# Einführung

## 1 Agenda 2011

- Vorstellung der Seminarteilnehmer
- Anmerkungen zum Seminarablauf
- Semesterapparat
- Anmerkungen zu Leistungsnachweisen
- Hinweise zur Installation von Python und Zusatzbibliotheken
- Grundsätzliche Anmerkungen zum Programmieren
- Übungsaufgaben zur nächsten Sitzung

## 2 Seminarablauf

### 2.1 Seminarstruktur

Die grundlegende Struktur der Seminare stelle ich mir wie folgt vor:

- **Wiederholung:** Besprechung von Übungsaufgaben, Beantwortung von Fragen, Wiederholung wichtiger Fragestellungen
- **Einführung in neue Themen:** Besprechung der für die jeweiligen Sitzungen vorgesehenen Themen, Einführung neuer Konzepte, Vorstellung neuer Lösungsstrategien
- **Übung:** Eigenständiges Aufbereiten der neuen Themen im Rahmen von Übungsaufgaben, die individuell oder allgemein betreut werden

### 2.2 Literatur

Zu jeder Sitzung werden Texte (im Web und als PDF zum Download) zur Verfügung gestellt, die als Einstieg in die neuen Themen der einzelnen Seminarsitzungen dienen sollen. Ich erwarte, dass jeder Seminarteilnehmer einem der beiden Literaturangebote (Webseite oder PDF) nachgeht, und sich damit auf die jeweilige Seminarsitzung vorbereitet. Um dies zu gewährleisten, und auch, um einer einseitigen Wahl der Quellen seitens der Seminarteilnehmer nachzugehen, werden in dieser Sitzung alle Seminarteilnehmer zwei Gruppen zugeteilt, die sich beim Lesen der Internet- und der Buchquellen abwechseln. Dadurch können wir im Seminar von allen Aspekten profitieren, ohne dass doppelte Arbeit geleisten werden muss.

### 2.3 Übungsaufgaben

Zu jeder Sitzung werden Übungsaufgaben verteilt, welche meist im Verfassen kleiner Programme oder Programmblöcke bestehen werden. Diese Übungsaufgaben sollen von jedem Seminarteilnehmer bearbeitet werden. Um zu gewährleisten, dass auch jeder Teilnehmer “mitkommt” und in der Lage ist, die Aufgaben zu bewältigen (damit es im Abschlusstest keine bösen Überraschungen gibt), werden im

Abstand von jeweils drei Sitzungen die Aufgaben nicht nur im Rahmen der Seminarsitzungen besprochen, sondern auch von mir eingesehen, korrigiert und kommentiert. Die entsprechenden Sitzungen, in denen diese Aufgaben besprochen werden, sind im Semesterplan grau markiert. Um mir genügend Gelegenheit zu geben, die Korrekturen durchzuführen, müssen die Lösungen zu den Aufgaben mir bis zum Sonntag VOR der jeweiligen "grauen" Sitzung vorliegen. Zur Bearbeitung und Übermittlung der Aufgaben sehe ich folgendes Schema vor:

- Bis zum Sonntag vor einer "grauen" Sitzung werden die Aufgaben von jedem Seminarteilnehmer eigenhändig und digital bearbeitet:
  - Wenn es sich um reine Programmieraufgaben handelt, werden diese in einer Skriptdatei verfasst (eine Textdatei mit der Endung ".py").
  - Im Falle von Textaufgaben, bei denen Fragen beantwortet werden, sollten diese in einem gängigen Textsatzprogramm (Latex, Word, OpenOffice) geschrieben, und in PDF umgewandelt werden.
- Die fertiggestellten Aufgaben werden per Email an mich gesandt (listm@phil.uni-duesseldorf.de), und zwar mit der Betreffszeile "Aufgabe: NR", wobei NR die Nummer der Aufgabe ist. Dies erleichtert mir das Sortieren der Daten. IN der Email sollte die Betreffszeile wiederholt werden, um zu verhindern, dass die Email als Email ohne Betreffszeile in meinem Spamordner landet.
- Die Aufgaben werden von mir eingesehen, korrigiert und gegebenenfalls kommentiert, und in der entsprechenden "grauen" Sitzung nochmals besprochen.
- Jeder Seminarteilnehmer bekommt von mir zwei Wildcards geschenkt, (eine zu Ostern und eine zu Pfingsten). Diese Wildcards können anstelle von Aufgaben eingereicht werden. In der jeweiligen Email wird dann anstelle von "Aufgabe: NR" der Betreff "Aufgabe: WILDCARD" geschrieben.

