zur rechten passen. Das heißt konkret, auf der linken Seite muss entweder eine einzige Variable stehen (sie bekommt dann das komplette Tupel zugewiesen) oder aber genau so viele Variable, wie das Tupel auf der rechten Seite Elemente hat. Ansonsten gibt es Ärger:

```
>>> a, b = p
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
>>> a, b, c, d = p
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 3 values to unpack
```

Tupel benehmen sich in vielerlei Hinsicht wie Zeichenketten:

Operationen

```
>>> 3, + 4,                               Aneinanderhängen
(3, 4)
>>> 3 * (7,)                               Vervielfachung
(7, 7, 7)
>>> (7,) * 3                               Dito andersrum
(7, 7, 7)
>>> len((1, 2, 3))                         Länge ist Anzahl der Elemente
3
```



An dieser Stelle zeigt sich, dass es in der Regel besser ist, Tupel in Klammern einzufassen. Gerade in den letzten drei Beispielen passieren sonst merkwürdige und wundersame Dinge:

```
>>> 3 * 7,
(21,)
>>> 7, * 3
File "<stdin>", line 1
```

```
SyntaxError: can use starred expression only as assignment target
>>> len(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: len() takes exactly one argument (3 given)
```



Man könnte sich auf den Standpunkt stellen, dass man die Klammern nur weglassen darf, damit Sachen wie

```
a, b = 1, 2
x, y = y, x
```

schöner aussehen. Überstrapazieren sollten Sie das nicht, sondern im Zweifel lieber die Klammern setzen.

Element herauspicken Wenn Sie aus einem Tupel ein bestimmtes Element herauspicken möchten, können Sie das tun, indem Sie dessen Index in eckigen Klammern an den Namen des Tupels anhängen:

```
>>> t = ("Hund", "Katze", "Maus", "Baum")
>>> t[2]
Maus
```

(Das erste Element des Tupels hat den Index 0.) Negative Indizes zählen vom Ende des Tupels (das letzte Element hat den Index -1):

```
>>> t[-1], t[-3]
('Baum', 'Katze')
```

Wenn ein Element eines Tupels selber ein Tupel ist, können Sie auf eines von dessen Elementen zugreifen, indem Sie mehr eckige Klammern benutzen:

```
>>> tt = ("A", ("B", "C"))
>>> tt[1]
('B', 'C')
>>> tt[1][0]
'B'
```

Stücke aus einem Tupel Zusammenhängende Stücke aus einem Tupel bekommen Sie mit etwas wie

```
>>> t[1:3]
('Katze', 'Maus')
```

Der erste Index ist dabei der des ersten zu übernehmenden Elements, und der zweite Index ist der des ersten *nicht mehr* zu übernehmenden Elements. Ist der zweite Index größer als die Anzahl der Elemente im Tupel, dann macht das nichts:

```
>>> t[1:999]
('Katze', 'Maus', 'Baum')
```

Wenn Sie den linken Index weglassen, dann wird mit dem ersten Element des Tupels angefangen. Wenn Sie den rechten Index weglassen (oder der rechte Index größer ist als die Anzahl der Elemente, wie gesehen), dann wird mit dem letzten Element des Tupels aufgehört:

```
>>> t[:3]
('Hund', 'Katze', 'Maus')
>>> t[2:]
('Maus', 'Baum')
>>> t[:]
('Hund', 'Katze', 'Maus', 'Baum')
```


(Wie Sie sehen können, ist »[:]« eine geschickte Methode, um ein Tupel komplett zu kopieren.) Ein Konstrukt der Form »<linker Index>:<rechter Index>« nennen wir auch *slice* (»Scheibe«).

Slices



Das auf den ersten Blick etwas merkwürdig anmutende Verhalten mit dem zweiten Index, der der des ersten nicht mehr zu übernehmenden Elements ist statt der des letzten noch zu übernehmenden (was einem möglicherweise naheliegender vorkäme), hat den Vorteil, dass ein Ausdruck wie

```
t[:i] + t[i:]
```

das komplette Tupel liefert.

Die Indizes müssen »vernünftig« sein in dem Sinne, dass der zweite Index sich auf ein Element weiter rechts im Tupel beziehen muss als der erste. Ist das nicht der Fall, dann bekommen Sie ein leeres Tupel als Ergebnis:

```
>>> t[2:1]
()
>>> t[2:2]
()
>>> t[2:-1]
('Maus',)
-1 ist hier dasselbe wie 3
```

Mit Slices können Sie nicht nur zusammenhängende Teilfolgen von Elementen aus einem Tupel herausholen, sondern auch Elemente überspringen, indem Sie als dritten Bestandteil (hinter einem weiteren Doppelpunkt) noch eine Schrittweite angeben. Wenn Sie zum Beispiel das zweite, vierte, ... Element haben möchten, dann geht das mit

Schrittweite

```
>>> t[1::2]
('Katze', 'Baum')
```

Natürlich können Sie auch hier eine Obergrenze angeben:

```
>>> t1 = t + ("Stein", "Fisch", "Auto")
>>> t1[0::2]
('Hund', 'Maus', 'Stein', 'Auto')
>>> t1[0:4:2]
('Hund', 'Maus')
```

Negative Schrittweiten durchlaufen das Tupel von rechts nach links. Damit das klappt, muss der zweite Index sich auf ein Element »weiter links« im Tupel beziehen als der erste:


```
>>> t[::-1]
('Baum', 'Maus', 'Katze', 'Hund')
>>> t1[-2:0:-1]
('Fisch', 'Baum', 'Katze')
```


Einzelne Elemente eines Tupels können Sie zwar lesen (denken Sie an die eckigen Klammern), aber nicht schreiben:

Tupel: Unveränderbar

```
>>> t[0] = "Wolf"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Tupel sind, wie der Python-Jargon sagt, »unveränderbar« (engl. *immutable*).

 Auf den ersten Blick mag Ihnen das wie eine ärgerliche Einschränkung vorkommen, aber es ist nicht wirklich schlimm. Im nächsten Abschnitt lernen Sie Listen kennen, die alles unterstützen, was Sie mit Tupeln machen können, und nicht unveränderbar sind.

 Wobei sich natürlich als Nächstes die Frage aufdrängt: Wenn es Listen gibt, wofür braucht man dann noch Tupel? Die Antwort darauf lautet, dass einerseits gewisse Innereien von Python sich implizit auf Tupel abstützen und es darum kein Problem ist, diese Funktionalität, die man sowieso braucht, auch Benutzern zugänglich zu machen. Die zweite Antwort lautet, dass die Unveränderbarkeit Tupel an ein paar Stellen zum Einsatz kommen lässt, wo eine Liste nicht funktionieren würde. Tupel können zum Beispiel als Indizes von Dictionaries fungieren, und das geht mit Listen nicht.

Außer den grundlegenden Operationen »Aneinanderhängen« und »Vervielfachen« unterstützen Tupel noch einige andere nützliche Operationen. Zum Beispiel:

Test auf Enthaltensein Mit den »in«- und »not in«-Operatoren können Sie prüfen, ob ein bestimmtes Element in einem Tupel enthalten ist:

```
>>> "Katze" in t
True
>>> "Löwe" in t
False
>>> "Löwe" not in t
True
```


Methode `index()`

Wenn Sie sich dafür interessieren, *wo* im Tupel das gewünschte Element auftaucht, dann müssen Sie die Methode `index()` benutzen:


```
>>> t.index("Katze")
1
```

Diese Methode liefert Ihnen den Index (beginnend bei 0) des ersten Auftretens des Elements im Tupel. Kommt das gesuchte Element im Tupel nicht vor, löst Python eine `ValueError`-Exception aus:


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

 Wenn Sie einen zweiten Parameter angeben (muss eine Zahl sein), dann gibt dieser den Index im Tupel an, wo mit der Suche angefangen wird:

```
>>> t2 = t + t
>>> t2.index("Katze")
1
>>> t2.index("Katze", 2)
5
```

 Mit einem dritten Parameter können Sie den Index angeben, wo die Suche aufhören soll. Gemäß den Konventionen für Slices (siehe oben) wird das Element des Tupels, auf das der Index sich bezieht, gerade *nicht* mehr angeschaut:

```
>>> t2.index("Katze", 2, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: tuple.index(x): x not in tuple
```

 »t2.index("Katze", 2, 4)« ist in etwa dasselbe wie »t2[2:4].index("Katze")«. Die beiden signifikante Unterschiede sind einerseits, dass nichts kopiert werden muss (das Suchen in der Slice würde dazu führen, dass die betroffenen Elemente erst einmal kopiert werden), und andererseits dass das Resultat den Index relativ zum Anfang des Tupels und nicht relativ zum Anfang der Slice angibt.

Minimum und Maximum Die Funktionen `min()` und `max()` liefern das Minimum bzw. das Maximum der Werte in einem Tupel:

```
>>> u = (2, 5, -3, 1)
>>> min(u), max(u)
(-3, 5)
```

Das Ganze funktioniert auch mit Tupeln, dessen Elemente Zeichenketten sind:

```
>>> min(t), max(t)
('Baum', 'Maus')
```

Allerdings müssen die Elemente untereinander vergleichbar sein:

```
>>> min(t+u)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: int() < str()
```

Diese Operation geht schief, weil Python (3) Zeichenketten und Zahlen nicht miteinander vergleichen kann (siehe Abschnitt 3.1.2).

Zählen Die Methode `count()` zählt, wie oft ein bestimmtes Element in einem Tupel vorkommt:

```
>>> (1,2,1,0,1).count(1)
3
```

4.2 Strukturierte Daten: Listen


»Listen« in Python benehmen sich wie Tupel, mit dem Unterschied, dass Sie Listen verändern können. Eine Liste konstruieren Sie, indem Sie die gewünschten Elemente, durch Kommas getrennt, in eckige Klammern schreiben (denken Sie dran, bei Tupeln waren es runde).

```
>>> liste = [1, 1, 2, 3, 5, 8, 13]
```



Bei Listen gibt es keine Verwechslungsgefahr mit mathematischen Ausdrücken, aber Sie dürfen trotzdem hinter dem letzten Element noch ein Komma haben:

```
>>> liste = [1, 1, 2, 3, 5, 8, 13,]
```

`list()`  Statt der eckigen Klammern funktioniert auch die Funktion `list()` (genaugenommen wieder der Konstruktor für die Klasse `list`). Damit können Sie zum Beispiel eine Liste aus einem Tupel erzeugen:

```
>>> liste = list((1, 1, 2, 3, 5, 8, 13))
```

Der Parameter von `list()` ist ein Objekt eines Folgen-Datentyps, also außer einer Liste etwa ein Tupel oder eine Zeichenkette. Das Ergebnis ist eine Liste, deren Elemente denen des Parameters entsprechen:

```
>>> list(t)
['Hund', 'Katze', 'Maus', 'Baum']
>>> list('Hund')
['H', 'u', 'n', 'd']
```

Elemente einer Liste ändern Sie können einzelne Elemente einer Liste ändern, indem Sie auf der linken Seite der Zuweisung ein Listenelement adressieren:

```
>>> liste = list(t)
>>> liste[1] = "Löwe"
>>> liste
['Hund', 'Löwe', 'Maus', 'Baum']
```

Natürlich geht das auch mit verschachtelten Listen:

```
>>> ll = [0, [1, 2]]
>>> ll[1][0]
1
>>> ll[1][0] = 3
>>> ll
[0, [3, 2]]
```

Allerdings können Sie keine Elemente ändern, die es in der Liste nicht gibt:

```
>>> ll[2] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

Teilstücke ändern Sie müssen also alle eventuell nötigen »Plätze« vorher vorbelegen. Teilstücke einer Liste können Sie ändern, indem Sie auf die linke Seite der Zuweisung eine Slice schreiben:

```
>>> liste[1:2] = ["Katze", "Tiger"]
>>> liste
['Hund', 'Katze', 'Tiger', 'Maus', 'Baum']
```

Auf diese Weise können Sie auch ein oder mehrere Elemente in die Liste einfügen:

```
>>> liste[1:1] = ["Luchs", "Elch"]
>>> liste
['Hund', 'Luchs', 'Elch', 'Katze', 'Tiger', 'Maus', 'Baum']
```

(Die Slice »1:1« beschreibt den »Zwischenraum« unmittelbar vor dem Element mit dem Index 1.)



Selbstverständlich dürfen Sie auch Slices mit Schrittweiten verwenden. Das hat allerdings schon ein gewisses Verwirrpotential:

```
>>> liste[1:4:2] = ['Wolf', 'Schaf']
>>> liste
['Hund', 'Wolf', 'Elch', 'Schaf', 'Tiger', 'Maus', 'Baum']
```

Hierbei müssen Sie (wie immer) genau aufpassen, dass die Anzahl der zu ersetzenden Elemente, die von der Slice auf der linken Seite beschrieben werden, mit der Anzahl der neuen Elemente auf der rechten Seite übereinstimmt.

Elemente an eine Liste anhängen können Sie am bequemsten mit der Methode `append()`:

```
>>> liste.append("Wal")
>>> liste
['Hund', 'Wolf', 'Elch', 'Schaf', 'Tiger', 'Maus', 'Baum', 'Wal']
```

Wenn Sie mehrere Elemente auf einmal anhängen wollen, können Sie sich der Methode `extend()` bedienen:

```
>>> liste2 = ["Amsel", "Drossel"]
>>> liste2.extend(['Fink', 'Star'])
>>> liste2
['Amsel', 'Drossel', 'Fink', 'Star']
```

Dabei müssen Sie die anzuhängenden Elemente als Folge (Liste, Tupel oder ähnlicher Typ) übergeben.

Für Uneingeweihte sind Listen für die eine oder andere zunächst verblüffende Verhaltensweise gut. Zum Beispiel:

```
>>> l1 = [1, 2, 3]
>>> l2 = l1
>>> l2[1] = 4
>>> l1
[1, 4, 3]
```

Die Variable `l1` enthält nicht etwa »die Liste«, sondern nur eine Referenz (einen Verweis) auf das Listenobjekt. Durch die Zuweisung »`l2 = l1`« wird `l2` zu einem zweiten Namen für *dasselbe* Listenobjekt. Wenn Sie also `l2[1]` einen neuen Wert geben, dann ist derselbe Wert auch sichtbar, wenn Sie auf das Listenobjekt über den ursprünglichen Namen `l1` zugreifen.



Alle Namen in Python sind Referenzen auf Objekte. Bei vielen davon fällt es nur nicht so sehr auf.



Wenn Sie eine Liste wirklich kopieren und nicht nur einen zweiten Namen für ein existierendes Objekt vergeben wollen, dann verwenden Sie etwas wie

```
>>> l2 = l1[:]
```

Natürlich kann das – je nach Länge der Liste – mit einem erheblichen Aufwand an Speicherplatz und Zeit zum Kopieren einhergehen.



Alternativ könnten Sie auch »`l1.copy()`« sagen. Python-Listen unterstützen diese Operation aus Kompatibilitätsgründen zu anderen Typen, bei denen die Slice-Notation nicht funktioniert.

»seichte« Kopie



Mit »l1[:]« bekommen Sie eine »seichte« Kopie (engl. *shallow copy*) der Liste, das heißt, wenn die Liste wiederum Listen, Tupel oder andere Objekte als Elemente hat, dann werden diese Elemente nicht kopiert, sondern die neue Liste (die Kopie) enthält Referenzen auf die ursprünglichen Objekte:

```
>>> l1 = [1, [2, 3], 4]
>>> l2 = l1[:]
>>> l2[l2[0]] = 5
>>> l1
[1, [5, 3], 4]
```

»tiefe« Kopie



Wenn diese Objekte auch alle verdoppelt werden sollen, müssen Sie sich der Funktion `deepcopy()` aus dem Modul `copy` bedienen:

```
>>> from copy import deepcopy
>>> l1 = [1, [2, 3], 4]
>>> l2 = deepcopy(l1)
>>> l2[l2[0]] = 5
>>> l2
[1, [5, 3], 4]
>>> l1
[1, [2, 3], 4]
```

`deepcopy()` gibt sich große Mühe, keine Probleme zu verursachen, indem es mehr kopiert als es sollte (etwa bei rekursiven Datenstrukturen mit Schleifen). Die Details stehen in der Python-Bibliotheksdokumentation.

Löschoperationen

Einzelne Elemente oder Bereiche von Elementen aus einer Liste löschen können Sie mit dem Kommando `del`:

```
>>> liste = [1, 2, 3, 4, 5]
>>> del liste[3]
>>> liste
[1, 2, 3, 5]
>>> del liste[1:3]
>>> liste
[1, 5]
```



»`del liste[:]`« löscht alle Elemente der Liste – die Liste existiert dann noch, aber hat die Länge 0, und Sie können neue Elemente einfügen oder anhängen. Alternativ dazu funktioniert die Methode »`liste.clear()`« (wiederum aus Kompatibilitätsgründen zu anderen Python-Datentypen ohne Slice-Notation).

Listen können alles, was Tupel können, allerdings mit ein paar möglicherweise überraschenden Effekten. Betrachten Sie etwa die Vervielfachung:

```
>>> liste = [[]] * 4
>>> liste[1].append("bla")
>>> liste
[['bla'], ['bla'], ['bla'], ['bla']]
```

In diesem Beispiel ist »`[[]]`« eine Referenz auf ein leeres Listenobjekt, die bei der Zuweisung einfach »vervielfacht« wird. Alle vier Elemente von `liste` verweisen also auf dasselbe Objekt, und wenn Sie irgendeines dieser Elemente ändern, ändern Sie alle anderen mit.



Wenn Sie eine Liste mit vier *verschiedenen* leeren Elementen haben wollen, können Sie das zum Beispiel über eine List Comprehension erreichen (siehe Abschnitt 4.5). Etwas wie

```
>>> liste = [[] for i in range(4)]
```

sollte das Gewünschte tun.


Neben den Methoden, die Sie noch von Tupeln kennen (etwa `count()`, `max()` oder `index()`), unterstützen Python-Listen noch einige nützliche Methoden, die nur für veränderbare Objekte Sinn ergeben. Zum Beispiel: Methoden

Elemente holen und gleichzeitig entfernen Die Methode `pop()` liefert das letzte Element der Liste und entfernt es gleichzeitig. Das ist nützlich zur Implementierung von warteschlangenartigen Objekten:

```
>>> liste = [1, 2, 3]
>>> liste.pop()
3
>>> liste
[1, 2]
```

Die Methode hat ein optionales numerisches Argument, das den Index des zu entfernenden Elements angibt (Standardwert ist, wie wir gesehen haben, `-1`). Damit können Sie zum Beispiel das erste Element entfernen:

```
>>> liste.pop(0)
1
>>> liste
[2]
```

 Es ist effizienter, Elemente vom Ende der Liste zu entfernen als vom Anfang – wenn Sie ein Element vom Anfang entfernen, müssen alle anderen Elemente »aufrücken«, und das kann sich bei langen Listen schon bemerkbar machen.

Bestimmte Werte entfernen Wenn Sie ein Element mit einem bestimmten Wert aus einer Liste entfernen wollen, aber nicht genau wissen, an welchem Index das Element zu finden ist, können Sie die Methode `remove()` benutzen:


```
>>> liste = [1, 2, 3, 2]
>>> liste.remove(2)
>>> liste
[1, 3, 2]
```

Dabei wird immer nur das erste passende Element entfernt. Hat kein Element der Liste den gesuchten Wert, gibt es einen `ValueError`.

Einzelne Elemente einfügen Zum Einfügen eines einzelnen Elements an einer beliebigen (gezielten) Stelle in der Liste gibt es die Methode `insert()`. Der erste Parameter ist der gewünschte Index und der zweite das neue Element:

```
>>> liste = [1, 2, 3]
>>> liste.insert(2, 4)
>>> liste
[1, 2, 4, 3]
```

Insbesondere können Sie mit »`liste.insert(0, ...)`« ein Element an den Anfang der Liste setzen.

 »`liste.insert(i, x)`« ist im Wesentlichen dasselbe wie »`liste[i:i] = [x]`«. Ersteres ist möglicherweise etwas offensichtlicher und stellt sicher, dass `i` nur einmal ausgewertet wird.

Reihenfolge der Elemente umkehren Die Methode `reverse()` kehrt die Reihenfolge der Elemente einer Liste um – das erste Element wird zum letzten, das zweite zum Vorletzten und so weiter (siehe Matth. 19:30). Aus Effizienzgründen passiert das »am Platz«, das heißt, die Liste wird selber geändert, statt dass eine neue Liste mit denselben Elementen in der umgekehrten Reihenfolge konstruiert und als Ergebnis geliefert wird. (Die Methode hat kein Ergebnis.)

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



Wenn Sie wirklich die originale Liste unverändert lassen und eine neue Liste haben wollen, dann verwenden Sie die Funktion `reversed()` (beachten Sie das »d«):

```
>>> liste = [1, 2, 3]
>>> list(reversed(liste))
[3, 2, 1]
>>> liste
[1, 2, 3]
```

(Sie müssen das Ergebnis von `reversed()` wieder explizit zu einer Liste machen, wenn Sie das Ergebnis sehen wollen, denn `reversed()` liefert eigentlich ein Objekt vom Typ `list_reverseiterator`. Aber für irgendwas muss `list()` ja gut sein.)

Übungen



4.1 [!2] Angenommen, die Methode `append()` existierte nicht. Wie könnten Sie ein Element an eine Liste anhängen, ohne die komplette Liste kopieren zu müssen? (In anderen Worten, »`liste = liste + [element]`« gilt nicht.)



4.2 [!1] Wie würden Sie, beginnend mit dem zweiten Element, jedes zweite Element aus einer Liste `liste` löschen?



4.3 [1] Vergewissern Sie sich, dass die Konstruktion

```
>>> liste = [[] for i in range(4)]
```

zu einer Liste führt, deren Elemente voneinander unabhängig sind.