## 3 Semesterapparat

Um das Uploaden von Daten zu erleichtern, habe ich einen Semesterapparat auf einer passwortgeschützten Domain eingerichtet. Die Adresse lautet: `http://lingulist.de/python/`. Das Passwort wird im Seminar genau JETZT verkündet. Die Texte können dort heruntergeladen werden. Übungsaufgaben und Musterlösungen ebenfalls. Ferner werde ich die Handzettel zu den einzelnen Sitzungen dort zur Verfügung stellen. Bei Fragen oder Problemen, die evt. beim Öffnen der Dateien auftreten könnten, bitte ich, mich möglichst früh zu kontaktieren. In diesem Zusammenhang möchte ich noch mal darauf hinweisen, dass kaputte Drucker oder PDFs, die sich eine halbe Stunde vor der Sitzung komischerweise nicht öffnen lassen, nicht als Gründe für das Nichtlesen von Texten akzeptiert werden. Für dieses Seminar wurde der Umfang der zu lesenden Literatur bewusst reduziert, da die programmiertechnische Praxis im Vordergrund stehen soll. Umso wichtiger wird daher das sein, was in den Sitzungen besprochen wird. Das Lesen der Literatur ist dennoch zentral für eine Teilnahme am Seminar.

## 4 Leistungsnachweise

- **Beteiligunsnachweise** werden im Rahmen eines Kurztests in der letzten Seminarsitzung erbracht (Dauer: ca. 45 Minuten). Selbstverständlich ist die Teilnahme an diesem Kurztest an eine Teilnahme am Seminar geknüpft, die über bloße Anwesenheit hinausgeht. Grundsätzlich wird erwartet, dass die im Seminarplan verzeichnete und im Semesterapparat zur Verfügung gestellte Literatur auch gelesen wird. Ferner sollten die zu jeder Sitzung gereichten Übungsaufgaben

bearbeitet werden. Ferner sollte jede Seminarsitzung nachbereitet werden, da die Fragen im Abschlusstest auf die in diesen Sitzungen erarbeiteten Erkenntnisse abgestimmt sein werden.

- **Leistungsnachweise** können im Rahmen der üblichen Möglichkeiten (mündliche Prüfung oder Hausarbeit) erworben werden. Hier bitte ich, mit mir einen Sprechstundentermin zu vereinbaren, damit Thema und Termin abgesprochen werden können.
  - Für mündliche Prüfungen wird ein dreiseitiges Exposé (inklusive Literaturangaben) für das Prüfungsthema erwartet, das spätestens acht Tage vor dem Prüfungstermin bei mir einzureichen ist, damit es spätestens eine Woche vor dem Prüfungstermin in der Sprechstunde besprochen werden kann. Der Inhalt dieses Exposés kann in der Sprechstunde zuvor abgestimmt werden.
  - In Bezug auf Hausarbeiten bitte ich, die entsprechenden Hinweise, die auf der Homepage der Allgemeinen Sprachwissenschaft zur Verfügung gestellt werden, zu beachten. Insbesondere beim Zitieren von Literatur, sowie den übrigen (bekannten) Formalia, wie bspw. Gliederung, Schreibstil, sollte sehr sorgfältig vorgegangen werden. In Fällen von Unsicherheit (wenn man beispielsweise nicht weiß, wie eine Internetquelle zitiert werden soll, o.ä.) bitte ich **um Rücksprache**, da Formfehler selbstverständlich in die Note mit einfließen und so relativ einfach vermieden werden können.