4.3 Mehr über Zeichenketten

Außer Tupeln und Listen gelten auch Zeichenketten als »Folgen«. Zeichenketten ähneln eher Tupeln als Listen, denn sie sind (wie Sie in Kapitel 2 gesehen haben) genau wie Tupel unveränderbar – etwas wie

```
>>> s = 'Hepp'
>>> s[1] = 'o'
```

funktioniert nicht. Alles, was wir in Abschnitt 4.1 über Tupel gesagt haben, gilt sinngemäß auch für Zeichenketten – die Verkettungs- und Vervielfachungsoperationen, Indizierung und Slices und sogar die Funktionen `min()` und `max()`, die Methode `index()` & Co.:



```
>>> s = 'Donaudampfschiffahrtsgesellschaft'
>>> s[5:16]
'dampfschiff'
>>> s[::2]
'Dnuapshffhtgslshf'
>>> min(s), max(s)
('D', 'u')
>>> s.index('f')
9
```


Außerdem haben Zeichenketten noch diverse andere nette Methoden, die für Tupel oder Folgen im Allgemeinen keinen Sinn ergeben. Hier ist eine Auswahl: Methoden für Zeichenketten

Groß- und Kleinschreibung Einige Methoden dienen dazu, Großbuchstaben in Kleinbuchstaben umzuwandeln und umgekehrt:

```
>>> s = "Habe nun, ach! Philosophie, Juristerei und Medizin"
>>> s.upper()
'HABE NUN, ACH! PHILOSOPHIE, JURISTEREI UND MEDIZIN'
>>> s.lower()
'habe nun, ach! philosophie, juristerei und medizin'
>>> s.swapcase()
'hABE NUN, ACH! pHILOSOPHIE, jURISTEREI UND mEDIZIN'
>>> s.title()
'Habe Nun, Ach! Philosophie, Juristerei Und Medizin'
>>> s.capitalize()
'Habe nun, ach! philosophie, juristerei und medizin'
>>> s
'Habe nun, ach! Philosophie, Juristerei und Medizin'
```

Die Methoden `upper()` und `lower()` liefern die Zeichenkette in respektive Groß- und Kleinbuchstaben. Mit `swapcase()` werden alle Großbuchstaben zu Kleinbuchstaben gemacht und umgekehrt. Die Methode `title()` macht den ersten Buchstaben jedes Worts zu einem Großbuchstaben und den Rest zu Kleinbuchstaben (was für englische Texte mehr Sinn ergibt als für deutsche). Die Methode `capitalize()` macht das erste Zeichen der Zeichenkette zu einem Großbuchstaben (falls das möglich ist) und den Rest zu Kleinbuchstaben. Die Zeichenkette selbst bleibt auf jeden Fall unverändert.

 Die Groß- und Kleinbuchstabenumwandlung folgt den Regeln des Unicode-Standards (Abschnitt 3.13, falls Sie es genau wissen müssen). Zugänglich dafür sind alle Zeichen, deren Kategorie eine der drei Möglichkeiten »Lu« (Großbuchstabe), »Ll« (Kleinbuchstabe) oder »Lt« (Titel-Buchstabe¹) ist.

 Für uns in Deutschland interessant ist der (Klein-)Buchstabe »ß«, für den es (noch) keinen allgemein üblichen Großbuchstaben gibt². Python wandelt »ß« in »SS« (zwei Zeichen) um; die Information, dass diese Zeichen ursprünglich mal ein »ß« waren, geht dabei verloren:

```
>>> "ß".upper()
'SS'
```

¹In Deutsch gibt es sowas nicht, aber zum Beispiel in Kroatisch – der Kleinbuchstabe »dž« wird als erster Buchstabe eines großgeschriebenen Worts als »Dž« geschrieben und innerhalb eines komplett in Großbuchstaben gesetzten Textes als »DŽ«. Im Unicode ist »Dž« ein Zeichen der Kategorie »Lt«, also ein »Titel-Buchstabe«.

²Seit Version 5.1 enthält der Unicode einen Codepunkt, namentlich U+1E9E, für einen »LATIN CAPITAL LETTER SHARP S«, also ein großes »ß«. Man ist sich allerdings noch nicht abschließend einig, wie dieses Zeichen aussehen soll.

```
>>> "ß".upper().lower()
'ss'
```

(Das ist übrigens durchaus konform mit den Unicode-Regeln.)



Die Methode `casefold()` dient dazu, Vergleiche ohne Rücksicht auf Groß- und Kleinschreibung zuzulassen. Sie funktioniert ähnlich wie `lower()`, ist aber aggressiver:

```
>>> "Masse".casefold()
'masse'
>>> "Maße".lower()
'maße'
>>> "Maße".casefold()
'masse'
```

Klassifizierung Oft ist es nützlich, herauszufinden, ob eine Zeichenkette nur aus bestimmten Zeichen (etwa Ziffern oder Kleinbuchstaben) besteht. Dafür gibt es eine Reihe von Methoden, deren Namen alle mit »is« anfangen und ein Boolesches Ergebnis (True oder False) liefern. Damit True herauskommt, müssen nicht nur alle Zeichen der gewünschten Kategorie angehören, sondern die Zeichenkette muss auch mindestens ein Zeichen lang sein:

isalpha() Alle Zeichen sind Buchstaben oder, formeller gesagt, haben eine der Unicode-Kategorien »Lm«, »Lt«, »Lu«, »Ll« oder »Lo«.

isidentifier() Die Zeichenkette ist ein gültiger Name im Sinne der Programmiersprache Python.



Dies schließt reservierte Wörter ein; wenn Sie herausfinden wollen, ob eine Zeichenkette ein reserviertes Wort in Python ist, also etwas wie `def` oder `import`, können Sie die Funktion `keyword.iskeyword()` benutzen.

islower() Alle Buchstaben in der Zeichenkette sind Kleinbuchstaben und es ist mindestens ein Kleinbuchstabe enthalten. (Die Zeichenkette darf auch Zeichen enthalten, die keine Buchstaben sind; diese beeinflussen das Ergebnis nicht.)

isupper() Alle Buchstaben in der Zeichenkette sind Großbuchstaben und es ist mindestens ein Großbuchstabe enthalten. (Die Zeichenkette darf auch Zeichen enthalten, die keine Buchstaben sind; diese beeinflussen das Ergebnis nicht.)

istitle() Die Groß- und Kleinbuchstaben in der Zeichenkette entsprechen dem, was die `title()`-Methode liefern würde, d. h. in jedem Wort ist der erste Buchstabe ein Großbuchstabe und der Rest Kleinbuchstaben.

isdigit() Alle Zeichen sind Ziffern. Das umfasst nicht nur die Ziffern im eigentlichen Sinne, die Sie über die Tasten in der obersten Reihe Ihrer Tastatur bekommen, sondern auch einige Sonderzeichen wie hochgestellte Ziffern und ähnliches.

isdecimal() Alle Zeichen können in Dezimalzahlen auftauchen. Neben den üblichen Ziffern zählen dazu zum Beispiel auch Ziffern in der arabisch-indischen Schreibweise (schauen Sie mal, wie Ihr Computer die Python-Zeichenkette »'\u00664'«, vulgo die »arabische« Vier, darstellt).



Wir bezeichnen unsere Ziffern 0123456789 gerne als »arabische« Ziffern, aber bis nach Arabien hat sich das anscheinend nicht herumgesprochen.

isnumeric() Alle Zeichen sind numerisch. Außer den Zeichen, für die `isdecimal()` True ergibt, gehören dazu auch Zeichen wie U+2155, besser bekannt als »VULGAR FRACTION ONE FIFTH« oder » $\frac{1}{5}$ «.

isalnum() Alle Zeichen sind alphanumerisch, das heißt, für jedes Zeichen liefert entweder `isalpha()`, `isdigit()` oder `isnumeric()` das Ergebnis `True`.

isprintable() Alle Zeichen sind »druckbar«, soll heißen keine Freiplatzzeichen außer dem Leerzeichen (U+0020). Diese Methode liefert auch `True`, wenn die Zeichenkette leer ist.

isspace() Alle Zeichen in der Zeichenkette sind Freiplatzzeichen.

Anfang, Ende und Enthaltensein Die Operatoren `in` und `not in` funktionieren auch für Zeichenketten. Tatsächlich können Sie sie nicht nur benutzen, um nach einzelnen Zeichen zu suchen, sondern nach Teilzeichenketten:

```
>>> "dampf" in "Donaudampfschiffahrtsgesellschaft"
True
```

Die Methode `index()` funktioniert wie bei Tupeln (inklusive dem `ValueError`, falls das Gesuchte nicht gefunden wird). Auch hier können Sie nach Teilzeichenketten suchen:

```
>>> s = "Donaudampfschiffahrtsgesellschaft"
>>> s.index("schiff")
10
>>> "abrakadabra".index("ra")
2
>>> "abrakadabra".index("ra", 3)
9
```

Wenn Sie Start- und Endpositionen angeben und nach Teilzeichenketten suchen, dann wird die Teilzeichenkette nur gefunden, wenn sie vollständig innerhalb des durch die Start- und Endpositionen gegebenen Teils der durchsuchten Zeichenkette liegt.

Wenn Sie keinen Wert auf `ValueError`-Exceptions legen, können Sie statt `index()` die Methode `find()` benutzen. Wenn diese Methode das Gesuchte nicht findet, liefert sie den Rückgabewert `-1` statt eines `ValueError`:

```
>>> "abrakadabra".index("xy")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> "abrakadabra".find("xy")
-1
```

Die Methoden `rindex()` und `rfind()` entsprechen `index()` und `find()`, aber suchen beginnend beim rechten Ende der (Teil-)Zeichenkette nach links, nicht wie diese beginnend beim linken Ende nach rechts:

```
>>> "abrakadabra".find("ab")
0
>>> "abrakadabra".rfind("ab")
7
>>> "abrakadabra".rfind("ab", 5)
7
>>> "abrakadabra".rfind("ab", 5, 7)
-1
```

(Beachten Sie, dass auch beim Suchen von rechts der linke Index kleiner als der rechte sein muss, wenn zwei angegeben wurden.)

Mit der Methode `startswith()` können Sie prüfen, ob eine Zeichenkette mit einer anderen Zeichenkette anfängt:

```
>>> "abrakadabra".startswith("abra")
True
>>> "abrakadabra".startswith("hokus")
False
```

Auch hier können Sie bestimmen, welcher Teil der Zeichenkette untersucht werden soll:

```
>>> "abrakadabra".startswith("abra", 5)
False
>>> "abrakadabra".startswith("abra", 7)
True
>>> "abrakadabra".startswith("abra", 7, 9)
False
```

Nützlicherweise können Sie sogar mehrere Möglichkeiten für den Anfang angeben (als Tupel):

```
>>> "abrakadabra".startswith(("abra", "kadabra"))
True
>>> "abrakadabra".startswith(("hokus", "pokus"))
False
```

Die Methode `endswith()` entspricht `startswith()`, testet aber das Ende der (Teil-)Zeichenkette.

Überflüssiges entfernen Die Methode `strip()` entfernt Freiplatz vom Anfang und Ende einer Zeichenkette. Freiplatz im Inneren der Zeichenkette bleibt unverändert:

```
>>> '  bli bla blubb  '.strip()
'bli bla blubb'
```

Wenn Sie eine Zeichenkette als Parameter angeben, werden alle Zeichen, die in dieser Zeichenkette vorkommen, entfernt (egal in welcher Reihenfolge sie auftreten):

```
>>> 'xyzxabczyzyxz'.strip('xyz')
'abc'
```

Die Methoden `lstrip()` und `rstrip()` betrachten jeweils nur den Anfang und nur das Ende der Zeichenkette:

```
>>> 'xyzxabczyzyxz'.lstrip('xyz')
'abczyzyxz'
>>> 'xyzxabczyzyxz'.rstrip('xyz')
'xyzxabc'
```

Aufteilen und Zusammenfügen Mit der Methode `split()` können Sie eine Zeichenkette an einem Trennzeichen (oder einer Trenn-Zeichenkette) in Stücke aufteilen, die die Methode als Liste zurückliefert:

```
>>> 'a,b,c,,d'.split(',')
['a', 'b', 'c', '', 'd']
```

(Wie Sie sehen, führen unmittelbar aufeinanderfolgende Trennzeichen zu leeren Zeichenketten im Ergebnis.) Mit einem numerischen zweiten Parameter können Sie angeben, wie viele Trennvorgänge maximal passieren sollen; die Anzahl der resultierenden Stücke ist dann also höchstens um eins größer als dieser Parameter:

```
>>> '1//2//3'.split('//', 1)
['1', '2//3']
```



Die Methode `rsplit()` benimmt sich wie `split()`, bis darauf, dass bei einer begrenzten Anzahl von Trennvorgängen die Trennvorgänge von rechts her vorgenommen werden statt von links.

Wenn Sie gar keinen Parameter übergeben, wird die Zeichenkette an Freiplatzzeichen aufgeteilt. Folgen von Freiplatzzeichen zählen als ein Trennzeichen (es gibt also keine zusätzlichen leeren Elemente in der Ausgabe) und Freiplatz am Anfang und am Ende der Zeichenkette wird ignoriert:

```
>>> ' x y z '.split()
['x', 'y', 'z']
```



Wenn Sie das Freiplatz-Trennverfahren nutzen und trotzdem eine Obergrenze für die Trennvorgänge angeben wollen, müssen Sie letztere mit ihrem Namen angeben:

```
>>> ' x y z '.split(maxsplit=1)
['x', 'y z']
```

(Mehr über benannte Parameter erfahren Sie in Kapitel 5.) Alternativ können Sie auch `None` als ersten Parameter übergeben.

Die gegenteilige Operation realisiert die Methode `join()`. Die Zeichenkette, auf die die Methode angewendet wird, dient dabei als Trenner für die Folge, die als Parameter angegeben wurde:

```
>>> "; ".join(('1', '2', '3'))
1; 2; 3
```

Die Folge darf dabei nur Zeichenketten enthalten.

Ebenfalls nützlich ist das Aufteilen einer Zeichenkette in zwei an einem Trennzeichen (oder einer Trenn-Zeichenkette). Dafür gibt es die Methode `partition()`:

```
>>> "foo.py".partition('.')
('foo', '.', 'py')
```

Wie Sie sehen, wird hier das Trennzeichen mit zurückgegeben. Kommt das Trennzeichen in der Zeichenkette nicht vor, dann liefert die Methode die ursprüngliche Zeichenkette gefolgt von zwei leeren Zeichenketten:

```
>>> "foo.py".partition('/')
('foo.py', '', '')
```

Tritt das Trennzeichen mehrmals auf, dann zählt das erste Auftreten:

```
>>> "www.example.com".partition('.')
('www', '.', 'example.com')
```



Auch von `partition()` gibt es eine Version, die am rechten Ende der Zeichenkette mit der Suche nach dem Trenner anfängt. Es wird Sie sicher nicht überraschen, zu hören, dass sie `rpartition()` heißt:

```
>>> "www.example.com".rpartition('.')
('www.example', '.', 'com')
```



Wenn `rpartition()` das Trennzeichen nicht findet, liefert die Methode ein Tripel aus zwei leeren und der ursprünglichen Zeichenkette:

```
>>> "www.example.com".rpartition('/')
('', '', 'www.example.com')
```

`format()` Eine wichtige weitere Methode für Zeichenketten ist `format()`. Diese Methode wird zur formatierten Ausgabe von Daten benutzt, etwa so:

```
>>> "Die Summe von {0} und {1} ist {2}.".format(2, 4, 6)
Die Summe von 2 und 4 ist 6.
```

Formatschlüssel Die Zeichenkette, auf die die Methode angewandt wird, kann Formatschlüssel der Form »{...}« enthalten. Im einfachsten Fall steht zwischen den geschweiften Klammern eine Zahl, die eines der Argumente der Methode bezeichnet; »{1}« wird zum Beispiel durch das zweite Argument ersetzt (wir zählen wie üblich ab Null).



Der Vorteil dieser Vorgehensweise ist, dass der Inhalt der Zeichenkette unabhängig von der Reihenfolge der Argumente beim Methodenaufruf ist. Das macht es zum Beispiel einfach, die Ausgabe eines Programms in eine andere Sprache zu übersetzen, deren Grammatik es nötig machen könnte, die Argumente in einer anderen Reihenfolge in die Zeichenkette einzubauen. Denn das kann passieren, ohne dass das Programm selbst geändert werden muss – ein Übersetzer benötigt also keine tiefgehenden Python-Kenntnisse. (Zum Übersetzen legt man typischerweise die Zeichenketten und ihre Übersetzungen in einer »Datenbank« ab, so dass das eigentliche Programm nicht geändert wird.)



Zwischen den geschweiften Klammern darf auch noch so einiges mehr stehen. Wir erklären das hier nicht komplett, sondern verweisen auf die Python-Dokumentation – und werden im weiteren Verlauf dieser Unterlage dieses oder jenes präsentieren.



Höchstens noch das Folgende: Wenn Sie einfach die Argumente des Methodenaufrufs in der angegebenen Reihenfolge nutzen wollen, müssen Sie keine Zahlen zwischen die geschweiften Klammern schreiben. Für das Beispiel oben funktioniert also auch

```
"Die Summe von {} und {} ist {}.".format(2, 4, 6)
```

4.4 Schleifen mit `for` und `Ranges`

In Python gibt es neben dem Kommando `while` für »vergleichsbasierte« Schleifen das Kommando `for`, mit dem Sie eine Folge Element für Element durchlaufen können, etwa wie

```
for farbe in ('rot', 'grün', 'gelb', 'blau'):
    print("Dasselbe in !".format(farbe))
```

Hierbei bekommt `farbe` nacheinander den Wert `rot`, `grün`, ... zugewiesen. Nach dem Durchlauf mit dem Wert `blau` endet die Schleife.



Wenn die Schleifenkontrollvariable (hier `farbe`) vorher schon einen Wert hatte, geht dieser verloren. Nach dem Ende der Schleife hat die Schleifenkontrollvariable den Wert aus dem letzten Durchlauf (hier `blau`).

Ansonsten entsprechen `for`-Schleifen ziemlich genau den `while`-Schleifen aus Kapitel 3: Sie können `break` und `continue` verwenden, um die komplette Schleife oder einen Schleifendurchlauf vorzeitig zu beenden, und mit `else` können Sie Kommandos angeben, die nach einem »normalen« Ende der Schleife ausgeführt werden:

```
for essen in ('Bratwurst', 'Spinat', 'Kartoffelbrei'):
    if essen == 'Spinat':
        print("Iih, {} ist eklig!".format(essen))
        break
    print("Mmh, {}".format(essen))
else:
    print("Alles lecker!")
```

Die Zuweisung an die Schleifenkontrollvariable einer `for`-Schleife folgt den üblichen Regeln für Python-Zuweisungen. (Stellen Sie sich vor, zwischen `for` und `in` steht die linke Seite einer Python-Zuweisung, und der Teil zwischen `in` und dem Doppelpunkt ist eine Folge von dazu passenden rechten Seiten.) Das heißt, Sie können etwas machen wie

Zuweisung an die Schleifenkontrollvariable

```
>>> for t in ((0,1), (1,2), (2,0)):
...     print(t)
...     ↩
(0, 1)
(1, 2)
(2, 0)
```

aber auch

```
>>> for u, v in ((0,1), (1,2), (2,0)):
...     print("u={}, v{}".format(u, v))
...     ↩
u=0, v=1
u=1, v=2
u=2, v=0
```

Das macht es einfach, mehrere Folgen gleichzeitig abzuarbeiten. Dabei hilft die Python-Funktion `zip()`:

```
>>> zip("abc", (1, 2, 3))
<zip object at 0x7f86dd5241c8>
>>> list(zip("abc", (1, 2, 3)))
[('a', 1), ('b', 2), ('c', 3)]
```

Das heißt, `zip()` übernimmt eine Anzahl von Folgen und liefert eine Folge von Tupeln, die jeweils das erste, zweite, ... Element *aller* dieser Folgen enthalten. Mit etwas wie

```
for b, z in zip("abc", (1, 2, 3)):
    <<<<<
```

können Sie also bequem über beide Folgen iterieren.



`zip()` setzt voraus, dass alle beteiligten Folgen dieselbe Länge haben; ist eine der Folgen abgearbeitet, produziert `zip()` keine weiteren Tupel mehr, egal ob in anderen Folgen noch weitere Elemente zur Verfügung stehen. Wenn das ein Problem darstellt, müssen Sie die Funktion `zip_longest()` aus dem Modul `itertools` benutzen.

Oft möchte man for-Schleifen über einen bestimmten Bereich von Ganzzahlen laufen lassen. Um das zu vereinfachen, bietet Python einen speziellen Folgentyp, die »Range« (neudeutsch für »Bereich«). Eine Range ist eine unveränderbare Folge von Zahlen mit einem Anfangswert, einem Endwert und einer Schrittweite.



Wenn Sie sich hier vage an die Slice-Operation zum Zugriff auf Teilbereiche von Tupeln, Listen und Zeichenketten erinnern fühlen, dann liegen Sie genau richtig.

Range-Objekte können Sie mit der Funktion `range()` erzeugen. Wenn Sie nur einen Parameter angeben, ist das der Endwert, der Anfangswert ist 0 und die Schrittweite ist 1. Wie üblich ist der Endwert selbst gerade nicht mehr Bestandteil der Range:

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

Wenn Sie eine Range haben möchten, die bei 1 anfängt und den »Endwert« tatsächlich enthält, müssen Sie die Argumente entsprechend anpassen:

```
>>> list(range(1, 6))
[1, 2, 3, 4, 5]
```

Auch Schrittweiten funktionieren wie erwartet:

```
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -5, -1))
[0, -1, -2, -3, -4]
```

Unsinnige Argumente führen zu leeren Listen als Ausgabe:

```
>>> list(range(0))
[]
>>> list(range(0, -10))
[]
```



Welchen Vorteil haben Ranges gegenüber Listen oder Tupeln? Ganz einfach: Range-Objekte speichern nur den Anfangswert, den Endwert und die Schrittweite (plus ein paar Verwaltungsinformationen), nicht etwa jede Zahl des Bereichs. Das heißt, »`range(10)`« und »`range(10000000)`« brauchen gleich viel (oder sollten wir »gleich wenig« sagen?) Speicher.



Wenn wir – wie hier aus optischen Gründen gemacht – Ranges in Listen umwandeln, geht dieser Vorteil natürlich flöten.

Ranges unterstützen die gängigen Operationen für Folgen, mit Ausnahme von Aneinanderhängen und Vervielfachen (diese würden die Annahme verletzen, dass eine Range immer eine ansteigende oder abfallende Folge von Zahlen darstellt). Dafür können Sie aber indizieren, Slices bilden und so weiter:

```
>>> r = range(1, 10)
>>> r[2]
3
>>> r[3:5]
range(4, 6)
>>> r[2::3]
```



```
range(3, 10, 3)
>>> 7 in r
True
>>> 33 in r
False
```

Sie können Ranges auch auf Gleichheit oder Ungleichheit vergleichen. Entscheidend dafür sind die produzierten Werte (nicht die Objektidentität):

```
>>> range(2) == range(3)
False
>>> range(0, 3, 2) == range(0, 4, 2)
True
```

Wenn Sie sich nicht mehr sicher sind, wie ein Range-Objekt konfiguriert ist, können Sie auch das nachträglich abfragen:


```
>>> r.start, r.stop, r.step
(1, 10, 1)
```


Ranges sind immer dann nützlich, wenn Sie eine Schleife mit numerischen Indizes brauchen – etwa um die Elemente einer Folge zu betrachten und dabei zu wissen, wo in der Folge Sie sich gerade befinden:


```
#!/usr/bin/python3
s = "Donaudampfschiffahrtsgesellschaft"
vokale = "aeiou"
vvk = 0
for i in range(len(s)-1):
    if s[i] in vokale and s[i+1] not in vokale:
        vvk += 1
print("{} Vokale standen vor einem Konsonant".format(vvk))
```

In diesem Beispiel müssen wir einen Blick auf das nächste Zeichen werfen können, und das können wir nicht, wenn wir über die einzelnen Zeichen laufen. Eine Schleife über die Indizes der Zeichen in der Zeichenkette macht das aber problemlos möglich, und wir müssen nur darauf achten, rechtzeitig aufzuhören.

Übungen

 **4.4** [!2] (Ein Klassiker.) Schreiben Sie ein Programm, das die Zahlen von 1 bis 50 ausgibt, mit den folgenden Ausnahmen: Ist eine Zahl ohne Rest durch 3 teilbar, soll statt dessen der Text HIPP ausgegeben werden. Ist eine Zahl ohne Rest durch 5 teilbar, soll statt dessen der Text HOPP ausgegeben werden. Ist eine Zahl sowohl ohne Rest durch 3 als auch ohne Rest durch 5 teilbar, soll statt dessen der Text HIPP HOPP ausgegeben werden.

 **4.5** [2] Schreiben Sie ein Programm, das eine Multiplikationstabelle für das »kleine Einmaleins« ausgibt. (*Tipps:* Verwenden Sie zwei verschachtelte for-Schleifen. Wenn Sie die Methode format() benutzen, können Sie einen Schlüssel der Form »{:4d}« verwenden, um einen numerischen Wert als Ganzzahl rechtsbündig in einem 4 Zeichen breiten Feld auszugeben. Mit »print(..., end='')« können Sie etwas ausgeben, ohne dass danach eine neue Zeile angefangen wird.)

 **4.6** [2] Versuchen Sie, das »Vokal vor Konsonant«-Programm so umzuschreiben, dass es direkt über die Buchstaben der Zeichenkette iteriert statt über deren Indizes. (Explizites Indizieren in einer Extra-Variablen ist geschummelt.)

4.5 List Comprehensions

Oft möchten Sie die Elemente einer Folge durchlaufen und eine neue Folge konstruieren, die für jedes Element der Ursprungsfolge ein Element enthält, das sich aus diesem ergibt. Typischerweise landen Sie dann bei etwas wie

```
doppelt = []
for i in range(10):
    doppelt.append(2*i)
```

Das ist natürlich relativ umständlich und unelegant. Deswegen bietet Python eine alternative Schreibweise für solche Schleifen an, nämlich

```
[2*i for i in range(10)]
```

Wichtig sind dabei die eckigen Klammern. Eine solche Struktur heißt *list comprehension*, und sie kann überall dort auftreten, wo Python eine Folge erlaubt.

Natürlich kann das `for in` in einer *list comprehension* über alle möglichen Arten von Folgen iterieren, nicht nur Ranges:

```
>>> print("\n".join(["Gib mir ein {}".format(c) for c in "HALLO"]))
Gib mir ein H!
Gib mir ein A!
Gib mir ein L!
Gib mir ein L!
Gib mir ein O!
```

Verschachtelung Sie dürfen auch mehr als ein `for` benutzen. Allerdings ist die Verschachtelung ein bisschen konterintuitiv:

```
>>> [i+j for i in ('a', 'b', 'c') for j in ('1', '2', '3')]
['a1', 'a2', 'a3', 'b1', 'b2', 'b3', 'c1', 'c2', 'c3']
```

Auf den ersten Blick würde man vielleicht annehmen, dass die »for j«-Schleife die äußere und die »for i«-Schleife die innere sein müsste. In Wirklichkeit ist es, wie Sie am Beispiel sehen können, genau umgekehrt. Stellen Sie sich einfach vor, dass die fors von links nach rechts verschachtelt sind wie in


```
for i in ('a', 'b', 'c'):
    for j in ('1', '2', '3'):
        <<<<<
```

list comprehensions mit `if` Der tatsächliche Grund, warum Python sich so benimmt, wie es sich benimmt, besteht darin, dass Sie hinter dem ersten `for` statt eines `for` auch ein `if` mit einer Bedingung haben können (nur das erste `for` *muss* ein `for` sein, damit eine Schleife zustandekommt):

```
>>> [i for i in range(10) if i % 2 == 0]
[0, 2, 4, 6, 8]
```

Hier hätte es natürlich keinen Sinn, wenn das `if` außen wäre und das `for` innen, deshalb werden diese Klauseln von links nach rechts interpretiert.

Übungen

 **4.7** [!] Erzeugen Sie eine Liste der ganzen Zahlen von 0 bis 20, die *nicht* ohne Rest durch 3 teilbar sind.



4.8 [!2] Erzeugen Sie eine Multiplikationstabelle wie in Übung 4.5, aber verwenden Sie für jede Zeile der Ausgabe eine *list comprehension*. Können Sie auch die komplette Tabelle mit einer einzigen *list comprehension* (mit zwei *for*-Klauseln) erzeugen?



4.9 [2] Angenommen, Sie haben zwei n -dimensionale Vektoren x und y (dargestellt etwa als Python-Tupel). Wie würden Sie mit einer *list comprehension* die Summe von x und y berechnen? Erinnern Sie sich: Die Summe von zwei Vektoren ist ein neuer Vektor, dessen Komponenten die paarweise Summe der Komponenten der Summanden sind. Mit anderen Worten,

$$z = (x_0 + y_0, x_1 + y_1, \dots, x_{n-1} + y_{n-1}).$$

Zusammenfassung

- Tupel sind unveränderbare Container, die beliebige Elemente enthalten dürfen. Sie erlauben unter anderem indizierten Elementzugriff, Slices, Längenbestimmung und Tests auf Enthaltensein bestimmter Elemente.
- Listen sind ähnlich wie Tupel, erlauben aber Modifikationen.
- Zeichenketten sind Folgen (wie Tupel und Listen). Sie unterstützen viele nützliche Methoden zur Klassifizierung, zum Aufteilen und Zusammenfügen und zum Suchen.
- Das *for*-Kommando erlaubt die Iteration über die Elemente einer Folge.
- Ranges sind spezielle Objekte, die eine Folge von Zahlen von einer Unter- bis zu einer Obergrenze (ggf. mit einer Schrittweite) liefern.
- *List comprehensions* sind eine bequeme und effiziente Methode, Listen ohne eine explizite Schleife zu konstruieren.



5

Funktionen

Inhalt

5.1	Einfache Funktionen.	70
5.2	Parameterübergabe	71
5.3	Variable Signaturen	75
5.4	Sichtbarkeitsbereiche	76

Lernziele

- Einfache Funktionen definieren können
- Mechanismen zur Parameterübergabe kennen und nutzen können
- Sichtbarkeitsbereiche verstehen

Vorkenntnisse

- Einfache (Kapitel 2) und simple strukturierte (Kapitel 4) Datentypen in Python
- Verzweigungen und Schleifen in Python (Kapitel 3)
- Erfahrung mit anderen Programmiersprachen ist von Vorteil

5.1 Einfache Funktionen

Die allermeisten Programmiersprachen erlauben es, bestimmte Befehlsfolgen in »vorgekochter« Form unter einem Namen abzulegen und dann bei Bedarf über diesen Namen aufzurufen. Wir sprechen von »Funktionen«, »Prozeduren«, »Sub-routinen« oder »Unterprogrammen«. Das hat diverse praktische Vorteile:


- Die unnötige Verdopplung von Programmtext wird vermieden.
- Funktionalität kann gekapselt und als *black box* realisiert werden, die auf der Basis von Eingabeparametern ein bestimmtes Ergebnis liefert (und möglicherweise bestimmte Seiteneffekte). Das macht es einfacher, Programme zu strukturieren und zu testen.
- Sie können »Bibliotheken« von häufig gebrauchten Funktionen aufstellen und diese in verschiedenen Programmen verwenden. Das spart Zeit und reduziert die Wahrscheinlichkeit von Fehlern, da Sie sich auf bereits erprobten Code verlassen können, statt das Rad immer wieder neu erfinden zu müssen.

Kommando `def` In Python können Sie eine Funktion einfach mit dem Kommando `def` definieren:

```
>>> def hallo():
...     print("Hallo Welt!")
...     ↵
>>> hallo()
Hallo Welt!
```

(Um eine Funktion aufzurufen, verwenden Sie den Funktionsnamen mit einem nachgestellten Paar runde Klammern.)

`def` ist ein »zusammengesetztes Kommando« wie `if` oder `for`, das heißt, nach dem Doppelpunkt kommt eine Kommandofolge entweder noch auf derselben Zeile (selten gesehen!) oder eingerückt in den Folgezeilen. Dort dürfen Sie wie üblich alle anderen Python-Kommandos (`for`, `if` & Co.) benutzen.

Zuweisung  Genaugenommen ist `def` eine glorifizierte Zuweisung: Der Python-Interpreter konstruiert aus der Kommandofolge ein Funktions-Objekt und hinterlegt im Funktionsnamen (hier `hallo`) eine Referenz auf dieses Objekt. Das können Sie sich anschauen:

```
>>> hallo
<function hallo at 0x7f45ff057d08>
```

Oder Sie können wie üblich mit dem Namen weiterarbeiten:

```
>>> huhu = hallo
>>> huhu()
Hallo Welt!
```

Dem Funktions-Objekt ist es gleichgültig, ob es über den Namen `hallo` oder den Namen `huhu` aufgerufen wird – beide sind Referenzen auf dasselbe Objekt.

Im Gegensatz zu anderen Programmiersprachen¹ unterscheidet Python nicht zwischen »Funktionen« (die einen Rückgabewert liefern) und »Prozeduren« (die keinen Rückgabewert liefern) – Python-Funktionen tun das entweder oder sie tun es nicht. Wie das genau geht, sehen wir im nächsten Abschnitt.



Es ist eine wichtige Beobachtung, dass `def` in Python ein ausführbares Kommando ist. Das heißt, der Kontrollfluss muss ein `def` erreichen, damit die betreffende Funktion tatsächlich definiert werden kann². Aus diesem Grund sollten Funktionen tendenziell eher am Anfang von Programmdateien stehen und das Hauptprogramm (wenn es eins gibt) am Ende. Steht das Hauptprogramm vorne und werden Funktionen aufgerufen, deren `def` noch nicht ausgeführt wurde, kommt es zu einem Fehler.

`def`: Ausführbares Kommando



Aus dieser Beobachtung folgt auch, dass Konstruktionen wie

```
if debug:
    def f(...):
        # eine Definition von f
else:
    def f(...):
        # eine andere Definition von f
```

erlaubt sind und funktionieren.

5.2 Parameterübergabe

Funktionen sind nützlich, aber zu großer Form laufen sie erst auf, wenn man dafür sorgen kann, dass sie sich von Aufruf zu Aufruf unterschiedlich verhalten. Prinzipiell haben Sie in Funktionen Zugriff auf »globale« Variable, aber wirklich elegant ist das nicht:

»globale« Variable

```
>>> s = "Welt"
>>> def hallo():
...     print("Hallo " + s + "!")
...     ↵
>>> hallo()
Hallo Welt!
>>> s = "Seemann"
>>> hallo()
Hallo Seemann!
```

Um diese Sorte Anwendungen bequemer zu machen, unterstützen Python-Funktionen die Übergabe von Parametern:

Parameter

```
>>> def hallo(s):
...     print("Hallo " + s + "!")
...     ↵
>>> hallo("Welt")
Hallo Welt!
>>> hallo("Seemann")
Hallo Seemann!
```

Umgekehrt können Sie aus einer Funktion einen Rückgabewert herausreichen. Dazu dient das Python-Kommando `return`:

Rückgabewert

```
>>> def addiere(u, v):
...     return u+v
...     ↵
```

¹Dinosaurier wie Pascal oder FORTRAN könnten einem in den Sinn kommen.

²Programmiersprachen wie C, die einen richtigen Compiler haben, haben dieses Problem nicht. Auch in Perl ist es zum Beispiel erlaubt, Subroutinen irgendwo in der Programmdatei zu definieren; da Perl als Erstes die komplette Programmdatei nach Definitionen absucht, ist das keine Einschränkung.

```
>>> addiere(2, 3)
5
```

Beim Funktionsaufruf bekommen die »formalen Parameter« der Funktion (in unserem Beispiel `u` und `v`) die Werte der »tatsächlichen Parameter« zugewiesen. Dies passiert analog zu einer »echten« Python-Zuweisung à la

```
>>> u, v = 2, 3
```

Parameter: wie lokale Variable Die formalen Parameter `u` und `v` verhalten sich wie Variable, die nur innerhalb der Funktion sichtbar sind. Etwa außerhalb der Funktion vorhandene Variable, die genauso heißen, werden davon nicht tangiert:

```
>>> u, v = 1, 2
>>> addiere(5, 6)
11
>>> u, v
(1, 2)
```

Niemand hindert Sie daran, die formalen Parameter innerhalb der Funktion zu manipulieren (es sind ja nur »lokale« Variable):

```
>>> def plus1(k):
...     k += 1
...     return k
...     ↩
>>> n = 5
>>> plus1(n)
6
>>> n
5
```

Wie Sie sehen: Wenn Sie eine numerische Variable als Parameter übergeben und den Parameter innerhalb der Funktion ändern, bekommt die Variable außerhalb der Funktion davon nichts mit. Anders ist das bei Listen:

```
>>> def f(liste):
...     liste[1] += 1
...     ↩
>>> l0 = [1, 2, 3]
>>> f(l0)
>>> l0
[1, 3, 3]
```

Überraschen sollte Sie das nicht besonders, weil Sie ja wissen, dass der Name einer Liste nur eine Referenz auf die Liste selbst ist. Das heißt, `l0` außerhalb und `liste` innerhalb der Funktion sind zwei unabhängige Referenzen auf dasselbe Listen-Objekt, und Sie können einzelne Elemente desselben Listen-Objekts über diese Referenzen (egal welche der zwei) ändern.



Das gilt natürlich nicht für die Liste selbst: Bei etwas wie

```
>>> def g(liste):
...     liste = ['a', 'b']
...     print(liste)
...     ↩
>>> l1 = [1, 2, 3]
>>> g(l1)
```



```
['a', 'b']           Ausgabe des print() in der Funktion
>>> l1
[1, 2, 3]
```

wird liste in der Funktion zu einer Referenz auf die neue Liste »['a', 'b']« gemacht, und das ist außerhalb der Funktion dann nicht mehr zu sehen.

Unter dem Strich können wir sagen, dass »einfache« Datentypen (Zahlen, Zeichenketten) als »Wert« übergeben werden und komplexe Datentypen wie Listen als »Referenz«. Wenn Sie nicht möchten, dass die Elemente einer Liste, die Sie als Parameter übergeben, in der Funktion geändert werden können, können Sie die Liste zum Beispiel bei der Übergabe kopieren:

```
>>> g(l1[:])
```

Funktionen können sich selbst aufrufen (der Fachausdruck ist »Rekursion«). Hier ist das klassische Beispiel: Rekursion

```
>>> def faktät(n):
...     if n == 1: return 1
...     return n * faktät(n-1)
...
>>> faktät(20)
2432902008176640000
```

Wenn Sie rekursive Funktionen definieren, ist es wichtig, dafür zu sorgen, dass die Rekursion irgendwann endet. In unserem Beispiel regelt das das `if` in der ersten Zeile der Funktion zusammen mit dem »n-1« in der zweiten Zeile.



Als Sicherheitsvorkehrung gegen wildgewordene rekursive Funktionen unterstützt der Python-Interpreter ein »Rekursions-Limit«. Werden mehr rekursive Aufrufe von Funktionen getätigt als das Rekursions-Limit festlegt, gibt es einen Laufzeitfehler (in Form einer `RuntimeError`-Exception).

Rekursions-Limit



Standardmäßig ist das Rekursions-Limit 1000. Sie können den aktuellen Wert mit

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

abfragen und mit etwas wie

```
>>> sys.setrecursionlimit(2000)
```

setzen. Je größer das Rekursions-Limit, um so tiefere Verschachtelung ist erlaubt, und um so mehr Speicher benötigt der Python-Interpreter auf dem (C-)Laufzeitstapel.

Bei einer Funktion mit mehreren Parametern ist es egal, in welcher Reihenfolge Sie die Parameter angeben, solange Python weiß, welcher Parameter welcher ist. Das ergibt sich entweder aus der Reihenfolge oder daraus, dass Sie die Namen der Parameter beim Aufruf ausdrücklich erwähnen: Reihenfolge

```
>>> def f(a, b, c):
...     print("a= b= c=".format(a, b, c))
...
>>> f(1, 2, 3)
a=1 b=2 c=3
```

```
>>> f(1, c=3, b=2)
a=1 b=2 c=3
>>> f(b=2, c=3, a=1)
a=1 b=2 c=3
>>> f(b=2, 3, a=1)
File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
>>> f(1, b=2, a=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() got multiple values for argument 'a'
```

Sie können unbenannte und benannte Parameter mischen, aber Sie müssen mit den unbenannten anfangen – diese werden dann den entsprechenden Parametern in der formalen Parameterliste der Funktion zugeordnet. Hinter einem benannten Parameter darf kein unbenannter mehr stehen. Außerdem ist es nicht erlaubt, mehr als einmal einen Wert für denselben Parameter anzugeben. Und übrigens muss für jeden Parameter ein Wert existieren.

Um die letztere Einschränkung können Sie sich herumklavieren, indem Sie in Standardwerte der Definition der Funktion Standardwerte für die Parameter vorgeben:

```
>>> def f(a, b=2, c=3):
...     print("a= b= c=".format(a, b, c))
...     ↵
>>> f(4)
a=4 b=2 c=3
>>> f(4, 5)
a=4 b=5 c=3
>>> f(4, c=6)
a=4 b=2 c=6
>>> f(4, c=6, b=5)
a=4 b=5 c=6
>>> f(c=6, b=5, a=4)
a=4 b=5 c=6
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() missing 1 required positional argument: 'a'
```

Auch hier gilt die Einschränkung, dass benannte Parameter immer hinter den unbenannten stehen müssen.



Der Vorteil der benannten Parameter mit Standardwerten ist, dass Sie dadurch die Möglichkeit bekommen, die tatsächlich für einen Funktionsaufruf benötigten Parameter auf ein Minimum zu beschränken. Sie sind also nicht gezwungen, bei einer Funktion mit vielen Parametern, von denen in der Regel nur ein paar von ihren Standardwerten abweichen, jeden einzelnen Parameter anzugeben, um die Syntax der Programmiersprache und die Anforderungen der Funktionsdefinition zu erfüllen, sondern müssen nur diejenigen Parameter aufführen, die von den Vorgaben abweichen.

Wenn ein Parameter entweder einen numerischen Wert haben soll oder aber gar nicht vorkommen, ist ein naheliegender Standardwert `None`. Wenn die Funktionsdefinition zum Beispiel

```
def f(a, b=None):
```

ist, dann können Sie innerhalb der Funktion mit etwas wie

```

if b is None:
    <<<<<<                                     b wurde nicht übergeben
else:
    <<<<<<                                     b wurde übergeben


```


prüfen, ob der Parameter angegeben wurde oder nicht. Für Parameter, die sonst Zeichenketten-, Tupel- oder Listen-Werte haben, bieten sich ' ', () oder [] als Standardwerte an. Alle diese ergeben als Boolesche Werte gesehen False, während alle anderen Zeichenketten, Tupel oder Listen True liefern.


Übungen

 **5.1** [!2] Schreiben Sie eine Funktion `skalar(a, b)`, die zwei gleich lange Folgen `a` und `b` übernimmt. Sie soll als Ergebnis das Skalarprodukt der Folgen liefern, also die Summe der paarweisen Produkte der Elemente:

$$\text{skalar}(a, b) = \sum_{k=0}^{n-1} a_k b_k$$

 **5.2** [2] Schreiben Sie eine Funktion, die mit einem oder zwei numerischen Parametern aufgerufen werden kann. Wenn nur ein Parameter angegeben wurde, soll der Rückgabewert der Funktion das Doppelte dieses Parameters sein. Wenn zwei Parameter angegeben wurden, soll der Rückgabewert das Produkt dieser Parameter sein.