## 5 Installation von Python und Zusatzbibliotheken

> Eine **Programmbibliothek** bezeichnet in der Programmierung eine Sammlung von Programmfunktionen für zusammengehörende Aufgaben. Bibliotheken sind im Unterschied zu Programmen keine eigenständig lauffähigen Einheiten, sondern Hilfsmodule, die Programmen zur Verfügung gestellt werden.(aus: Wikipedia: Programmbibliothek)

Neben der generellen Installation von Python (Version 2.6, erhältlich unter `http://www.python.org/download/releases/2.6/`) werden wir im Laufe des Seminars auch mit sogenannten Zusatz- oder Programmbibliotheken arbeiten. Das sind Erweiterungen der Standardbibliothek, welche das Bearbeiten verschiedener Aufgaben ungemein erleichtern. Im Laufe des Seminars werden wir den folgenden Zusatzbibliotheken begegnen:

- **NumPy:** Numeric Python (`http://numpy.org`) ist eine große Bibliothek, die eine Vielzahl mathematischer Funktionen zur Verfügung stellt.
- **SciPy:** Scientific Python (`http://scipy.org`) ist die beste und umfangreichste Bibliothek für Python und einer der Gründe, warum es sich lohn, Python zu lernen. SciPy bietet eine Vielzahl von Funktionen, die das tägliche Leben des Wissenschaftlers (auch des Sprachwissenschaftlers) erleichtern.
- **NLTK:** Das Natural Language Toolkit (`http://nltk.org`) ist eine speziell für linguistische Anwendungen geschaffene Pythonbibliothek, die eine Vielzahl von Funktionen für allgemeine computer- und korpuslinguistische Aufgaben zur Verfügung stellt. Im Gegensatz zu SciPy und NumPy kann NLTK jedoch zuweilen ein wenig langsam und umständlich anmuten, und die Funktionen sind nicht sehr schnell in der Durchführung.

# 6 Warum Programmieren? Warum Python?

## 6.1 Warum Programmieren?

Programmieren zu können ist immer dann sinnvoll, wenn man in seinem Beruf oder den Studien, denen man nachgeht, häufig wiederholte, redundante Operationen ausführen muss, die sich ebensogut automatisch erledigen lassen würden. Wenn man zum Beispiel ein psycholinguistisches Experiment mit 200 Stimuli durchführen will, und wissen möchte, wie häufig die Wörter im Durchschnitt vorkommen, dann kann man zur Webseite `http://wortschatz.informatik.uni-leipzig.de/` gehen, wo die Worthäufigkeit für eine Vielzahl von Wörtern verzeichnet ist, und jedes der Wörter einzeln in das Suchfenster eingeben, um dessen Häufigkeit zu ermitteln. Dies wird dann mindestens zwei Stunden stumpfer Arbeit zur Folge haben, während der man jedes einzelnen Wort kopiert, die Webseite aufruft, die Häufigkeit kopiert und in eine Tabelle einträgt. Spätestens beim dritten Experiment, das man durchführt, wird man diese Arbeit hassen und sich einen Hiwi wünschen. Alternativ kann man auch einfach programmieren: Die Leipziger Wortschatzprojekt bietet eine Zusatzbibliothek für Python an (`http://pypi.python.org/pypi/libleipzig`), mit deren Hilfe man ganz schnell ein kleines Programm schreiben kann, das einem für eine beliebige Liste von Wörtern in Sekundenschnelle alle Frequenzen (und noch viel mehr Informationen, wenn man will) aus dem Internet herunterlädt, und — wenn man will, auch noch den Durchschnitt und die Standardabweichung aller Frequenzen errechnet. Die Eingabe ist dabei denkbar einfach. Um zum Beispiel die absolute Frequenz des Wortes “Python” zu erhalten, muss man auf der Kommandozeile einfach nur den folgenden Befehl eingeben:

```
>>> Frequencies(”Python”)
```

Als Antwort erhält man dann das Folgende:

```
[( Anzahl: u’129’,  Frequenzklasse: u’17’)]
```

Programmieren kann redundante Arbeiten also ungemein erleichtern.

## 6.2 Warum Python?

Bassi (2010:10) nennt die folgenden Argumente, die für das Erlernen der Programmiersprache Python sprechen:

- Readability: When we talk about readability, we refer as much to the original programmer as any other person interested in understanding the code. It is not an uncommon occurrence for someone to write some code then return to it a month later and find it difficult to understand. Sometimes Python is called a “human readable language.”
- Built-in features: Python comes with “Batteries included.” It has a rich and versatile standard library which is immediately available, without the user having to download separate packages. With Python you can, with few lines, read an XML file, extract files from a zip archive, parse and generate email messages, handle files, read data sent from a Web browser to a Web server, open a URL as if were a file, and many more possibilities.