 **5.3** [2] Schreiben Sie eine Funktion, die mit einem oder zwei numerischen Parametern aufgerufen werden kann. Wenn nur ein Parameter angegeben wurde, soll der Rückgabewert der Funktion das Doppelte dieses Parameters sein. Wenn zwei Parameter angegeben wurden, soll der Rückgabewert die Summe dieser Parameter sein.

 **5.4** [2] Nach Rózsa Péter (1935) ist die Ackermann-Funktion wie folgt definiert:

$$\begin{aligned}
 a(0, m) &= m + 1 \\
 a(n + 1, 0) &= a(n, 1) \\
 a(n + 1, m + 1) &= a(n, a(n + 1, m))
 \end{aligned}$$

Implementieren Sie die Ackermann-Funktion in Python und experimentieren Sie mit verschiedenen Aufrufen.

5.3 Variable Signaturen

Sie müssen sich bei der Definition von Funktionen nicht notwendigerweise auf eine bestimmte Anzahl von Parametern mit bestimmten Namen festlegen. Betrachten Sie die folgende Funktion:

```

>>> def f(*args):
...     print("args=".format(args))
...     ↩
>>> f()
args=()
>>> f(1)
args=(1,)
>>> f('a', 3, 'x', 55, 17)
args=('a', 3, 'x', 55, 17)

```

Sie können in der Funktionsdefinition einen Parameternamen mit einem vorgestellten Sternchen (*) angeben. Dieser Parameter erhält beim Funktionsaufruf dann ein Tupel als Wert, in dem alle unbenannten Parameter gesammelt sind, für die in der Funktionsdefinition nicht anderweitig Namen vorgesehen sind. Gibt es keine solchen Parameter, ist das Tupel leer. Ein weiteres Beispiel:

```
>>> def g(a, b, *args):
...     print("a= b= args=".format(a, b, args))
...     ↵
>>> g()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: g() missing 2 required positional arguments: 'a' and 'b'
>>> g(1, 2)
a=1 b=2 args=()
>>> g(1, 2, 3, 4)
a=1 b=2 args=(3, 4)
```

Hier müssen Sie auf jeden Fall zwei Parameter angeben, die an a und b zugewiesen werden; alle weiteren Parameter landen in args.



Sie können den *-Parameter nennen, wie Sie wollen. args ist nur eine verbreitete Konvention.

benannte Parameter Es gibt auch einen ähnlichen Mechanismus für benannte Parameter. Damit Sie diesen angemessen würdigen können, müssen Sie wissen, was »Dictionaries« sind. Diese – und ihre Anwendung bei Funktionsaufrufen mit variabler Signatur – behandeln wir im Kapitel 6.

Übungen



5.5 [!2] Schreiben Sie eine Funktion, die beliebig viele numerische Parameter übernimmt und deren arithmetisches Mittel (Summe geteilt durch Anzahl der Werte) als Ergebnis zurückliefert. Was passiert, wenn Sie die Funktion ohne Parameter aufrufen?

5.4 Sichtbarkeitsbereiche

Python unterscheidet zwischen »globalen« Namen (etwa für Variable), die außerhalb von Funktionen definiert worden sind, und »lokalen« Namen innerhalb von Funktionen. Erstere sind unabhängig von letzteren, und es macht nichts, wenn Sie innerhalb von einer Funktion Namen »wiederverwenden«, die außerhalb der Funktion schon anderweitig benutzt wurden:

```
>>> x = 11
>>> def f():
...     x = 22
...     print(x)
...     ↵
>>> print(x)
11
>>> f()
22
>>> print(x)
11
```

Außer für Namen von Variablen (die durch Zuweisungen vergeben werden) gilt das auch für andere Namen, etwa Funktionsnamen – es spricht nichts dagegen, innerhalb einer Funktion eine »lokale« Funktion zu definieren, die dann nur innerhalb dieser verwendet werden kann: »lokale« Funktion

```
>>> def f():
...     def g():
...         print("Hallo aus g")
...         print("Hallo aus f")
...         g()
...     ↩
>>> f()
Hallo aus f
Hallo aus g
>>> g()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'g' is not defined
```

Innerhalb der Funktion `f` können Sie `g` aufrufen, aber außerhalb von `f` ist `g` nicht bekannt.

Wenn Sie in einer Funktion lokale Variable benutzen, können Sie auch in einer »lokalen« Funktion auf diese zugreifen:

```
>>> x = 33
>>> def f():
...     x = 44
...     def g():
...         print(x)
...     ↩
>>> f()
44
>>> x
33
```

Solche lokalen Variablen in »umschließenden« Funktionen nennen wir (aus der Sicht der »inneren« Funktion) »nichtlokal« (*nonlocal*). Damit können wir drei »Sichtbarkeitsbereiche« (engl. *scopes*) für Namen identifizieren: »lokal« in der Funktion selbst, »nichtlokal« in umschließenden Funktionen und »global« außerhalb von allen Funktionen, auf der obersten Ebene. Sichtbarkeitsbereiche



Weiter oben hatten wir gesagt, dass Funktionsdefinitionen auch nur eine Art Zuweisung sind – der Python-Interpreter generiert ein Funktions-Objekt und definiert den Namen der Funktion als Referenz auf das Objekt. Das Funktions-Objekt bleibt erhalten, solange eine Referenz auf es existiert – in unserem Beispiel eben wird der in der Funktion `f` lokale Name `g` gelöscht, wenn die `f` verlassen wird, und darum wird das Funktions-Objekt für `g` wieder entsorgt. Anders ist das, wenn Sie eine Referenz auf die Funktion `g` »nach außen« reichen, etwa als Resultat der Funktion `f`:

```
>>> def f():
...     def g():
...         print("Hallo aus g")
...     ↩
...     return g
>>> func = f()
>>> func()
Hallo aus g
```

Ist dasselbe wie `g`

Fabrikfunktionen



Ausnutzen können Sie das zum Beispiel zur Definition von »Fabrikfunktionen«. Es zeigt sich nämlich, dass nicht nur lokale Funktions-Objekte wie `g` erhalten bleiben, solange Referenzen auf sie existieren, sondern auch nicht-lokale Namen, die in diesen Funktionen benutzt werden:

```
>>> def x_mal(k):
...     def aktion(n):
...         return k * n
...     return aktion
... ↵
>>> doppelt = x_mal(2)
>>> doppelt(3)
6
>>> doppelt(11)
22
>>> dreifach = x_mal(3)
>>> dreifach(11)
33
```

In diesem Beispiel ist `x_mal` eine Funktion, mit der Sie Funktionen »herstellen« können, die einen Parameter übernehmen und das n -Fache des Parameters zurückgeben (wenn die Funktion mit dem Parameter n »hergestellt« wurde).

LEGB-Regel Allgemein gilt für die Namensauflösung in Python die **LEGB-Regel**, kurz für »lokal, umschließend (*enclosing*), global, eingebaut (*built-in*)«. Das heißt, Namen werden zuerst im lokalen Sichtbarkeitsbereich gesucht, dann sukzessive in den umschließenden Funktionen (falls es welche gibt), dann im globalen Sichtbarkeitsbereich und zuletzt unter den in Python fest eingebauten Namen.

Dass ein Name aufgelöst, also das dazugehörige Objekt gefunden, werden kann, heißt noch lange nicht, dass Sie den Namen auch umdefinieren können. Wie Sie schon gesehen haben, führen Zuweisungen an Variable innerhalb einer Funktion dazu, dass lokale Variable angelegt werden, selbst wenn gleichnamige nichtlokale oder globale Variable bereits existieren. Dasselbe gilt für Funktionsdefinitionen.

Zumindest für Variable können Sie dafür sorgen, dass Zuweisungen innerhalb einer Funktion »global« ausgeführt werden, indem Sie die betreffenden Variablennamen *vor der ersten Verwendung in der Funktion* mit dem Kommando `global` zu globalen Variablen erklären:

```
>>> x, y = 1, 2
>>> def f():
...     global x, y
...     x, y = 5, 6
... ↵
>>> x, y
(1, 2)
>>> f()
>>> x, y
(5, 6)
```

(Die in `global`-Kommandos aufgezählten Namen dürfen nicht als formale Parameter, Schleifenkontrollvariablen in `for`-Schleifen oder Funktionsnamen in lokalen Funktionsdefinitionen benutzt werden. Wenn Sie sich auf »normale« Variable beschränken, sind Sie auf der sicheren Seite.)

nonlocal



In Analogie zu `global` gibt es auch das Kommando `nonlocal`, mit dem Sie für die betreffenden Namen im Wesentlichen den lokalen Sichtbarkeitsbereich

bei der Namensauflösung und -definition überspringen können. Die Namen, die Sie in `nonlocal` aufzählen, müssen bereits in einer umschließenden Funktion (aber nicht auf der globalen Ebene) existieren, um Mehrdeutigkeiten zu vermeiden. Im Zweifelsfall wird die »innerste« nichtlokale Definition benutzt:

```
>>> def f():
...     x, y = 1, 3
...     def g():
...         y = 2
...         def h():
...             nonlocal x, y
...             x, y = 5, 6
...             h()
...             print("y =", y)
...         g()
...         print("x =", x, "y =", y)
...     ↵
>>> f()
y = 6
x = 5 y = 3
```

Die Funktion `h` überschreibt hier die Variable `x` aus der Funktion `f` und die Variable `y` aus der Funktion `g`. Die Variable `y` aus der Funktion `f` bleibt, wie sie ist, weil die Variable `y` aus der Funktion `g` eher gefunden wird.

Übungen



5.6 [!2] Definieren Sie eine Fabrikfunktion `makeUnderline`, die Funktionen definiert, mit denen Sie eine Zeichenkette mit einem beliebigen Zeichen »unterstreichen« können. Für Anwender soll sich das etwa so präsentieren:

```
>>> dash = makeUnderline('-')
>>> dash("Huhu")
'Huhu\n---'
>>> print(dash("Huhu"))
Huhu
----
>>> stars = makeUnderline('*')
>>> print(stars("Hallo Welt!"))
Hallo Welt!
*****
```

Zusammenfassung

- Mit dem Python-Kommando `def` können Sie Funktionen definieren. `def` ist ein ausführbares Kommando.
- Funktionen können Parameter haben und mit `return` einen Rückgabewert liefern.
- Einfache Python-Datentypen werden als »Wert« übergeben und strukturierte Datentypen als »Referenz«. Gegebenenfalls müssen Sie kopieren.
- Funktionen dürfen sich (oder einander) rekursiv aufrufen. Aus Sicherheitsgründen ist die maximale Rekursionstiefe nach oben begrenzt.
- Beim Funktionsaufruf können die Namen der Parameter genannt werden. Dann dürfen sie in beliebiger Reihenfolge angegeben werden.
- In der Funktionsdefinition können Sie Standardwerte für Parameter vorgeben, die in Kraft treten, wenn der betreffende Parameter im Funktionsaufruf nicht angegeben wurde.
- Funktionen können variable Parametersignaturen haben.
- Python unterscheidet für Namen die Sichtbarkeitsbereiche »lokal«, »nicht-lokal« und »global«, ferner gibt es eingebaute Namen. Bei der Namensauflösung gilt die »LEGB-Regel«.



6

Strukturierte Datentypen: Dictionaries und Mengen

Inhalt

6.1	Dictionaries	82
6.2	Funktionsaufrufe mit beliebigen benannten Parametern	89
6.3	Dictionary Comprehensions	90
6.4	Mengen	92

Lernziele

- Dictionaries und Mengen einsetzen können
- Funktionsaufrufe mit beliebigen benannten Parametern verstehen
- *Dictionary comprehensions* einsetzen können

Vorkenntnisse

- Einfache (Kapitel 2) und simple strukturierte (Kapitel 4) Datentypen in Python
- Verzweigungen und Schleifen in Python (Kapitel 3)
- Erfahrung mit anderen Programmiersprachen ist von Vorteil

6.1 Dictionaries

Python unterstützt diverse »Container«-Typen, die zusammengehörende Python-Objekte abspeichern können. Folgen, also Listen und Tupel (und Zeichenketten), speichern Elemente in einer linearen Reihenfolge und erlauben den Zugriff auf einzelne Elemente über (numerische) Indizes und Teilfolgen über Slices. Manchmal ist es aber nützlich, statt numerischer Indizes zum Beispiel Zeichenketten zu verwenden, um Elemente eines Container-Typs speichern und wiederfinden zu können.



Stellen Sie sich zum Beispiel vor, Sie möchten zählen, wie oft bestimmte Wörter in einem Text vorkommen. Dafür ist es nützlich, wenn Sie die Wörter als Indizes benutzen können, um die aktuelle Anzahl zu finden oder zu manipulieren.

Dictionary Der Python-Container-Typ, der das erlaubt, heißt Dictionary¹. Ein literales Dictionary können Sie mit Hilfe von geschweiften Klammern hinschreiben:

```
>>> d = {'A': 1, 'B': 2, 'C': 3}
```

Zum Zugriff auf einzelne Elemente werden wie bei Listen, Tupeln und Zeichenketten eckige Klammern benutzt:

```
>>> d['B']
2
>>> d['D'] = 4
```

Wenn Sie ein Dictionary ausgeben, bekommen Sie die Elemente in einer nicht vorhersehbaren Reihenfolge:

```
>>> d
{'D': 4, 'C': 3, 'B': 2, 'A': 1} Oder sonstwas
```

Werte von Elementen Die Werte von Elementen in Dictionaries dürfen beliebige Python-Objekte sein (also zum Beispiel auch Tupel, Listen oder andere Dictionaries). Auch bei den Indizes – oft »Schlüssel« genannt – sind Sie nicht auf Zeichenketten beschränkt: Praktisch jeder unveränderbare Python-Typ ist zulässig, außer Zeichenketten also zum Beispiel auch Zahlen oder Tupel. Objekte, die Listen, Dictionaries oder andere veränderbare Typen enthalten, kommen als Schlüssel nicht in Frage.

Schlüssel



Damit ein Objekt als Dictionary-Schlüssel in Frage kommt, muss es *hashable* sein (noch so ein unübersetzbares Wort). Mit anderen Worten, es muss einen über seine Lebensdauer unveränderlichen Hash-Wert haben (berechnet über seine Methode `__hash__()`) und es muss mit anderen Objekten verglichen werden können (es braucht eine Methode `__eq__()`). Objekte, die als »gleich« gelten, müssen denselben Hash-Wert haben. (Wahrscheinlich können Sie diesen Absatz erst wirklich würdigen, wenn Sie Kapitel 7 gelesen haben.)



Alle unveränderbaren eingebauten Datentypen von Python sind *hashable*, alle veränderbaren (etwa Listen und Dictionaries) sind es nicht. Zahlen werden über ihren Wert verglichen, das heißt, die Zahlen 1 (Ganzzahl) und 1.0 (Gleitkommazahl) adressieren als Schlüssel dasselbe Element eines Dictionary².

¹Wir sparen es uns auch an dieser Stelle, diesen Begriff einzudeutschen (damit würden wir niemandem einen Gefallen tun). Andere Programmiersprachen haben sowas übrigens auch, jedenfalls im Großen und Ganzen: Perl nennt den Typ *hash*, Awk *associative array*, und Tcl und PHP sagen dazu einfach nur *array*. Nur C-Programmierer schauen in die Röhre, da weder Sprache noch Standardbibliothek etwas Entsprechendes zu bieten haben.

²Sie sollten Gleitkommazahlen als Schlüssel für Dictionaries nicht überstrapazieren, da sie von Python möglicherweise nicht exakt dargestellt werden können.



Alle Objekte, die zu benutzerdefinierten Klassen gehören, sind standardmäßig *hashable*: Sie sind ungleich allen Objekten außer sich selbst und ihr Hash-Wert entspricht dem Rückgabewert der Funktion `id()`, angewendet auf das betreffende Objekt. (Natürlich hält Sie gegebenenfalls niemand davon ab, die »magischen« Methoden `__hash__()` und `__eq__()` so zu definieren, wie Sie wollen.)

Der Konstruktor `dict()` kann aus einer Liste oder einem Tupel ein Dictionary machen. Voraussetzung dafür ist, dass die Elemente dieser Folgen Paare aus Schlüssel und Wert sind: Konstruktor `dict()`

```
>>> dict((('bla', 'blubb'), ('bli', 'fasel')))
{'bla': 'blubb', 'bli': 'fasel'}
```

Auch für Dictionaries gibt es einige nützliche Funktionen und Methoden. Das Funktionen und Methoden beginnt mit Standardkost. Die Funktion `len()` liefert die Anzahl von Elementen in einem Dictionary:

```
>>> len(d)
4
```

Mit den Operatoren `in` und `not in` können Sie überprüfen, ob im Dictionary ein Operatoren `in` und `not in` Element mit einem bestimmten Schlüssel vorkommt:

```
>>> 'A' in d
True
>>> 'Z' in d
False
>>> 'Z' not in d
True
```

Das Kommando `del` entfernt ein Element mit einem gegebenen Schlüssel aus dem Kommando `del` Dictionary:

```
>>> d['C']
3
>>> del d['C']
>>> d['C']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'C'
```

(Hier sehen Sie auch gleich, was passiert, wenn Sie auf ein Element zugreifen, das es im Dictionary gar nicht gibt.)

Den `KeyError` beim Zugriff auf ein nicht vorhandenes Element können Sie vermeiden, indem Sie die Methode `get()` benutzen. Wenn der betreffende Schlüssel Methode `get()` nicht existiert, liefert die Methode `None` oder einen vorgegebenen Standardwert:

```
>>> d.get('A')
1
>>> d.get('C')
>>> d.get('C', 99999)
99999
```

Die Methode `setdefault()` prüft, ob es im Dictionary ein Element mit dem angegebenen Schlüssel gibt. Falls nein, fügt sie das Element mit dem angegebenen Wert (ersatzweise `None`) hinzu. In jedem Fall liefert sie anschließend den Wert des Elements zurück:

```
>>> d.setdefault('A', 111)
1
>>> d.setdefault('C', 111)
111
>>> d['C']
111
```

Wenn Sie ein bestimmtes Element aus dem Dictionary entfernen wollen, sich aber für dessen Wert interessieren, können Sie die Methode `pop()` benutzen:

```
>>> d
{'D': 4, 'C': 111, 'B': 2, 'A': 1}
>>> d.pop('C')
111
>>> d
{'D': 4, 'B': 2, 'A': 1}
```

Wenn Sie versuchen, ein nicht existierendes Element zu »poppen«, dann bekommen Sie einen `KeyError`, es sei denn, Sie haben einen Standardwert angegeben:

```
>>> d.pop('Z')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Z'
>>> d.pop('Z', 'Nicht enthalten')
'Nicht enthalten'
```

Kopieren und Leeren Mit der Methode `copy()` können Sie ein Dictionary (seicht) kopieren, und `clear()` löscht alle Elemente (das Dictionary ist danach leer):

```
>>> d = {'A': 1, 'B': 2, 'C': 3, 'D': 4}
>>> d0 = d.copy()
>>> len(d0)
4
>>> d0.clear()
>>> len(d0)
0
```

Elemente eines Dictionary sukzessive durchlaufen Sie können alle Elemente eines Dictionary sukzessive durchlaufen, einfach indem Sie eine `for`-Schleife verwenden:

```
>>> for k in d:
...     print(k)
... 
D
C
B
A
```

Liste aller Schlüssel Die Reihenfolge ist dabei (wie üblich) nicht vorhersehbar. Eine Liste aller Schlüssel des Dictionary bekommen Sie über die Methode `keys()`:

```
>>> d.keys()
dict_keys(['D', 'C', 'B', 'A'])
```



Sie bekommen genau genommen keine echte Liste, sondern ein Objekt vom Typ `dict_keys`. Der Vorteil ist, dass dieses Objekt sich anpasst, wenn das Dictionary sich ändert:

```

>>> k = d.keys()
>>> k
dict_keys(['D', 'C', 'B', 'A'])
>>> d['E'] = 5
>>> k
dict_keys(['D', 'C', 'B', 'E', 'A'])
>>> del d['B']
>>> k
dict_keys(['D', 'C', 'E', 'A'])

```

Die Methoden `values()` und `items()` funktionieren entsprechend, liefern allerdings die Werte beziehungsweise Schlüssel-Wert-Paare (als Tupel) des Dictionary.



Die genaue Reihenfolge, in der die Elemente geliefert werden, ist nicht garantiert, aber was garantiert wird, ist, dass – solange Sie das Dictionary nicht ändern – die Methoden `keys()`, `values()` und `items()` die Elemente in derselben Reihenfolge durchlaufen. Sie können also Sachen machen wie

```

>>> rev = dict(zip(d.values(), d.keys()))

```

Andersrum

(das funktioniert natürlich nur, solange jeder *Wert* im Dictionary nur einmal vorkommt).

Sie können ein Dictionary »destruktiv« durchlaufen, indem Sie die Methode `popitem()` verwenden. Diese entfernt ein (unvorhersehbares) Element aus dem Dictionary und gibt es als Paar zurück: Dictionary »destruktiv« durchlaufen

```

>>> d = {'A': 'Alpha', 'B': 'Bravo', 'C': 'Charlie'}
>>> while len(d) > 0:
...     print("{} = {}".format(*d.popitem()))
...     ↩
C = Charlie
B = Bravo
A = Alpha
>>> d
{}

```

Beachten Sie die trickreiche Parameterübergabe in der `format()`-Methode: Die Methode erwartet eigentlich zwei separate Parameter, aber `d.popitem()` liefert ein Tupel der Länge 2. Mit dem vorgesetzten Stern wird dieses Tupel in einzelne Parameter aufgeteilt.

Die Methode `update()` dient dazu, ein Dictionary um zusätzliche Elemente zu erweitern. Der Parameter ist entweder ein anderes Dictionary oder eine Folge von Paaren (Tupeln oder Listen der Länge 2). Treten darin Schlüssel auf, die bereits im Dictionary existieren, dann werden die betreffenden Elemente überschrieben: Dictionary erweitern

```

>>> d1 = {'B': 'Beta', 'K': 'Kappa', 'T': 'Tau'}
>>> d.update(d1)
>>> d
{'C': 'Charlie', 'B': 'Beta', 'K': 'Kappa', 'T': 'Tau', 'A': 'Alpha'}

```

Hier ein Beispiel für den praktischen Gebrauch von Dictionaries: »Monoalphabetische Substitution« ist ein simples Verschlüsselungsverfahren, bei dem jeder Buchstabe im Alphabet durch einen anderen Buchstaben ersetzt wird. Ausgehend von einem geheimen Codewort (etwa FLIEGENPILZ) könnte sich die Ersetzungsvorschrift

```

ABCDEFGHIJKLMNPOQRSTUVWXYZ
FLIEGNPZABCDHJKMOQRSTUVWXYZ

```

ergeben (zur Verschlüsselung wird jeder Buchstabe durch den darunterstehenden ersetzt, zur Entschlüsselung durch den darüberstehenden). Umlaute lassen wir mal außer Acht.

Um beliebige Texte zu verschlüsseln, müssen wir zunächst die Ersetzungsvorschrift aufstellen. Natürlich könnten wir das per Hand machen und die Tabelle fest ins Programm aufnehmen, aber es ist besser, von einer Zeichenkette mit dem geheimen Codewort auszugehen. Was wir zum Schluss haben wollen, ist ein Dictionary encode mit den Klartextbuchstaben als Schlüssel und den Codebuchstaben als Wert, so dass wir einen Text einfach mit

```

cipher = ''
for c in text.upper():
    cipher += encode.get(c, ' ')

```

verschlüsseln können (alles, was nicht im Dictionary encode steht, wird durch ein Leerzeichen ersetzt).

Anfangen könnten wir mit zwei Variablen

```

alphabet = 'ABCDEFGHIJKLMNPOQRSTUVWXYZ'
code = 'FLIEGNPILZ'

```

Wir müssen beachten, dass im Codewort ein paar Buchstaben mehrmals vorkommen. Von diesen möchten wir jeweils nur das erste Vorkommen beachten. Unser prinzipieller Ansatz besteht darin, zuerst die *Dekodiertabelle* aufzustellen, indem wir ein Dictionary decode konstruieren, das die *Codebuchstaben* als Schlüssel und die Klartextbuchstaben als Wert hat. Auf diese Weise können wir einfach sicherstellen, dass jeder Codebuchstabe nur einmal benutzt wird. Wir erzeugen also im Prinzip die zweite Zeile unserer Übersetzungstabelle (FLIEGNPZABC...) als Schlüssel für das Dictionary:

```

decode = {}
buchstabe = ord('A') # Position im Zeichencode (65)
for c in code + alphabet:
    if c not in decode: # als Codebuchstabe unbenutzt
        decode[c] = chr(buchstabe) # Klartext: nächster Buchstabe
        buchstabe += 1

```

Die Funktion `chr()` konvertiert einen numerischen Zeichencode (vulgo »ASCII-Code«) in das dazugehörige Zeichen (als Zeichenkette der Länge 1). Das heißt, die Variable `buchstabe` durchläuft die Werte A, B, ...



Wir können sicher sein, dass `buchstabe` am Ende genau bei Z herauskommt, weil in `code` keine Zeichen stehen, die nicht auch in `alphabet` stehen. Da jeder Großbuchstabe nur maximal einmal in `decode` einsortiert wird (Dubletten werden durch die »`c not in decode`«-Abfrage übersprungen) und `alphabet` alle Großbuchstaben enthält, wissen wir, dass `decode` zum Schluss Einträge für alle Großbuchstaben enthält und sonst nichts.

Anschließend können wir das Dictionary encode aufbauen, einfach indem wir das Dictionary decode elementweise »umdrehen«, also jeden Schlüssel zum Wert und jeden Wert zum Schlüssel machen. Das geht, weil wir wissen, dass die Werte in `decode` genau die Buchstabe A bis Z sind und es keine Dubletten geben kann:

```

encode = dict([(v, k) for k, v in decode.items()])

```

Das tatsächliche Verschlüsseln hatten wir ja schon besprochen. Wir holen uns hier den zu verschlüsselnden Text von der Kommandozeile – wenn wir »import sys« benutzen, stehen die Kommandoparameter des Programms als Liste in `sys.argv` zur Verfügung. Dabei ist das erste Element der (für uns uninteressante) Programmname, der Rest kann unser Text sein:

```

cipher = ''
for c in " ".join(sys.argv[1:]).upper():
    cipher += encode.get(c, ' ')
print(cipher)

```

Dann bekommen wir etwas wie

```

$ python monoalph.py DAS PFERD FRISST KEINEN GURKENSALAT
EFR MNGQE NQARRS CGAJGJ PTQCGJRFDFS

```

Übungen



6.1 [1] Angenommen, Sie haben zwei gleich lange Folgen `a` und `b`. Geben Sie an, wie Sie mit `dict()` – und ohne Verwendung einer expliziten `for`- oder `while`-Schleife – aus diesen beiden Listen ein Dictionary `d` konstruieren, bei dem die Elemente von `a` als Schlüssel und die von `b` als korrespondierende Werte fungieren. (Mit anderen Worten: Wenn `a` den Wert `abc` hat und `b` den Wert `(11, 22, 33)`, soll `d['b']` den Wert `22` bekommen.)



6.2 [2] Nach altem Brauch werden chiffrierte Texte nicht mit den ursprünglichen Leerzeichen übermittelt, sondern zum Beispiel in Gruppen von je fünf Zeichen, wobei die Leerzeichen vorher entfernt wurden. Also zum Beispiel statt

```
EFR MNGQE NQARRS CGAJGJ PTQCGJRFDFS
```

lieber

```
EFRMN GQENQ ARRSC GAJGJ PTQCG JRFDF S
```

(im wirklichen Leben würde man noch ein paar zufällige Zeichen ans Ende hängen, die beim Entschlüsseln keinen Sinn ergeben, um die letzte Fünfergruppe voll zu machen). Ändern Sie das Chiffrierprogramm so, dass die Ausgabe in Fünfergruppen erfolgt.



6.3 [1] Können Sie die Verschlüsselungsroutine im Chiffrierprogramm auch ohne eine explizite `for`-Schleife ausdrücken. (Wobei wir nicht »ja, mit einer `while`-Schleife« meinen.)



6.4 [!2] Ändern Sie das Chiffrierprogramm so, dass es den Text auf der Kommandozeile dechiffriert statt chiffriert. Wie können Sie beide Funktionen bequem im selben Programm realisieren, so dass es entweder chiffriert oder dechiffriert?



6.5 [!2] Schreiben Sie ein Programm, das berechnet, wie viele Tastendrucke man braucht, um ein gegebenes Wort auf der Zifferntastatur eines Telefons einzutippen (denken Sie an SMS aus der Prä-Smartphone-Ära). Ignorieren Sie dabei Groß- und Kleinschreibung, Umlaute und die Textvoraussage moderner Telefone. Das Wort können Sie sich von der Kommandozeile holen – nach einem

```
>>> import sys
```

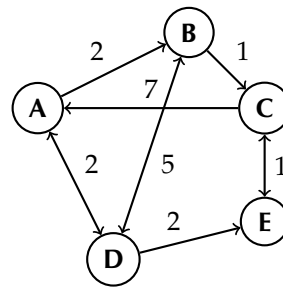
steht es in `sys.argv[0]`. Hier für Smartphone-Besitzer eine Erinnerung an die Tastatur:

---	ABC	DEF
1	2	3
GHI	JKL	MNO
4	5	6
PQRS	TUV	WXYZ
7	8	9

(*Tipps:* (a) Verwenden Sie ein Dictionary, um für jeden Buchstaben des Alphabets die Anzahl der benötigten Tastendrucke zu speichern. Dann müssen Sie nur noch über die Buchstaben des Worts laufen und die Tastendrucke addieren. Wie können Sie dieses Dictionary bequem initialisieren? (b) Verwenden Sie ggf. die Zeichenketten-Methode `upper()`, um das Eingabewort in Großbuchstaben umzuwandeln, wenn Ihr Dictionary Großbuchstaben als Schlüssel verwendet.) Für Zusatzpunkte: Betrachten Sie alle Wörter auf der Kommandozeile (implizite Leerzeichen zwischen den Wörtern gelten als ein Tastendruck).



6.6 [2] In Python können Sie Dictionaries verwenden, um einen »gewichteten gerichteten Graphen« zu repräsentieren. Die entsprechende Datenstruktur nennt man eine »Adjazenzliste«, weil sie für jeden Knoten alle Knoten angibt, zu denen eine Verbindung (Kante) besteht (und das Gewicht der betreffenden Kante). Geben Sie eine Adjazenzliste für den folgenden Graphen an:



6.7 [3] »Dijkstras Algorithmus« [?, S. 207 f.] ist eine Methode, um ausgehend von einem Knoten in einem gewichteten gerichteten Graphen die »kürzeste« Verbindung (also die Verbindung mit der minimalen Summe von Gewichten einzelner Kanten) zu allen anderen Knoten im Graph zu finden. Das Ganze geht – in einer halbwegs »mathematischen« Notation – wie folgt. Dabei ist v der Graph bestehend aus den Knoten v_0, \dots, v_{n-1} und $l[v, w]$ ist das Gewicht der Kante zwischen den Knoten v und w bzw. ∞ , wenn zwischen den Knoten keine Kante existiert. Am Ende enthält D die minimale »Entfernung« von v_0 zu jedem anderen Knoten:

```

1:  $S \leftarrow \{v_0\}$  Startknoten
2:  $D[v_0] \leftarrow 0$ 
3: for all  $v \in V - \{v_0\}$  do
4:    $D[v] \leftarrow l[v_0, v]$   $\infty$ , falls keine Kante
5: end for
6: while  $S \neq V$  do
7:   Wähle  $w \in V - S$  mit minimalem  $D[w]$ 
8:   Füge  $w$  zu  $S$  hinzu
9:   for all  $v \in V - S$  do
10:     $D[v] \leftarrow \min(D[v], D[w] + l[w, v])$ 
11:   end for
12: end while

```

Implementieren Sie Dijkstras Algorithmus in Python.

6.2 Funktionsaufrufe mit beliebigen benannten Parametern

In Kapitel 5 hatten wir versprochen, zu erklären, wie Sie Python-Funktionen definieren können, die mit beliebigen benannten Parametern umgehen können – auch solchen, die bei der Funktionsdefinition nicht im Einzelnen angegeben wurden. Zur Erinnerung: Beliebige *unbenannte* Parameter können Sie übernehmen, indem Sie in der Funktionsdefinition einen Parameter namens `*args` deklarieren. Alle unbenannten Parameter, die nicht zu tatsächlich deklarierten Parametern in der Funktionsdefinition passen, werden dann zu einem Tupel zusammengefasst, das innerhalb der Funktion unter dem Namen `args` zur Verfügung steht.



Der Name `args` ist, wie erwähnt, reine Konvention und der Mechanismus würde genauso funktionieren, wenn Sie irgendeinen anderen Namen wählen. Der vorgesetzte Stern macht den Unterschied. Es gibt aber auch keinen speziellen Grund, `args` nicht zu benutzen.

Der Mechanismus für beliebige *benannte* Parameter funktioniert ganz ähnlich: Wenn Sie in der Funktionsdefinition einen Parameter namens `**kwargs` deklarieren (wichtig: *zwei* vorgesetzte Sterne), dann ist `kwargs` innerhalb der Funktion ein Dictionary, das alle benannten Parameter enthält, die nicht zu tatsächlich deklarierten Parametern in der Funktionsdefinition passen. Betrachten Sie zum Beispiel

```
>>> def f(**kwargs):
...     for k in kwargs:
...         print("{} => {}".format(k, kwargs[k]))
...
>>> f()
>>> f(a=1, b=2)
b => 2
a => 1
>>> f(1, b=2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes 0 positional arguments but 1 was given
```



Auch hier ist der Name `kwargs` (kurz für *keyword arguments*) reine Konvention. Sie könnten also absolut auch etwas Anderes benutzen, wenn Sie wollten.

Es spricht nichts dagegen, die Übergabemechanismen zu kombinieren. Sie müssen nur darauf achten, dass die »Stern-Parameter« ganz am Ende der Parameterliste stehen: Übergabemechanismen kombinieren

```
>>> def g(a, b=0, *args, **kwargs):
...     print("a = {}".format(a))
...     print("b = {}".format(b))
...     print("Weitere unbenannte Parameter:")
...     for p in args:
...         print(args[p])
...     print("Weitere benannte Parameter:")
...     for p in kwargs:
...         print("{} = {}".format(p, kwargs[p]))
...
>>> g(123, 456, 789, 999, c="blubb", d="bla")
a = 123
b = 456
```

```

Weitere unbenannte Parameter:
789
999
Weitere benannte Parameter:
c = blubb
d = bla

```

Wenn zwei oder mehr unbenannte Parameter im Funktionsaufruf stehen, werden die ersten beiden davon auf jeden Fall zu `a` und `b` – wenn später noch ein benannter Parameter `b` auftaucht, gibt es eine Fehlermeldung. Höchstens wenn Sie nur einen unbenannten Parameter im Aufruf haben, können Sie etwas schreiben wie

```
>>> g(123, c="bli", b=456, d="bla")
```

»Durchreichen« Eine wichtige Anwendung ist das »Durchreichen« von in `args` und `kwargs` gesammelten Parametern an andere Funktionen. Wenn Sie etwas wie `*args` nicht in einer Funktionsdefinition verwenden, sondern in einem Funktionsaufruf, dann werden die Elemente der Folge `args` zu einzelnen unbenannten Parametern gemacht. Wenn Sie etwas wie `**kwargs` in einem Funktionsaufruf verwenden, dann werden die Elemente des Dictionary `kwargs` zu einzelnen *benannten* Parametern gemacht. Das heißt, etwas wie

```

>>> def f(a=0, b=0):
...     print("a={} b={}".format(a, b))
...     ←
>>> k = {'a': 123, 'b': 456}
>>> f(**k)
a=123 b=456

```

funktioniert durchaus – die Elemente des Dictionary werden auf die gleichnamigen Parameter der Funktion verteilt. Das funktioniert sogar mit literalen Dictionaries und (dank der in der Funktionsdefinition angegebenen Standardwerte) mit Dictionaries, die nicht für jeden Parameter in der Funktionsdefinition ein passendes Element enthalten:

```

>>> f(**{'b': 789})
a=0 b=789

```

Sie können genauso gut eine Funktion definieren, die sich die übergebenen benannten Parameter anschaut, Änderungen macht und dann die eigentliche Funktion aufruft:

```

>>> def g(**kwargs):
...     c = kwargs.pop('c', None)
...     if c is not None:
...         kwargs['a'] = c+2
...     f(**kwargs)
...     ←
>>> g(b=123, c=987)
a=989 b=123

```

Diese Technik ist vor allem bei der objektorientierten Programmierung nützlich. Wir werden im Kapitel 7 noch weitere Beispiele hierfür sehen.

6.3 Dictionary Comprehensions

Im Kapitel 4 haben wir *list comprehensions* als bequeme Möglichkeit kennengelernt, Listen zu konstruieren, ohne diese Element für Element in einer Schleife

aufbauen zu müssen. Entsprechend dienen *dictionary comprehensions* dazu, Dictionaries zu konstruieren, ohne diese Element für Element in einer Schleife aufbauen zu müssen. Das könnte zum Beispiel so aussehen:

```
>>> { c: c*(ord(c)-64) for c in 'ABCDE' }
{'B': 'BB', 'E': 'EEEE', 'D': 'DDDD', 'C': 'CCC', 'A': 'A'}
```

Dictionary comprehensions haben diverse nützliche Anwendungen. Zum Beispiel können Sie damit bequem Dictionaries initialisieren:

```
>>> { k: 0 for k in "abcdefg" }
{'e': 0, 'g': 0, 'b': 0, 'c': 0, 'f': 0, 'd': 0, 'a': 0}
```

Oder stellen Sie sich vor, Sie müssen die Elemente in einem Dictionary bearbeiten. Zum Beispiel könnten Sie ein Dictionary haben, das für verschiedene Buchstaben in einem Text (als Schlüssel) angibt, wie oft diese Buchstaben vorkommen (Wert):

```
>>> f = {'a': 12, 'b': 5, 'e': 15, 'A': 3, 'C': 2, 'E': 10}
```

Allerdings unterscheidet dieses Dictionary zwischen Groß- und Kleinbuchstaben, und Sie interessieren sich für die Gesamtanzahl unabhängig von Groß- und Kleinschreibung. Wie können Sie ein neues Dictionary konstruieren, das für jeden Buchstaben – beispielsweise mit der Großbuchstaben-Version als Schlüssel – die Summe der Anzahlen für Groß- und Kleinschreibung enthält? Mit *dictionary comprehensions*, nichts einfacher als das:

```
>>> f0 = { c.upper(): f.get(c.upper(), 0)+f.get(c.lower(), 0)
...       for c in f }
>>> f0
{'E': 25, 'B': 5, 'C': 2, 'A': 15}
```

Natürlich können Sie auch hier mit *if* operieren:

```
>>> s = "DONAUDAMPFSCHIFFFAHRTSGESELLSCHAFT"
>>> { c: s.count(c) for c in s if c in "AEIOU" }           Nur Vokale
{'I': 1, 'E': 2, 'U': 1, 'O': 1, 'A': 4}
```

Auch verschachtelte *for*-Klauseln sind kein Problem:

```
>>> { '{0}{1}'.format(j, q): 0 for j in range(2010,2015)
...   for q in range(1, 5) }
{'2011Q4': 0, '2012Q2': 0, '2014Q2': 0, '2011Q3': 0, '2011Q2': 0,
 '2014Q4': 0, '2013Q2': 0, '2013Q4': 0, '2010Q4': 0, '2010Q3': 0,
 '2014Q3': 0, '2011Q1': 0, '2012Q3': 0, '2010Q1': 0, '2013Q1': 0,
 '2013Q3': 0, '2010Q2': 0, '2014Q1': 0, '2012Q4': 0, '2012Q1': 0}
```

Übungen



6.8 [!1] Geben Sie eine *dictionary comprehension* an, die in Analogie zum Beispiel oben im Wort OBERFINANZLANDESDIREKTIONSPFORTENKLINKE die Häufigkeiten der Konsonanten bestimmt.



6.9 [3] Schreiben Sie ein Programm, das Zeilen von der Standardeingabe liest. Jede Zeile enthält entweder eine ganze («arabische») Zahl von 1 bis 3999 (einschließlich) oder eine römische Zahl von I bis MMMCMXCIX (einschließlich). Das Programm soll die jeweils andere Zahlendarstellung ausgeben oder eine Fehlermeldung, wenn eine Ganzzahl außerhalb des gültigen Bereichs oder eine ungültige römische Zahl eingegeben wurde. –

Zur Erinnerung: Die gültigen römischen Zahlen sind: I=1, V=5, X=10, L=50, C=100, D=500, M=1000. Es dürfen maximal drei gleiche Ziffern hintereinanderstehen; für Zahlen wie 9, 40 usw. muss getrickst werden (9=IX, 40=XL usw.).

(*Tipps:* Stellen Sie zunächst eine Umwandlungstabelle von arabischen nach römischen Zahlen auf, die für jede arabische Zahl im Zahlenbereich die zugehörige römische Zahl angibt. Diese Umwandlungstabelle können Sie dann bequem zu einem Dictionary machen, das römische Zahlen in ihr arabisches Äquivalent umwandelt. Fangen Sie damit an, die arabischen und römischen Zahlen und die »Sonderfälle« zu sammeln:

```
d = (., (10, 'X'), (9, 'IX'), (5, 'V'), (4, 'IV'), (1, 'I'))
```

(Wir sparen uns zur Verdeutlichung hier mal die größeren Zahlen.) Aus Gründen, die gleich noch offensichtlich werden, ist diese Liste absteigend geordnet. Die römische Darstellung für eine Zahl t können Sie dann nach dem folgenden Algorithmus bestimmen:

```
R = ''
for a, S in d do
    (k, t) ← (⌊t/a⌋, t mod a)
    R ← R || (S * k)
end for
```

Ergebnis
In absteigender Reihenfolge
 k -mal die römische »Ziffer« S

Betrachten wir als Beispiel $t = 17$. Im ersten Schleifendurchlauf ist $a = 10$, t/a ist 1,7 und der ganzzahlige Anteil davon 1. Zum Ergebnis wird also ein »X« beigetragen und t ist jetzt der Divisionsrest, also 7. Im zweiten Schleifendurchlauf ist $a = 9$, und da $7/9$ auf 0 abgerundet wird, kommt kein »IX« ins Ergebnis. Der dritte Durchlauf mit $a = 5$ macht das Ergebnis zu »XV« und lässt t auf dem Wert 2; der vierte Durchlauf mit $a = 1$ beendet das Ganze, das Ergebnis ist jetzt »XVII«, weil $2/1 = 2$.)

6.4 Mengen

Als letzten Python-Container-Typ betrachten wir noch Mengen. Mengen sind ungeordnete Sammlungen verschiedener Objekte (jedes Objekt kann nur einmal in derselben Menge auftauchen). Sie eignen sich für Anwendungen wie Enthaltensein-Tests, das Entfernen von Duplikaten oder allgemein für Algorithmen, die Mengenoperationen (Vereinigungs-, Schnitt-, Restmenge, ...) voraussetzen.