- Availability of third party modules: 2/3D plotting, PDF generation, bioinformatics analysis, animation, game development, interface with popular databases, and application software are only a handful of ex- amples of modules that can be installed to extend Python functionality.
- High level built-in data structures: Dictionaries, sets, lists, and tuples help to model real world data.
- Multiparadigm: Python can be used as a "classic" procedural language or as "modern" object oriented programming (OOP) language. Most programmers start writing code in a procedural way and when they are ready, they upgrade to OOP. Python doesn't force programmers to write OOP code when they just want to write a simple script.
- Extensibility: If the built-in methods and available third party mod- ules are not enough for your needs, you can easily extend Python, even in other programming languages. There are some applications written mostly in Python but with a processor demanding routine in C or FOR- TRAN. Python can also be extended by connecting it to specialized high level languages like R or MATLAB.
- Open source: Python has a liberal open source license that makes it freely usable and distributable, even for commercial use.
- Cross platform: A program made in Python can be run under any computer that has a Python interpreter. This way a program made under Windows Vista can run unmodified in Linux. Python interpreters are available for most computer and operating systems, and even some devices with embedded computers like the Nokia 6630 smartphone.
- Thriving community: Python is gaining momentum among the scientific community. This translates into more libraries for your projects and people you can go to for support.

Welche Termini in dem Textausschnitt sagen Dir überhaupt nichts? Unterstreiche Sie, und informiere dich im Internet über sie. Wenn es mehr als drei Termini sein sollten, wähle die drei, die Dir am besten gefallen.

## 7 Übungsaufgaben zur nächsten Sitzung

1. Installiere Python (Version 2.6) und die Bibliotheken NumPy, SciPy und NLTK auf Deinem Computer. Falls Du nicht weißt, wie das geht, suche Anleitungen im Internet (bspw. **http://diveintopython.org**) oder in Büchern aus der Bibliothek. Dokumentiere in Stichworten, wie man am einfachsten vorgeht, um Python auf seinem Computersystem zu installieren.
2. Schreibe Dein erstes Hallo-Welt-Programm in Python. Falls Du nicht weißt, was das ist, informiere dich darüber. Das Programm muss als solches nicht in einer Skriptdatei abgefasst werden. Es genügt, wenn Du den Code für ein Hallo-Welt-Programm aufschreibst und zusätzlich anmerkst, woher Du die Informationen hast.

## Literatur

Bassi, Sebastian. 2010. *Python for bioinformatics*. Boca Raton and London and New York: CRC Press.

# Programmieren

## 1 Was ist Programmieren?

### Von Programmen und Vorschriften

Alan Gauld erklärt den Begriff Programmieren wie folgt:

> Computer-Programmierung ist die Kunst, dass ein Computer das macht, was du willst. (`http://www.freenetpages.co.uk/hp/alan.gauld/german/tutwhat.htm`)

Wikipedia ist ein bisschen ausführlicher:

> Ein Computerprogramm oder kurz Programm ist eine Folge von den Regeln der jeweiligen Programmiersprache genügenden Anweisungen, die auf einem Computer ausgeführt werden können, um damit eine bestimmte Funktionalität zur Verfügung zu stellen. (vgl. `http://de.wikipedia.org/wiki/Computerprogramm`)

Im Rahmen einer *etymologischen Erklärung*, die insbesondere unter Pastoren sehr beliebt ist, spaltet man gewöhnlich ein Wort in seine Wurzelbestandteile auf, schaut nach, was diese bedeuten, und konstruiert aus dieser Bedeutung der Einzelteile dann eine Gesamtbedeutung für das Wort. Das Ganze sieht dann ungefähr so aus:

- *Programmieren* heißt *Programme erstellen*.
- Ein *Programm* ist der *vorgesehene Ablauf der Einzelheiten eines Vorhabens*.
- Das Wort *Programm* wurde laut Kluge & Seebold (2002) im 18. Jh. zum ersten Mal bezeugt.
- Es kommt vom lateinischen Wort *programma* “Bekanntmachung, Tagesordnung”.
- Das Wort *programma* kommt vom griechischen Wort πρόγραμμα ”id.”.
- Das Wort πρόγραμμα setzt sich zusammen aus προ “vor” und γράμμα “Schrift”.
- Ein *Programm* ist also eine *Vor-Schrift*.
- *Programmieren* heißt also *Vor-Schriften erstellen*.
- Wer Vorschriften erstellt, ist ein *Programmierer*.