Python unterstützt zwei Arten von Mengen, nämlich veränderbare (Typ `set` und nicht veränderbare (Typ `frozenset`). Diese verhalten sich zueinander etwa wie Listen und Tupel – erstere können nach dem Erzeugen noch geändert werden, letztere nicht. Dafür können Sie `frozensets` als Schlüssel in Verzeichnissen verwenden (sie sind im Gegensatz zu `sets` *hashable*).

set-Literale Veränderbare Mengen können Sie erzeugen, indem Sie die gewünschten Elemente in geschweifte Klammern schreiben. Die Elemente müssen *hashable* sein – Zeichenketten, Zahlen oder Tupel kommen zum Beispiel in Frage:

```
>>> s = { 'Alpha', 'Bravo', 'Charlie', 'Bravo' }
>>> s
{'Alpha', 'Bravo', 'Charlie'}
```

(Das doppelte Bravo wurde anscheinend korrekt entfernt.) Alternativ können Sie den Konstruktor `set()` benutzen, der eine Folge übernimmt:

```
>>> s = set(('Alpha', 'Bravo', 'Charlie', 'Bravo'))
>>> s
{'Alpha', 'Bravo', 'Charlie'}
```



Wenn Sie eine leere Menge brauchen, müssen Sie den Konstruktor `set()` verwenden: `»{}«` liefert Ihnen ein leeres Dictionary, keine leere Menge.



Eine nicht veränderbare Menge müssen Sie über den Konstruktor `frozenset()` anlegen (dieser benimmt sich wie `set()`). Es gibt keine literale Darstellung für `frozensets`.

Mengen (beide Geschmacksrichtungen) unterstützen `len()`, die Operatoren `in` und `not in` sowie das Iterieren über die Elemente mit `for`. Da es keine Ordnung gibt, können Sie Operatoren wie Indizieren, Slices und andere »folgenmäßigen« Zugriffsmethoden nicht verwenden. Dafür bieten Mengen einige andere an:

Tests Diese Tests liefern Boolesche Ergebnisse. Der Parameter kann eine andere Menge oder eine beliebige Folge sein.

Mit der Methode `disjoint()` können Sie testen, ob zwei Mengen »disjunkt« sind, also keine Elemente gemeinsam haben (mathematisch gesprochen ist die Schnittmenge leer):

```
>>> s.disjoint({'Bravo', 'Hotel', 'Tango'})
False
>>> s.disjoint(['Delta', 'Echo'])
True
```

Die Methoden `issubset()` und `issuperset()` testen, ob die Menge eine Unter- oder Obermenge der als Parameter angegebenen Menge ist:

```
>>> s = set("ABC")
>>> s.issubset("ABCDE")
True
>>> s.issubset("DEF")
False
>>> s.issuperset("AB")
True
```

(Wir nutzen hier den Umstand aus, dass der `set()`-Konstruktor aus einer beliebigen Folge eine Menge macht. Zeichenketten sind Folgen.)



Statt `issubset()` und `issuperset()` können Sie auch die Operatoren `<=` und `>=` benutzen. Allerdings müssen dann beide Operanden Mengen sein:


```
>>> s0 = set("ABC")
>>> s1 = set("BC")
>>> s2 = set("ABCDE")
>>> s1 <= s0
True
>>> s0 >= s2
False
```



Für Mengen gibt es außerdem noch die Operatoren `<` und `>` (die keine Äquivalente als Methoden haben). Mit `s0 < s1` können Sie prüfen, ob `s0` eine »echte Untermenge« von `s1` ist, das heißt, es gilt `s0 <= s1`, aber nicht `s0 == s1`. `>` funktioniert entsprechend für »echte Obermenge«.




Zwei Mengen gelten als gleich, wenn die eine eine Untermenge der anderen ist und gleichzeitig die andere eine Untermenge der einen. (Das ist eine umständliche Methode, auszudrücken, dass sie dieselben Elemente haben müssen.)

 Es ist wichtig zu wissen, dass die Tests auf Untermengen und Gleichheit keine totale Ordnung für alle Mengen induzieren. Für die Mengen $A = \{a_0, a_1, a_2\}$ und $B = \{b_0, b_1\}$ gilt weder $A \subset B$ noch $B \subset A$ noch $A = B$. Aus diesem Grund können Sie Listen von Mengen in Python nicht sortieren.


Schnittmenge und Vereinigung Die Methode `intersection()` liefert eine Menge, die genau diejenigen Elemente enthält, die in der betreffenden Menge und gleichzeitig in allen als Parameter übergebenen Mengen enthalten sind:

```
>>> s0 = set("ABC")
>>> s1 = set("CDE")
>>> s2 = set("BCFG")
>>> s0.intersection(s2)
{'B', 'C'}
>>> s0.intersection(s1, s2)
{'C'}
>>> s0 & s2
{'B', 'C'}
>>> s0 & s1 & s2
{'C'}
```

 `&` ist die »Operatorversion« von `intersection()`. Die Methode akzeptiert außer Mengen auch beliebige Folgen als Parameter, die Operatorversion nicht.

Mit der Methode `union()` können Sie die Vereinigung von Mengen bestimmen, also eine Menge, die alle Elemente der betreffenden Menge und aller als Parameter übergebenen Mengen enthält:

```
>>> s0.union(s1)
{'B', 'A', 'C', 'E', 'D'}
>>> s0.union(s1, s2)
{'B', 'F', 'G', 'A', 'C', 'E', 'D'}
>>> s0 | s1
{'B', 'A', 'C', 'E', 'D'}
```

 Die Operatorversion von `union()` ist `|` (der vertikale Balken/das Pipe-Symbol).

Restmenge Mit der Methode `difference()` (Operatorversion »-«) erhalten Sie eine neue Menge, die alle Elemente der ursprünglichen Menge enthält, die *nicht* in einer der als Parameter angegebenen Mengen enthalten sind:

```
>>> s2.difference("AB")
{'C', 'F', 'G'}
>>> s2.difference(s0, s1)
{'F', 'G'}
>>> s2 - set("AB")
{'C', 'F', 'G'}
```

Symmetrische Differenz Die Methode `symmetric_difference()` (Operatorversion »^«) liefert eine neue Menge, die genau diejenigen Elemente enthält, die entweder in der ursprünglichen Menge oder der als Parameter übergebenen Menge enthalten sind, aber nicht in beiden (die Methode erlaubt nur einen Parameter):

```
>>> s1.symmetric_difference(s2)
{'B', 'E', 'F', 'G', 'D'}
>>> s1 ^ s2
{'B', 'E', 'F', 'G', 'D'}
```



Ausdrücke, die sets und frozensets mischen, liefern Objekte zurück, deren Typ dem des ersten Operanden entspricht:

```
>>> t = frozenset('abc') | set('bcd')
>>> type(t)
<class 'frozenset'>
```

Für set-Objekte gibt es noch ein paar weitere Methoden und Operanden, mit denen Sie die Mengen ändern können (bei frozensets ist das nicht erlaubt):

Methoden und Operanden für sets

Elemente hinzufügen und entfernen Mit `add()` können Sie einzelne Elemente zu einer Menge hinzufügen:

```
>>> s = set("AB")
>>> s.add("C")
>>> s
{'B', 'C', 'A'}
```

Die Methode `remove()` entfernt ein Element aus einer Menge. Wenn Sie versuchen, ein Element aus der Menge zu entfernen, das überhaupt nicht enthalten ist, gibt es einen `KeyError`:

```
>>> s.remove("B")
>>> s
{'C', 'A'}
>>> s.remove("Z")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Z'
```

Die Methode `discard()` entfernt ein Element nur, wenn es tatsächlich existiert; wenn das nicht der Fall ist, passiert nichts:

```
>>> s.discard("C")
>>> s
{'A'}
>>> s.discard("Z")
>>> s
{'A'}
```

Mit der Methode `pop()` wird *irgendein* Element aus der Menge entfernt und als Ergebnis zurückgeliefert. Ist die Menge leer, gibt es einen `KeyError`:

```
>>> s = set("ABC")
>>> while len(s) > 0:
...     print(s.pop())
...     ↵
B
C
A
>>> s.pop()
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
KeyError: 'pop from an empty set'
```

Die Methode `clear()` entfernt *alle* Elemente einer Menge:

```
>>> s = set("ABC")
>>> s.clear()
>>> s
set()
```

Update-Operationen Die Operationen »Schnittmenge«, »Vereinigung«, »Restmenge« und »symmetrische Differenz« haben Versionen, die keine neue Menge liefern, sondern die ursprüngliche Menge aktualisieren. Für die Vereinigungsmenge heißt diese Methode `update()`:

```
>>> s = set("AB")
>>> s.update("BC")
>>> s
{'B', 'C', 'A'}
>>> s.update("AC", "DBF")
>>> s
{'B', 'C', 'F', 'A', 'D'}
```

Der Methode `update()` entspricht der Operator `|=`. Wie üblich muss hier die »rechte Seite« schon eine Menge sein:

```
>>> s = set("AB")
>>> s |= set("BC")
>>> s
{'B', 'C', 'A'}
```

Die Methoden `intersection_update()` (Operator `&=`), `difference_update()` (Operator `-=`) und `symmetric_difference_update()` (Operator `^=`) funktionieren entsprechend.

Übungen



6.10 [1] Überzeugen Sie sich, dass

```
s0.symmetric_difference(s1) == s0.union(s1) - s0.intersection(s1)
```



6.11 [!2] Untersuchen Sie das Systemwörterbuch (typischerweise unter `/usr/share/dict/words` gespeichert – schauen Sie sich im Zweifel um) auf Anagramme, also Wörter, die aus denselben Buchstaben bestehen, aber in einer anderen Reihenfolge, etwa TIGERIN, REINIGT und INTRIGE. Geben Sie alle tatsächlichen Anagramme aus, also Gruppen von zwei oder mehr Wörtern, die aus denselben Buchstaben zusammengesetzt sind.

(*Tipp:* Verwenden Sie ein Dictionary mit den Buchstaben jedes Worts als sortiertes Tupel als Schlüssel. Die korrespondierenden Werte sind Mengen der angetroffenen Wörter. Das heißt, die Beispielwörter TIGERIN, REINIGT und INTRIGE landen alle in der Menge, die der Wert des Dictionary-Elements mit dem Schlüssel ('E', 'G', 'I', 'I', 'N', 'R', 'T') ist.)



6.12 [1] Erweitern Sie das Programm aus der vorigen Aufgabe so, dass die größte Anagramm-Menge gefunden und ausgegeben wird. (Achtung: Mehrere Anagramm-Gruppen könnten gleich groß sein.)

Zusammenfassung

- Dictionaries erlauben das Ablegen beliebiger Daten unter einem Schlüssel. Der Schlüssel muss *hashable* sein.
- Enthält eine Funktionsdefinition einen Parameter der Form `**kwargs`, so bekommt dieser ein Dictionary aller benannten Parameter zugewiesen, die nicht anderweitig in der Funktionsdefinition auftauchen.
- *Dictionary comprehensions* sind eine bequeme Möglichkeit, Dictionaries zu erzeugen.
- Mengen sind ungeordnete Sammlungen verschiedener Objekte. Jedes Objekt taucht in der Menge höchstens einmal auf.
- Als Mengen unterstützt Python veränderbare `sets` und unveränderbare `frozensets`.



7

Objektorientierte Programmierung

Inhalt

7.1	Klassen, Attribute und Methoden	100
7.2	Konstruktoren und Destruktoren	105
7.3	Vererbung	107
7.4	Operatoren überladen	111
7.5	»Duck Typing«	115

Lernziele

- Klassen, Attribute und Methoden definieren können
- Konstruktoren und Destruktoren einsetzen können
- Vererbung in Python verstehen und benutzen können
- Spezielle Methoden und überladene Operatoren verwenden können
- »Duck Typing« verstehen und nutzen können

Vorkenntnisse

- Funktionen in Python (Kapitel 5)
- Dictionaries (Kapitel 6)
- Erfahrung mit anderen Programmiersprachen und mit Objektorientierung ist von Vorteil

7.1 Klassen, Attribute und Methoden

Objektorientierte Programmierung (OOP) ist ein Ansatz zur Softwareentwicklung, der große Systeme leichter versteh- und beherrschbar machen soll. OOP hat sich in den letzten Jahrzehnten als ganz nützlich herausgestellt, auch wenn sie weit hinter den Erwartungen zurückgeblieben ist, die man ursprünglich von ihr hatte. Python unterstützt viele wichtige Merkmale einer »objektorientierten Programmiersprache«.



Die Grundideen hinter der »objektorientierten Programmierung« reichen zurück bis in die 1960er Jahre. Wirklich in Mode kam die Idee allerdings erst während der 1980er.

Der wesentliche Grundgedanke hinter Objektorientierung ist, dass man viele interessante Aufgaben als System miteinander kommunizierender Objekte modellieren kann. Diese Objekte haben jeweils einen (möglicherweise von außen nicht wahrnehmbaren) inneren Zustand und tauschen mit anderen Objekten Nachrichten aus, die dann möglicherweise zu Zustandsänderungen führen. Die Aufstellung eines solchen Modells nennt man »objektorientierte Analyse« (OOA). Auf der Basis eines objektorientierten Modells für ein System kann man eine Lösung für die gewünschte Aufgabe entwerfen, die ebenfalls auf kommunizierenden Objekten beruht (»objektorientiertes Design«, OOD) und diesen Entwurf anschließend in einer geeigneten Programmiersprache implementieren (»objektorientierte Implementierung«).



Diese drei Stufen gelten als Teilaspekte der »objektorientierten Programmierung«, aber es ist zum Beispiel auch ohne weiteres möglich, einen objektorientierten Entwurf in einer Programmiersprache umzusetzen, die selbst keine ausgeprägten OOP-Eigenschaften hat. (Gründe dafür könnten zum Beispiel Portabilitäts- oder Performance-Erwägungen sein.) Allerdings ist sowas oft unbequem und fehleranfällig.

In dieser Schulungsunterlage beschränken wir uns zum allergrößten Teil auf »objektorientierte Implementierung«, indem wir die Eigenschaften von Python beleuchten, die OOP unterstützen. Über OOA und OOD gibt es andere gute Bücher.

Attribute Objekte haben **Attribute**, die ihren inneren Zustand speichern, und **Methoden**,
Methoden mit denen man sie dazu bringen kann, etwas zu tun – die »Nachrichten« aus der
objektorientierten Analyse werden in der Praxis in der Regel durch Methodenauf-
rufe realisiert. In vielen – aber nicht allen – objektorientierten Programmierspra-
chen definieren Sie Objekte allerdings nicht direkt, sondern über **Klassen**: Eine
Klassen Klasse legt einen gewissen Vorrat an Attributen und Methoden fest, und Sie kön-
nen Objekte der betreffenden Klasse erzeugen, die dann über diese Attribute und
Methoden verfügen. Die Attribute und Methoden jedes Objekts sind dann aber
grundsätzlich unabhängig von den Attributen und Methoden anderer Objekte
derselben Klasse.

In Python (3) sieht das ungefähr so aus:

```
class Animal:
    def define(self, species, name):
        self.species = species
        self.name = name
    def describe(self):
        print("{} ist ein {}".format(self.name, self.species))
```

Mit dem Schlüsselwort `class` wird eine Klassendefinition eingeleitet. Der Rest der Kommandofolge besteht normalerweise zum größten Teil aus Funktionsdefinitionen mit `def`, die die Methoden der Klasse implementieren. In unserem Beispiel definieren wir eine Klasse `Animal` mit den Methoden `name()` und `describe()`. Benutzen könnten Sie diese Klasse ungefähr so:

```
>>> lassie = Animal()
>>> lassie.define("Hund", "Lassie")
>>> lassie.describe()
Lassie ist ein Hund
>>> flipper = Animal()
>>> flipper.define("Delfin", "Flipper")
Flipper ist ein Delfin
```

Das ist jetzt nicht gerade der Gipfel der Überraschung, aber wir können trotzdem ein paar interessante Dinge lernen:

- Objekte werden erzeugt, indem Sie den Klassennamen als Funktion aufrufen. Der Klassenname wirkt also als **Konstruktor** für Objekte der Klasse. (Wenn Sie sich genau erinnern, haben wir das schon für Funktionen wie `list()` und `set()` gesagt.)

- Die Syntax für Methodenaufrufe unterscheidet sich von der Syntax für Funktionsaufrufe dadurch, dass das Objekt, dessen Methode gemeint ist, davorsteht. Die Aufrufe

```
>>> lassie.describe()
>>> flipper.describe()
```

beziehen sich also auf die `describe()`-Methoden von zwei verschiedenen Objekten.



Hier gehören `lassie` und `flipper` beide zur Klasse `Animal`, so dass die Methode in beiden Fällen gleich ist. Das muss aber nicht unbedingt so sein, wenn die Objekte zu unterschiedlichen Klassen gehören, denn jede Klasse hat einen unabhängigen Namensraum für Methoden.

- Methoden bekommen das Objekt als impliziten ersten Parameter übergeben. Deswegen ist die Methode `Animal.define()` definiert als

```
class Animal:
    def define(self, species, name):
```

– das Objekt wird als Parameter `self` übergeben.



Wie üblich ist es nicht vorgeschrieben, dass der erste Parameter einer Methode `self` heißen *muss* – es ist nur die allgemein akzeptierte Konvention.

- Zuweisungen der Form

```
self.species = species
```

beziehen sich auf Attribute des Objekts `self`. Das heißt, das Attribut `species` des aktuellen Objekts wird auf den Wert der Variablen `species` gesetzt (denken Sie daran: Python jongliert hier mit Referenzen).



In Python 2.x entspricht der Klassendefinition

```
class Animal:
```

die Definition

```
class Animal(object):
```

Das heißt, `Animal` wird explizit zur Unterklasse der Klasse `object` erklärt (siehe Abschnitt 7.3 für Vererbung). In Python 3.x passiert dasselbe, auch ohne dass Sie das ausdrücklich angeben. Wenn Sie in Python 2.x eine Klasse mit

```
class Animal:
```

definieren, dann wird eine veraltete Art von Klasse mit subtil anderer Semantik definiert. Ersparen Sie sich das besser.

Unser Beispiel hat die Macke, dass Sie `Animal.define()` aufrufen müssen, bevor Sie mit dem Objekt etwas anfangen können: Ein vorzeitiger Aufruf von `Animal.describe()` macht nur Ärger:

```
>>> maja = Animal()
>>> maja.describe()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in describe
AttributeError: 'Animal' object has no attribute 'name'
```

Attribute, die Sie in der Klassendefinition festlegen, – »Klassenattribute« – sind für alle Objekte der Klasse sichtbar und können wie Objektattribute durch ein vorgesehtes »self.« angesprochen werden. Sie werden von gleichnamigen Objektattributen »überdeckt«. Ein gängiger Einsatzzweck ist die Vergabe von Standardwerten für Objektattribute:

```
class Animal:
    species = "Tier"
    name = "Unbekanntes Tier"
    def define(self, species, name):
        self.species = species
        self.name = name
    def describe(self):
        print("{} ist ein {}".format(self.name, self.species))
```

Damit sollten wir keine Fehlermeldung mehr bekommen, wenn wir ein Tier noch nicht genau erklärt haben:


```
>>> maja = Animal()
>>> maja.describe()
Unbekanntes Tier ist ein Tier
>>> maja.define("Biene", "Maja")
Maja ist ein Biene
```

Upps. OK, OK ...



Wenn Sie aufgepasst haben, dann ist Ihnen wahrscheinlich aufgefallen, dass Methoden in einer Klassendefinition auch nur Attribute sind – wir hatten ja gesagt, dass Funktionsdefinitionen im Grunde dasselbe sind wie Zuweisungen, denn in beiden Fällen werden Namen im Namensraum (hier der Klasse) zu einer Referenz auf ein Objekt. Im einen Fall ist es ein Datenobjekt (Zahl, Zeichenkette, Liste, ...), im anderen eben ein Funktionsobjekt.

Im Prinzip können Sie Methoden auch außerhalb einer Klasse definieren:

```
>>> def getSpecies(self):
>>>     return self.species
>>> 
>>> Animal.getSpecies = getSpecies
>>> maja.getSpecies()
'Biene'
```

Das sollten Sie allerdings nicht überstrapazieren – in der Regel ist es besser, die Methoden einer Klasse in der Klassendefinition unterzubringen, weil sie dort am leichtesten wiederzufinden sind.

Python unternimmt keine besonderen Anstrengungen, die Attribute einer Klasse vor Manipulation »von außen« zu schützen¹. Grundsätzlich können Sie auf jeden Fall etwas sagen wie

```
>>> maja.legs = 6
>>> maja.wings = 2
>>> print("Gliedermaßen:", maja.legs+maja.wings)
Gliedermaßen: 8
```

Das heißt aber nicht, dass es in jeden Fall eine grandiose Idee ist (es ist halt nur nicht verboten).



Sie sollten sich vor Augen halten, dass eine Klasse immer eine bestimmte Schnittstelle definiert, und nicht notwendigerweise alle Attribute Bestandteil dieser Schnittstelle sind – einige sind wirklich nur für den internen Gebrauch gedacht. In unserem Beispiel gibt es etwa für die Attribute `name` und `species` eine Methode, mit der sie gesetzt werden können; die direkten Attribute sind also tabu.



Attribute, die nicht Bestandteil der offiziellen dokumentierten Schnittstelle einer Klasse sind, können sich theoretisch von heute auf morgen ändern. Wenn Sie dann zu tief in den Innereien Ihrer Objekte herumgeprokelt haben, stehen Sie möglicherweise dumm da.



Eine gängige Konvention besteht darin, »interne« Namen mit einer Unterstreichung anfangen zu lassen. Wenn Sie also eine Variable mit einem Namen wie `_bla` sehen, dann heißt das »Finger weg«.



Python stört sich normalerweise nicht daran, dass die Attribute `legs` und `wings` aus unserem Beispiel in der Klassendefinition gar nicht vorgekommen sind. Sie können mit Ihren Objekten machen, was Sie wollen, auch völlig neue Attribute definieren (die dann natürlich nur dieses spezielle Objekt hat, nicht irgendwelche anderen Objekte der Klasse). Wenn das für Sie ein Problem bedeutet, können Sie in der Klassendefinition die Namen der Attribute, die erlaubt sein sollen, in das Attribut `__slots__` schreiben (zum Beispiel als Tupel oder Liste). Python meckert dann Attributnamen in Objekten der Klasse an, wenn diese nicht in `__slots__` vorkommen:

```
>>> class C0:
>>>     __slots__ = ('a',)
>>> 
>>> obj = C0()
>>> obj.a = 1
>>> obj.b = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'C0' object has no attribute 'b'
```

»Statische« Methoden sind Methoden, die zwar innerhalb einer Klasse definiert sind, aber nicht auf bestimmte Objekte angewendet werden (man könnte auch »Funktionen« zu ihnen sagen). Statische Methoden können Sie definieren wie im folgenden Beispiel gezeigt:


¹Dies im Gegensatz zu Programmiersprachen wie C++, wo es aufwendige Zugriffsregeln für öffentlich sichtbare, private und für abgeleitete Klassen sichtbare Attribute gibt.

```
class Animal:
    all_animals = {}

    def define(self, species, name):
        self.species = species
        self.name = name
        self.all_animals[self.name] = self

    @staticmethod
    def enumerate():
        for name, animal in Animal.all_animals.items():
            animal.describe()
```


Die etwas ungewöhnlich aussehende Konstruktion `@staticmethod` weist die Methodendefinition in der nächste Zeile als statische Methode aus.

Dekorator  »`@staticmethod`« nennen wir einen Dekorator. Dekoratoren dienen in Python dazu, Funktionen mit besonderen Eigenschaften auszustatten. Man kann mit ihnen sehr interessante Sachen machen, die den Rahmen eines Einführungskurses aber sprengen dürften.

Hier sehen Sie die statische Methode im Einsatz:

```
>>> lassie = Animal()
>>> lassie.define("Hund", "Lassie")
>>> flipper = Animal()
>>> flipper.define("Delfin", "Flipper")
>>> Animal.enumerate()
Flipper ist ein Delfin
Lassie ist ein Hund
```

(Sie müssen `Animal.enumerate()` aufrufen und nicht einfach nur `enumerate()`, denn Python muss die statische Methode ja immer noch in der richtigen Klasse finden.)

 Außer statischen Methoden gibt es in Python noch »Klassenmethoden«, die als ersten Parameter das Klassenobjekt (und nicht das spezifische Objekt aus der Klasse) übergeben bekommen. Sie könnten die Methode `enumerate()` als Klassenmethode definieren wie


```
@classmethod
def enumerate(cls):
    for name, animal in cls.all_animals.items():
        animal.describe()
```

(denken Sie daran, dass `class` als reserviertes Wort nicht als Parametername in Frage kommt) und aufrufen wie

```
>>> Animal.enumerate()
```

Dabei dient das »`Animal.`« nicht mehr nur zum Finden der Methode, sondern bezeichnet auch das Klassenobjekt, das als erster Parameter übergeben wird.

Übungen

 7.1 [!1] Beschreiben Sie den Unterschied zwischen

```
>>> maja.describe()
```


und

```
>>> Animal.describe(maja)
```



7.2 [2] Ändern Sie die Methode `Animal.define()` so, dass Name und Tierart nur gesetzt werden, wenn sie im Aufruf explizit angegeben wurden. Mit anderen Worten, die Aufrufe

```
>>> flipper.define("Delfin")
>>> flipper.define(name="Flipper")
>>> flipper.define(name="Flipper", species="Delfin")
```

sollten alle funktionieren und das gewünschte Ergebnis bewirken.



7.3 [2] Sie haben gesehen, dass Sie nachträglich Methoden in einer Klasse definieren können. Geht das auch für Attribute, in dem Sinne, dass neue Objekte einer Klasse dann diese Attribute mit einem Standardwert enthalten?

7.2 Konstruktoren und Destruktoren

Wie Sie gesehen haben, fungiert der Name einer Klasse als »Konstruktor« für Objekte der Klasse, wenn Sie ihn als Funktion aufrufen. Als Resultat erhalten Sie ein Objekt, das über die in der Klassendefinition angegebenen Attribute (mit den in der Klassendefinition angegebenen Werten als Vorgabe) und Methoden enthält. Im letzten Abschnitt haben Sie dann eine weitere Methode aufgerufen, um Details des Objekts festzulegen, die nicht aus Standardvorgaben hervorgehen:

```
>>> lassie = Animal()
>>> lassie.define("Hund", "Lassie")
```

Das ist natürlich umständlich und außerdem ein bisschen gefährlich – es wäre leicht, den Methodenaufruf zu vergessen und dann mit einem unkonfigurierten Objekt zu operieren. Aus diesem Grund können Sie arrangieren, dass Sie schon im Konstruktoraufruf Parameter übergeben, die dann die Initialisierung des neuen Objekts beeinflussen. Das könnte zum Beispiel so aussehen:

```
>>> lassie = Animal(name="Lassie", species="Hund")
```

Damit das klappt, müssen Sie eine Methode namens `__init__()` definieren, die die gewünschte Parametersignatur hat – natürlich zuzüglich zum ersten Parameter `self`, der auf das neu erzeugte (und mit den Standardwerten aus der Klassendefinition versehene) Objekt verweist:

```
class Animal:
    species = "Tier"
    name = "Unbekanntes Tier"

    def __init__(self, name=None, species=None):
        if name: self.name = name
        if species: self.species = species

    def describe(self):
        print("{} ist ein {}".format(self.name, self.species))
```

Beachten Sie hier, dass wir die Attribute `name` und `species` in der `__init__()`-Methode nur dann setzen, wenn der betreffende Parameter tatsächlich angegeben wurde. Damit funktionieren dann Aufrufe wie

```
>>> lassie = Animal(name="Lassie", species="Collie")
>>> lassie.describe()
Lassie ist ein Collie
>>> flipper = Animal(name="Flipper")
>>> flipper.describe()
Flipper ist ein Tier
>>> anonymous = Animal(species="Elefant")
>>> anonymous.describe()
Unbekanntes Tier ist ein Elefant
```



Natürlich zwingt Sie niemand dazu, eine 1 : 1-Korrespondenz zwischen Parametern der `__init__()`-Methode und Attributen des Objekts aufrechtzuerhalten. In der `__init__()`-Methode dürfen Sie machen, was Sie wollen:

```
def __init__(self, name=None, aclass=None, species=None):
    if name: self.name = name
    if species: self.species = species
    if aclass: self.aclass = aclass
    if aclass == 'Insekt':
        self.legs, self.wings = 6, 2
    elif aclass == 'Vogel':
        self.legs, self.wings = 2, 2
    elif aclass == 'Säugetier':
        self.legs, self.wings = 4, 0
```

Entsprechend dann (siehe hierzu Übung 7.4):

```
>>> lassie = Animal("Lassie", "Säugetier", "Hund")
>>> lassie.describe()
Lassie ist ein Hund (ein Säugetier mit 4 Beinen)
>>> maja = Animal("Maja", "Insekt", "Biene")
Maja ist ein Biene (ein Insekt mit 6 Beinen und 2 Flügeln)
```

Selbstverständlich können Sie in einer `__init__()`-Methode auch mit Parametern wie `*args` und `**kwargs` operieren, um auf beliebige unbenannte oder benannte Parameter im Aufruf zu reagieren.

Objekte können Sie mit dem Kommando `del` löschen. In der Regel klappt das gut, weil Python den größten Teil der Arbeit erledigt. Hin und wieder ist es nötig, erweiterte Aufräumungsarbeiten vorzunehmen. Zum Beispiel könnte es eine Art Objekt geben, in dessen Konstruktor eine temporäre Datei angelegt wird. Dann sollte beim Entfernen des Objekts diese Datei auch wieder entfernt werden.



Das wäre sicher kein schlechtes Beispiel, wenn Sie wüßten, wie man Dateien anlegt und wieder entfernt. Das haben wir aber noch nicht durchgenommen.

Sie können festlegen, was passieren soll, wenn Sie `del` auf ein Objekt anwenden, indem Sie in der betreffenden Klasse eine Methode namens `__del__()` definieren. (Eine solche Methode nennen wir einen **Destruktor**².) Hier ist ein simples Beispiel für eine Klasse, die eine Meldung ausgibt, wenn ein Objekt entfernt wird:

```
class DeleteLogger:
    def __del__(self):
        print("Objekt \{\} wird gelöscht!".format(id(self)))
```

Beobachten Sie die Klasse in Aktion:

²Auch wenn sich das anhört wie ein außerirdischer Mega-Schurke aus einem Superhelden-Film.

```
>>> obj = DeleteLogger()
>>> del obj
Objekt 140683695718976 wird gelöscht!
>>> obj
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'obj' is not defined
```



In Wirklichkeit wird die `__del__()`-Methode eines Objekts nicht zwangsläufig aufgerufen, wenn `del` auf ein Objekt angewendet wird. `del` entfernt ja nur Referenzen auf Objekte und nicht die Objekte selbst – ein Objekt wird vom Python-Interpreter erst tatsächlich entfernt, wenn die letzte Referenz auf das Objekt gelöscht wurde:

```
>>> obj1 = DeleteLogger()
>>> obj2 = obj1                Zweite Referenz auf das Objekt
>>> del obj1                    Nichts passiert
>>> del obj2
Objekt 140683695718976 wird gelöscht!
```

Übungen



7.4 [!2] Definieren Sie die Methode `Animal.describe()` so, dass hinter dem Namen und der Tierart in Klammern die Klasse und die Anzahl der Gliedmaßen angegeben werden. *Für Sonderpunkte:* Unterdrücken Sie die Ausgabe einer Art von Gliedmaßen, wenn deren Anzahl bei dem betreffenden Tier Null ist.



7.5 [2] Erweitern Sie die Klasse `Animal` (inklusive der Methode `Animal.describe()`) so, dass außer Insekten, Vögeln und Säugetieren mit Beinen und Flügeln auch Tiere mit Flossen (etwa Fische) und Tiere ganz ohne Gliedmaßen (etwa Würmer) unterstützt werden. In Extremfällen sollen Beschreibungen wie

```
... (ein Wurm ohne Gliedmaßen)
... (ein Fabeltier mit 6 Beinen, 2 Flügeln und 4 Flossen)
```

ausgegeben werden.

7.3 Vererbung

In Aufgabenstellungen aus der wirklichen Welt zeigt sich oft, dass verschiedene Objekte (oder ihre Klassen) einiges gemeinsam haben, aber es auch kleine Unterschiede gibt. Alle Tiere in unserem Beispiel haben einen Namen und eine Tierart, aber Vögel haben zwei Beine und zwei Flügel und Säugetiere haben vier Beine und gar keine Flügel. Wir könnten uns die Mühe ersparen, im Konstruktor für `Animal`-Objekte die (biologische) »Klasse« des Tiers anzugeben, wenn wir gleich ein Tier der richtigen Klasse erzeugen könnten.

Python unterstützt diese Vorgehensweise – wie viele andere objektorientierte Programmiersprachen auch – über **Vererbung** (engl. *inheritance*). Wir können eine Klasse `Bird` definieren, die von der Klasse `Animal` abgeleitet ist und – statt alle Attribute und Methoden von `Animal` nochmal selbst zu definieren – nur diejenigen Attribute und Methoden hinzufügt, die für Vögel Sinn ergeben und für andere Tiere (möglicherweise) nicht. Etwa so:

Vererbung

```

class Animal:
    def __init__(self, name=None, species=None):
        ...

class Bird(Animal):
    legs = 2
    wings = 2

    def fly(self):
        print("{} fliegt!".format(self.name))

    def sing(self):
        print("{} singt!".format(self.name))

```

Damit funktioniert dann

```

>>> tweety = Bird(name="Tweety", species="Kanarienvogel")
>>> tweety.legs, tweety.wings
(2, 2)
>>> tweety.fly()
Tweety fliegt!
>>> tweety.sing()
Tweety singt!

```

Um eine Klasse von einer anderen Klasse (der »Oberklasse«) abzuleiten, müssen Sie in der Klassendefinition der abgeleiteten Klasse direkt hinter deren Klassennamen die Oberklasse in runden Klammern angeben. Vererbung wird in Python realisiert, indem Namen, die weder im Objekt noch in der Klasse des Objekts gefunden werden, als nächstes in der Oberklasse (falls vorhanden) gesucht werden. In unserem Beispiel wird beim Aufruf des `Bird()`-Konstruktors auf die Methode `Animal.__init__()` zurückgegriffen, da die Klasse `Bird` keine eigene `__init__()`-Methode definiert. Die Methode `fly()` hingegen ist in `Bird` definiert und wird dort gefunden, ohne dass eine Suche in der Oberklasse nötig wird.



Die Oberklasse kann wiederum von einer anderen Klasse abgeleitet sein. In diesem Fall wird eine erfolglose Suche in der Oberklasse in der Oberklasse der Oberklasse fortgeführt und so weiter.



Letzendlich sind (in Python 3) alle Klassen von einer Klasse namens `object` abgeleitet, auch wenn das nicht explizit in der Klassendefinition angegeben wurde. Klassen ohne explizite Oberklasse sind direkte »Unterklassen« von `object`.

Manchmal müssen Sie in einer Methode auf die gleichnamige Methode einer Oberklasse zurückgreifen (die typische Anwendung ist `__init__()`, weil Sie sicherstellen möchten, dass ein neues Objekt ordentlich gemäß der Oberklasse(n) initialisiert ist, bevor Sie ihm in der `__init__()`-Methode Ihrer Klasse den letzten Schliff geben). Dazu können Sie natürlich die Methode der Oberklasse direkt aufrufen:

```

class Bird(Animal):
    def __init__(self, *args, **kwargs):
        Animal.__init__(self, *args, **kwargs)

```

(die Parameter `*args` und `**kwargs` stellen sicher, dass jegliche Parameter aus dem Aufruf des Konstruktors an die Oberklasse durchgereicht werden). Der Haken an diesem Ansatz ist jedoch, dass der Name der Oberklasse (`Animal`) in der `__init__()`-Methode »fest verdrahtet« ist. Das wird zu einem Problem, wenn Sie zwischen `Bird` und `Animal` noch eine weitere Klasse einbauen:

```
class Animal:
    ...
class Vertebrate(Animal):
    ...
class Bird(Vertebrate):
    ...
```

In diesem Moment müssen Sie nämlich auch daran denken, die Methode `Bird.__init__()` anzupassen (und natürlich sämtliche anderen Stellen, wo in der Definition von `Bird` möglicherweise explizit auf `Animal` verwiesen wurde). Gutes Software Engineering ist das nicht gerade. Die geschicktere Methode verwendet die eingebaute Funktion `super()`, die automatisch die »nächstobere« Version einer Methode in der Vererbungshierarchie findet. Das erspart Ihnen die explizite Angabe der Klasse und sieht dann so aus:

```
class Bird(Animal):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
```



In Python 2 müssen Sie den Namen der Klasse, wo die Suche anfangen soll, sowie das Objekt explizit angeben:

```
class Bird(Animal):
    def __init__(self, *args, **kwargs):
        super(Bird, self).__init__(*args, **kwargs)
```

Eine Klasse kann auch von mehreren Oberklassen gleichzeitig abgeleitet sein – dieses Phänomen nennen wir **Mehrfachvererbung** (engl. *multiple inheritance*). Mehrfachvererbung Dazu zählen Sie einfach alle Oberklassen in der Klassendefinition auf:

```
class A:
    def x(self):
        print("Methode x!")

class B:
    def y(self):
        print("Methode y!")

class C(A, B):
    pass
```





Das Python-Kommando `pass` in der Definition von `C` tut nichts – es ist nur Füllstoff, weil in der Klassendefinition hinter dem Doppelpunkt irgendetwas stehen muss, um den Regeln der Programmiersprache Genüge zu tun.


Anschließend können Sie versuchen, ein Objekt der Klasse `C` zu erzeugen:


```
>>> obj = C()
>>> obj.x()
Methode x!
>>> obj.y()
Methode y!
```

Wie Sie sehen, erbt `obj` die Methode `x()` von der Klasse `A` und die Methode `y()` von der Klasse `B`.

 Bei Mehrfachvererbung werden Namen gesucht, indem zuerst (rekursiv nach demselben Verfahren) in der ersten aufgelisteten Oberklasse und ggf. in deren Oberklassen gesucht wird. Danach kommt die nächste Oberklasse in der Liste dran und so weiter, bis der Name gefunden wurde oder die komplette Liste der Oberklassen abgearbeitet ist.

 Für Fachleute: Python verwendet Tiefensuche. Breitensuche – wo zuerst die komplette Liste der Oberklassen angeschaut wird, bevor es in deren Oberklassen weitergeht – wäre auf den ersten Blick vielleicht netter, weil Sie dann nicht unbedingt die komplette Oberklassenhierarchie kennen müssten, aber sie ist anderweitig problematisch: Stellen Sie sich vor, die Klasse C ist abgeleitet von den Klassen A und B. Wenn sowohl A und B denselben Namen – vielleicht x enthalten, müssen Sie, um den Konflikt auflösen zu können, wissen, ob das x von A in der Klasse A selbst oder in einer Oberklasse von A definiert ist. Im ersteren Fall hat es Vorrang gegenüber dem x in B, aber im letzteren hat das x in B Vorrang gegenüber dem x aus der Oberklasse von A. Das wäre deutlich weniger intuitiv.

 In Wirklichkeit ist es noch ein bisschen komplizierter, weil Python mit dem Fall umgehen können muss, dass Klassen in der Vererbungshierarchie auf mehreren Pfaden erreicht werden können (etwa wenn zwei Oberklassen einer Klasse eine gemeinsame Oberklasse haben). Da in Python – wie erwähnt – alle Klassen letztendlich von der Klasse object abgeleitet sind und darum in jedem Fall von Mehrfachvererbung mehrere Wege zu object existieren, ist das sogar unvermeidlich. Deswegen verwendet Python ein dynamisches Verfahren zur Namensauflösung, das sicherstellt, dass jede Oberklasse nur einmal angeschaut wird. Der Endeffekt ist aber derselbe.

 In Python 2 wird für »traditionelle« Klassen (die nicht explizit von object abgeleitet sind) »reine« Tiefensuche verwendet und für »neumodische« Klassen das geschicktere Verfahren aus dem vorigen Absatz.

Mehrfachvererbung klingt auf den ersten Blick wie eine sehr nützliche Sache, aber Sie sollten sie trotzdem mit Vorsicht genießen. Wenn Sie bei der Analyse nicht genau aufpassen, kann es sein, dass Sie per Mehrfachvererbung merkwürdige Chimären produzieren:

```
>>> class FlyingFish(Fish, Bird):
>>>     pass
>>> kurt = FlyingFish(name="Kurt", species="Fliegender Fisch")
>>> kurt.fly()
Kurt fliegt!
>>> kurt.sing()
Kurt singt!
```

Ahem.

Ein fliegender Fisch mag zwar fliegen können (für eine geeignete Definition von »fliegen«), aber er ist halt trotzdem kein Vogel.

Mixin-Klassen

Eine bessere Methode, Mehrfachvererbung einzusetzen, ist über **Mixin-Klassen**, die gezielt Funktionalität beisteuern, ohne gleich eine Vielzahl von Attributen und Methoden einzuschleppen:

```
class FlightCapable:
    def fly(self):
        print("{} fliegt!".format(self.name))

class Insect(Animal, FlightCapable):
    ...

class Bird(Animal, FlightCapable):
    ...
```

```
class FlyingFish(Fish, FlightCapable):
    ...
```

Hier können Sie fast jede Art von Tier »flugfähig« machen, indem Sie die Mixin-Klasse `FlightCapable` in die Liste der Oberklassen aufnehmen. Da die Klasse `FlightCapable` außer der Methode `fly()` nichts enthält, ist das Risiko, unerwünschte Funktionalität verfügbar zu machen, deutlich geringer.

Übungen



7.6 [!2] Schreiben Sie einige Klassen, die verschiedene geometrische Objekte repräsentieren. Insbesondere sollen sie die Möglichkeit anbieten, Umfang und Fläche der Objekte zu bestimmen. Hier ist ein Überblick über die denkbare Funktionalität:

```
>>> c = Circle(r=5)
>>> c.area()
78.53981633974483
>>> c.circumference()
31.41592653589793
>>> r = Rectangle(a=3, b=4)
>>> r.area()
12
>>> s = Square(6)
>>> s.circumference()
24
```

(*Tip*: Definieren Sie eine Klasse `Shape`, von der Sie die anderen Klassen ableiten können.)



7.7 [2] In der vorigen Aufgabe haben wir Ihnen empfohlen, die Klassen für die verschiedenen geometrischen Objekte von einer gemeinsamen Basisklasse `Shape` abzuleiten. Die Klassen `Circle` und `Rectangle` haben aber nicht wirklich viel gemeinsam außer den Methodennamen `area()` und `circumference()` (die *Implementierungen* der Methoden sind ja durchaus verschieden). Welchen Vorteil könnte es haben, trotzdem eine gemeinsame Basisklasse zu verwenden? Finden Sie ein instruktives Beispiel und implementieren Sie es.

7.4 Operatoren überladen

In Abschnitt 7.2 haben Sie gesehen, wie Sie mit speziellen Methoden wie `__init__()` das Verhalten von Objekten konfigurieren können. Es gibt noch verschiedene andere Methoden, die Sie definieren können, um zum Beispiel die Bedeutung von bestimmten eingebauten Funktionen oder Operatoren für Objekte der Klasse festzulegen.

Stellen Sie sich zum Beispiel vor, Sie wollen eine Klasse `Vector` definieren, die zweidimensionale Vektoren (mit x - und y -Koordinaten) repräsentieren soll. Im einfachsten Fall könnte das so aussehen:

```
class Vector:
    def __init__(self, x, y):
        self.x, self.y = x, y
    def add(self, v):
        return Vector(self.x+v.x, self.y+v.y)
```

Mit dieser Definition können Sie Vektor-Objekte definieren und addieren:

```
>>> u = Vector(1, 2)
>>> v = Vector(-1, 0)
>>> t = u.add(v)
>>> t.x, t.y
(0, 2)
```

Darstellung Sie können die Methode `__repr__()` definieren, um die Darstellung des Objekts festzulegen. Solange die Methode nicht definiert ist, bekommen Sie eine standardisierte und nicht besonders hilfreiche Ausgabe, wenn Sie ein Objekt anzeigen lassen:

```
>>> t
<__main__.Vector object at 0x7ff4ac055a58>
```

Mit einer geeigneten Definition von `__repr__()` können Sie eine bequemere Darstellung erreichen:

```
class Vector:
    ...
    def __repr__(self):
        return "Vector({}, {})".format(self.x, self.y)
```

Sie erhalten dann die Ausgabe

```
>>> t
Vector(0, 2)
```



Die Spielregeln sagen, dass `__repr__()` eine Zeichenkette liefern muss. Im Idealfall enthält die Zeichenkette einen Python-Ausdruck mit allen nötigen Informationen, um ein Objekt mit dem entsprechenden Wert neu anzulegen. Sollte das nicht möglich oder sinnvoll sein, dann sollte die Ausgabe aussehen wie `<irgendetwas Nützliches>`. `__repr__()` ist vor allem hilfreich für Debugging.

Die Methode `__str__()` dient dazu, eine »freundliche« Darstellung des Objekts zu erzeugen, die von den eingebauten Funktionen `str()` und `print` und der Zeichenketten-Methode `format()` benutzt wird. Im Gegensatz zu `__repr__()` wird nicht erwartet, dass `__str__()` einen Python-Ausdruck liefert; Sie können irgendetwas festlegen, das Sie nützlich oder dekorativ finden:

```
def __str__(self):
    return "({}/{})".format(self.x, self.y)
```

Damit funktionieren zum Beispiel die folgenden Aufrufe:

```
>>> print(t)
(0/2)
>>> str(t)
(0/2)
>>> "Punkt: {}".format(t)
Punkt: (0/2)
```

Die Klasse enthält eine Methode `add()`, mit der Sie zwei Vektoren addieren können. Das Resultat ist ein neuer Vektor, der die Summe der beiden Vektoren enthält:

```
>>> Vector(-1, 3).add(Vector(2, -2))
Vector(1, 1)
```


Tabelle 7.1: Mathematische Operatoren und die dazugehörigen Methodennamen

Operation	Operator	Methode
Addition	+	<code>__add__()</code>
Subtraktion	-	<code>__sub__()</code>
Multiplikation	*	<code>__mul__()</code>
Division	/	<code>__truediv__()</code>
Division (ganzzahlig)	//	<code>__floordiv__()</code>
Modulo	%	<code>__mod__()</code>
Potenz	**	<code>__pow__()</code>

Das funktioniert, ist syntaktisch aber nicht unbedingt genial. Es wäre wesentlich eleganter, einfach ein »+« benutzen zu können:

```
>>> Vector(-1, 3) + Vector(2, -2)
Vector(1, 1)
```

Dafür müssen Sie in der Klassendefinition einfach eine Methode namens `__add__()` definieren. Der erste Parameter ist (wie üblich) `self`, der zweite ist das hinzuzugeworfene Objekt (ein anderer Vektor). Die Methode liefert ein neues `Vector`-Objekt. Tatsächlich funktioniert die schon definierte `add()`-Methode dafür sehr gut:

```
class Vector:
    ...
    __add__ = add
```



Sie können alle mathematischen Operatoren auf diese Weise »überladen«. (Die wichtigsten und die dazugehörigen Methodennamen stehen in Tabelle 7.1.)

Eine andere Operation, die für Vektoren nützlich ist, ist das »Skalieren« eines Vektors um einen konstanten (skalaren) Faktor. Eine entsprechende Methode könnte aussehen wie

```
class Vector:
    ...
    def scale(self, f):
        return Vector(f*self.x, f*self.y)
```

Damit funktioniert der Aufruf

```
>>> Vector(1, 2).scale(2)
Vector(2, 4)
```

Auch hier wäre es wieder nützlich, diese Operation als Multiplikation des Vektors mit dem Skalar hinschreiben zu können. Der Name der Methode, die Sie dafür definieren müssen, heißt `__mul__()`:

```
>>> Vector.__mul__ = Vector.scale
>>> Vector(1, 2) * 2
Vector(2, 4)
```

Die Sache hat allerdings einen Haken: Die Multiplikation funktioniert so nur, wenn der Vektor links vom »*« steht und der skalare Faktor rechts. Damit Sie auch

```
>>> 2 * Vector(1, 2)
Vector(2, 4)
```

schreiben können, müssen Sie außerdem die Methode `__rmul__()` definieren (die allerdings dasselbe sein kann wie `__mul__()`).



Das »r« in »`__rmul__`« steht für »reflektiert«. Jede Methode, die mit einem mathematischen Operator korrespondiert, hat eine »r-Version«, die aufgerufen wird, wenn der linke Operand die gewünschte Operation nicht unterstützt, aber der rechte. (Die Methode wird dann trotzdem mit dem rechten Operand als erstem und dem linken Operand als zweitem Parameter aufgerufen – deswegen können `__mul__()` und `__rmul__()` in unserem Beispiel identisch sein.)

Prinzipiell spricht nichts dagegen, dass ein überladener Operator mit verschiedenen Datentypen zurechtkommen kann. Betrachten Sie die folgende Implementierung der Vektor-Addition:

```
class Vector:
    ...
    def __add__(self, other):
        if isinstance(other, Vector):
            return Vector(self.x+other.x, self.y+other.y)
        elif isinstance(other, int) or isinstance(other, float):
            return Vector(self.x+other, self.y+other)
        return NotImplemented
```

Diese Version unterstützt die bisherige Addition zweier Vektoren:

```
>>> Vector(1, 2) + Vector(-2, -1)
Vector(-1, 1)
```

Außerdem können Sie (etwas unmathematisch) eine Konstante zu beiden Komponenten des Vektors addieren:

```
>>> Vector(1, 2) + 3
Vector(4, 5)
```



Mit der eingebauten Funktion `isinstance()` können Sie testen, ob ein Python-Objekt zu einer bestimmten Klasse gehört. Genauer gesagt ist `isinstance(foo, bar)` logisch »wahr«, wenn `foo` entweder ein Objekt der Klasse `bar` oder aber ein Objekt einer von `bar` abgeleiteten Klasse ist.



Eine Methode kann `NotImplemented` zurückgeben, wenn sie sich nicht für eine bestimmte Kombination von Parametern interessiert. Wenn zum Beispiel die `__add__()`-Methode für einen bestimmten zweiten Parameter `NotImplemented` liefert, kann das für Python ein Signal sein, die Methode `__radd__()` des zweiten Parameters auszuprobieren (falls sie existiert).

Auch hier kann `__add__()` als `__radd__()` fungieren, damit

```
>>> 1 + Vector(2, 2)
Vector(3, 3)
```

funktioniert.

Neben den »r-Methoden« gibt es auch noch »i-Methoden«, die sich um die »kombinierten« Zuweisungsoperatoren `+=`, `-=` usw. kümmern. Diese Methoden sollten `self` modifizieren und das Ergebnis zurückgeben (`self` wäre dafür naheliegend, aber nicht zwingend). Hier ist eine Implementierung für Vektor-Addition:

```
class Vector:
    ...
    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        return self
```

Damit funktioniert dann

```
>>> u = Vector(1, 2)
>>> u += Vector(-2, 1)
>>> u
Vector(-1, 3)
```

Übungen



7.8 [!2] Definieren Sie eine Klasse `FInt`, die Arithmetik »modulo n « unterstützt. Das heißt, die Grundrechenarten Addition, Subtraktion, Multiplikation und Division sollen Ergebnisse »modulo n « liefern:

```
>>> FInt.setN(7)                                     Alle Berechnungen sind modulo 7
>>> s = FInt(5)
>>> s + FInt(4)
FInt(2)
>>> FInt(4) * FInt(6)
FInt(3)
>>> FInt2 - FInt5
FInt(4)
```

7.5 »Duck Typing«

Viele Operationen in Python setzen voraus, dass ihre Operanden bestimmte Eigenschaften haben. Zum Beispiel erwartet das `for`-Kommando, dass hinter dem in ein »iterierbares« (engl. *iterable*) Objekt steht. Dazu gehören zum Beispiel die Python-Folgen wie Tupel, Listen und Zeichenketten, aber auch Dictionaries oder Mengen. Viele objektorientierte Programmiersprachen würden hier voraussetzen, dass alle diese Klassen von einer gemeinsamen Oberklasse `Iterable` (oder so ähnlich) abgeleitet sind. Python verfolgt dagegen den Ansatz, dass Objekte genau dann in einem bestimmten Kontext auftauchen können, wenn sie die Operationen unterstützen, die für diesen Kontext nötig sind. Abstammung von einer bestimmten Klasse ist nicht erforderlich.

Diese Philosophie wird auch als *duck typing* bezeichnet, gemäß der taxonomischen Beobachtung, dass, wenn etwas watschelt wie eine Ente und quakt wie eine Ente, es auch eine Ente ist.

Insbesondere können Sie Klassen definieren, deren Objekte sich benehmen wie Folgen, Dictionaries und andere eingebaute Datentypen. Zum Beispiel können Sie dafür sorgen, dass ein Objekt die Index-Syntax »`object[key]`« unterstützt, indem Sie die Methode `__getitem__(self, key)` implementieren.



Die Python-Sprachdefinition enthält eine komplette Liste der verschiedenen Methoden, die Sie definieren können, um das Verhalten von Objekten zu beeinflussen.

Als einfaches Beispiel definieren wir eine Klasse `PowersOfTwo`, die für einen gegebenen maximalen Exponenten n die Zweierpotenzen $2^0, 2^1, \dots, 2^{n-1}$ zur Verfügung stellt. Diese Klasse soll sich benehmen wie eine unveränderbare Folge (vulgo ein

Tupel), das heißt, sie soll (lesenden) Elementzugriff mit eckigen Klammern über Indizes und Slices, die Funktion `len()` und Iteration zulassen. Allerdings wollen wir es uns ersparen, eine explizite Liste der Zweierpotenzen zu bestimmen, sondern die Werte lieber von Fall zu Fall berechnen.

Damit die Klasse als unveränderbare Folge funktioniert, muss sie das »Folgenprotokoll« (*sequence protocol*) implementieren. Konkret heißt das, sie muss eine Methode `__getitem__()` haben, die den Zugriff auf einzelne Elemente über einen ganzzahligen Index zulässt, sowie eine Methode `__len__()` haben, die die klassenspezifische Unterstützung für die eingebaute Funktion `len()` liefert. Über diese beiden Methoden werden die wesentlichen Eigenschaften einer unveränderbaren Folge realisiert.

Im Konstruktor merken wir uns den maximalen Exponenten n , den unser Objekt unterstützen soll. (Zwingend nötig wäre das nicht, aber wir möchten, dass eine Iteration über die Zweierpotenzen irgendwann aufhört und nicht bis in die Unendlichkeit fortgesetzt wird.) Also:

```
def __init__(self, n):
    self.n = n
```

Die Methode `__len__()` liefert einfach den im Konstruktor angegebenen maximalen Exponenten zurück. Wie in Python üblich ist das der Exponent, der gerade *nicht mehr* erreicht wird:

```
def __len__(self):
    return self.n
```

Die Methode `__getitem__()` dagegen ist etwas interessanter:

```
def __getitem__(self, index):
    if isinstance(index, int):
        if 0 <= index < self.n: return 2**index
        raise IndexError
    raise TypeError
```

Dies ist einer der (seltenen) Fälle, wo wir uns nicht auf *duck typing* verlassen können, sondern explizit prüfen, dass der angegebene Index eine Ganzzahl ist. Ist das nicht der Fall, lösen wir eine `TypeError`-Exception aus. Bei einem ganzzahligen Index überprüfen wir den Wertebereich und lösen eine `IndexError`-Exception aus, falls der Index außerhalb der gültigen Grenzen liegt.



Die Exceptions gehören zum Folgenprotokoll. Insbesondere der `IndexError` ist wichtig, weil er dafür sorgt, dass eine Iteration beendet wird, wenn das Ende der Folge erreicht ist.

Mit diesen Methoden wird der größte Teil der gewünschten Eigenschaften der Klasse abgedeckt:

```
>>> p2 = PowersOfTwo(10)
>>> len(p2)
10
>>> p2[6]
64
>>> list(p2)
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

Das Einzige, was noch nicht funktioniert, sind Slices:

```
>>> p2[3:6]
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
File "revlist.py", line 40, in __getitem__
    raise TypeError
TypeError
```

Um Unterstützung für Slices realisieren zu können, müssen Sie wissen, dass Python bei einem Zugriff wie `p2[3:6]` die `__getitem__()`-Methode mit einem »Slice-Objekt« aufruft, das die gewünschte Slice beschreibt. Slice-Objekte unterstützen eine Methode `indices()`, die die numerischen Werte liefert, die Sie mit `range()` verwenden müssen, um die tatsächlichen Indizes zu erzeugen. (Schon verwirrt?) In Python-Code sieht das dann so aus:

```
def __getitem__(self, index):
    if isinstance(index, int):
        if 0 <= index < self.n: return 2**index
        raise IndexError
    elif isinstance(index, slice):
        start, stop, step = index.indices(self.n)
        return [2**i for i in range(start, stop, step)]
    raise TypeError
```

Damit funktionieren dann auch Slice-Zugriffe:

```
>>> p2[3:6]
[8, 16, 32]
```

Das Folgenprotokoll für unveränderbare Folgen ist das einfachste Protokoll für komplexe Python-Datenstrukturen. Für Klassen, die sich verhalten sollen wie veränderbare Folgen (Listen) oder Abbildungstypen (Dictionaries), müssen Sie noch weitere Methoden implementieren. Die Details stehen in der *Python Language Reference*.

Übungen



7.9 [!2] Die *Fibonacci-Zahlen* F_k sind definiert als $F_0 = 0, F_1 = 1, F_k = F_{k-1} + F_{k-2}$ für $k > 1$ (mit anderen Worten, jede Zahl ist die Summe der beiden vorhergehenden). Definieren Sie in Analogie zur Klasse `PowersOfTwo` eine Klasse `Fibonacci`, die die ersten $n - 1$ Fibonacci-Zahlen als unveränderbare Folge zur Verfügung stellt.



A

Musterlösungen

Dieser Anhang enthält Musterlösungen für ausgewählte Aufgaben.

2.1 Letzteres ist eine genau dargestellte (lange) Ganzzahl, ersteres eine – intern angenähert dargestellte – Gleitkommazahl.

3.1 Versuchen Sie etwas wie

```
i = 10
while i >= 0:
    print i
    i -= 1
```

3.4 Für die Zahl 97 sind 118 Schritte nötig. Wenn wir Zahlen bis 1000 betrachten, beträgt das Maximum 178 Schritte (für die Zahl 871). Wenn Sie Zahlen bis 10.000.000 untersuchen, kommen Sie mit 8.400.511 auf ein Maximum von 685 Schritten.

4.1 Eine Möglichkeit wäre

```
liste[len(liste):len(liste)] = [element]
```

4.2 Versuchen Sie mal »del liste[1::2]«.

4.6 Diese Lösung verlangt nach einem (einfachen) Zustandsautomaten:

```
#!/usr/bin/python3

s = "Donaudampfschiffahrtsgesellschaft"
vokale = "aeiou"
vvk = 0
state = 0
for c in s:
    if state == 0 and c in vokale:
        state = 1
    if state == 1 and c not in vokale:
        vvk += 1
        state = 0
print("{} Vokale standen vor einem Konsonant".format(vvk))
```

Welcher Ansatz ist Ihnen sympathischer?

4.7 Zum Beispiel:

```
>>> [i for i in range(21) if i % 3 != 0]
```

4.8 Probieren Sie etwas wie

```
#!/usr/bin/python3
for i in range(1, 11):
    print("".join(["{:4d}".format(i*j) for j in range(1, 11)]))
```

Es geht auch mit einer einzigen *list comprehension* – es ist sehr simpel, die Zahlen zu berechnen, aber der Haken ist der Zeilenvorschub nach jedem Vielfachen von 10. Sie könnten da natürlich mit etwas wie

```
"".join(["{:4d}{}".format(i*j, '\n' if j==10 else '')])
```

tricksen, aber den Lesern Ihres Codes tun Sie damit keinen großen Gefallen. Unter dem Strich ist die ursprüngliche Methode mit den zwei verschachtelten for-Schleifen vermutlich am elegantesten.

4.9 Hier gibt es im Grunde zwei verschiedene Ansätze.

```
>>> x = (1, 2, 3)
>>> y = (4, 5, 6)
>>> [x[i]+y[i] for i in range(len(x))]
[5, 7, 9]
>>> [xi+yi for xi, yi in zip(x, y)]
[5, 7, 9]
```

Der erste Ansatz verwendet einen numerischen Index, um die jeweiligen Elemente zu identifizieren. Der zweite verwendet die Python-Funktion `zip()`, um eine Folge aus Paaren der jeweils zu addierenden Komponenten zu erzeugen. Jedes Paar wird an die Variablen `xi` und `yi` zugewiesen und kann dann weiterverarbeitet werden.

6.1 Dies ist ein Einsatz für unsere alte Freundin, die Funktion `zip()`:

```
>>> d = dict(zip(a, b))
```

6.2 Ersetzen Sie

```
print(cipher)
```

durch

```
cipher = cipher.replace(' ', '')
print("".join([cipher[i:i+5] for i in range(0, len(cipher), 5)]))
```

(Natürlich können Sie auch von Anfang an statt dem Leerzeichen in `encode.get()` eine leere Zeichenkette verwenden.)

6.3 Das ist kein Problem, das Zauberwort heißt *list comprehension*:

```
cipher = "".join([encode.get(c, '')
                  for c in " ".join(sys.argv[1:]).upper()])
```


6.4 Zum Entschlüsseln müssen Sie einfach das Dictionary decode statt dem Dictionary encode verwenden. Wenn Sie ganz clever sind, dann machen Sie die Auswahl vom Programmnamen abhängig:

```
if sys.argv[0].startswith('decode'):
    subst = decode
else:
    subst = encode
```

und später dann

```
cipher += subst.get(c, '')
```

6.6 Eine Möglichkeit:

```
graph = {
    'A': { 'B': 2, 'D': 2 },
    'B': { 'D': 5, 'C': 1 },
    'C': { 'A': 7, 'E': 1 },
    'D': { 'A': 2, 'B': 5, 'E': 2 },
    'E': { 'C': 1 },
}
```

(Beachten Sie, dass Kanten mit einer Pfeilspitze einmal auftauchen und Kanten mit zwei Pfeilspitzen zweimal.)

6.8 Etwas wie

```
{ c: s.count(c) for c in s if c not in "AEIOU" }
```

sollte reichen (beachten Sie das not).

6.11 Der interessante Teil des Programms ist etwas wie

```
import sys

anagrams = {}
while True:
    line = sys.stdin.readline()
    if line == '': break
    line = line.strip().upper()
    key = tuple(sorted(line))
    if key not in anagrams:
        anagrams[key] = set()
    anagrams[key].add(line)
```

7.1 Letzteres ist mehr Tipp-Arbeit und nagelt Sie auf die Methode describe() der Klasse Animal fest, was möglicherweise ein Problem wird, wenn Vererbung ins Spiel kommt (siehe Abschnitt 7.3). Ansonsten ist zwischen den beiden Aufrufen überhaupt kein Unterschied.

7.2 Versuchen Sie etwas wie

```
def define(self, species=None, name=None):
    if species: self.species = species
    if name: self.name = name
```

7.3 Natürlich. Es zeigt sich, dass sogar existierende Objekte rückwirkend das Attribut mit dem angegebenen Standardwert dazubekommen. (Erinnern Sie sich daran, dass die Namen von Attributen zuerst im Namensraum des Objekts und dann im Namensraum der Klasse gesucht werden. Namen, die Sie nachträglich in den Namensraum der Klasse hineinpraktizieren, werden selbstverständlich auch dann gefunden, wenn die Suche bei einem existierenden Objekt losgeht. Betrachten Sie das Beispiel:

```
>>> class C:
>>>     a = 1
>>>     ↵
>>> obj1 = C()
>>> obj1.a
1
>>> obj1.b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'C' object has no attribute 'b'
>>> C.b = 2
>>> obj2 = C()
>>> obj2.b
2
>>> obj1.b
2
```

Jetzt dann doch.

7.4 Probieren Sie etwas wie

```
def describe(self):
    limbs = []
    for count, name in ((self.legs, "Beinen"),
                       (self.wings, "Flügel")):
        if count > 0:
            limbs.append("{} {}".format(count, name))
    limbs = " und ".join(limbs)
    print("{} ist ein {} (ein {} mit {})".format(self.name,
                                                self.species,
                                                self.aclass,
                                                limbs))
```

7.7 Es könnte Eigenschaften von geometrischen Objekten geben, die nichts mit der Geometrie im engeren Sinne zu tun haben, etwa die Farbe. Die Basisklasse Shape könnte ein Attribut colour sowie Methoden getColour() and setColour() definieren, die die abgeleiteten Klassen dann erben können. Der Konstruktor könnte einen Parameter colour unterstützen.



Index

Dieser Index verweist auf die wichtigsten Stichwörter in der Schulungsunterlage. Besonders wichtige Stellen für die einzelnen Stichwörter sind durch **fette** Seitenzahlen gekennzeichnet. Sortiert wird nur nach den Buchstaben im Indexeintrag; „~/ .bashrc“ wird also unter „B“ eingeordnet.

ABC, 12
Ackermann-Funktion, 75
Attribute, **100**

Boole, George, 35
Byte-Strings, **29**
Bytecode, 14

Collatz, Lothar, 41

Definitionen, 9
Destruktor, **106**

Klassen, **100**
Kommandofolge, **34**
Konstruktor, **101**, 105

LEGB-Regel, 78
Logische Operatoren, **36**

Mehrfachvererbung, **109**
Methoden, **100**
Mixin-Klassen, **110**

Oberklasse, 108

PATH (Umgebungsvariable), 17
Péter, Rózsa, 75

Schlüsselwörter, **28**

Umgebungsvariable 0 PATH, 17

van Rossum, Guido, 12, 36
Vererbung, **107**
Vergleichsoperatoren, 35

zusammengesetztes Kommando, **34**