Ignoriere die ironische Note der obigen Ausführungen einen Moment, und überlege, ob die Bezeichnung `Programmieren' für das, was darunter gewöhnlich verstanden wird, angesichts der etymologischen Hintergründe gerechtfertigt ist.

### Von Algorithmen und Rezepten

Im Zusammenhang mit dem Programmieren ist häufig von Algorithmen die Rede, und es stellt sich die Frage, in welcher Relation *Programm* und *Algorithmus* zueinander stehen. Im Gegensatz zur langweiligen Geschichte des Wortes *Programmieren*, ist die des *Algorithmus* viel schillernder, wie der folgende Eintrag im Kluge zeigt:

> **Algorithmus**. Substantiv Maskulinum, “Berechnungsverfahren”, peripherer Wortschatz, fachsprachlich (13. Jh., Form 16. Jh.), mhd. *algorismus*. Onomastische Bildung. Entlehnt aus ml. *algorismus*, das das Rechnen im dekadischen Zahlensystem und dann die Grundrechenarten bezeichnet. Das Wort geht zurück auf den Beinamen Al-Hwārizmī (”der Chwaresmier”, eine Herkunftsbezeichnung) eines arabischen Mathematikers des 9. Jhs., durch dessen Lehrbuch die (indischen und dann) arabischen Ziffern in Europa allgemein bekannt wurden. Das Original des hier in Frage kommendes Buches ist verschollen, die ml. Übersetzung ist *Liber algorismi de practica arismetrice*. Die Schreibung mit <th> in Anlehnung an gr. *arithmós* “Zahl”. Die Bedeutung ist (seit dem 13. Jh.) zunächst “Rechenkunst” (im Deutschen untergegangen, im Englischen als *algorism* von *algorithm* getrennt); die moderne Bedeutung “festgelegter komplexer Rechenvorgang” im Deutschen seit Ende des 19. Jhs. Ebenso ne. *algorithm*, nfrz. *algorithme*, nschw. *algoritm*, nnorw. *algoritme*. (Kluge & Seebold 2002)

Eine weniger die Geschichte, als vielmehr den Inhalt des Begriffs fassende Definition finden wir bei Brassard & Bratley (1993):

> Das *Concise Oxford Dictionary* definiert einen Algorithmus als “Verfahren oder Regeln für (speziell maschinelle) Berechnung”. Die Ausführung eines Algorithmus darf weder subjektive Entscheidungen beinhalten noch unsere Intuition und Kreativität fordern. Wenn wir über Algorithmen sprechen, denken wir meistens an Computer. Nichtsdestoweniger könnten andere systematische Methoden zur Lösung von Aufgaben eingeschlossen werden. So sind zum Beispiel die Methoden der Multiplikation und Division ganzer Zahlen, die wir in der Schule lernen, ebenfalls Algorithmen. […] Es ist sogar möglich, bestimmte Kochrezepte als Algorithmen aufzufassen, vorausgesetzt, sie enthalten keine Anweisungen wie “nach Geschmack salzen”. (Brassard & Bratley 1993)

Vergleiche die bisherigen Ausführungen zu den Begriffen *Programm* und *Algorithmus*. Worin könnte der grundlegende Unterschied zwischen den beiden Begriffen bestehen?

## 2 Ein Beispiel für einen Algorithmus: Kölner Phonetik

### Das Problem

Karlheinz, ein Düsseldorfer Student der Lingustik, hat auf einer Mediziner-Party in Köln eine hübsche Medizinstudentin kennengelernt, die ihr Physikum bereits abgeschlossen hat, und möchte sie gerne wiedersehen. Dummerweise kann er sich jedoch nicht mehr genau daran erinnern wie sie heißt. Ihr Nachname klang irgendwie nach [maiɐ], und ihr Vorname war irgendwas in Richtung [krɪsti:nə], jedoch weiß er nicht, welche Schreibung er zugrunde legen soll. Zufällig hat er eine Liste mit den wirklichen Namen und den dazugehörigen Facebook-Namen von allen Bürgern aus Köln und Umgebung als Excel-Tabelle auf seinem Computer zu Hause. Wenn er jetzt noch ein Verfahren finden könnte, das ihm alle Namen anzeigt, die wie [krɪsti:nə maiɐ] klingen, dann wäre es sicherlich ein Leichtes, herauszufinden, wo die unbekannte Studentin ihre Tierversuche veranstaltet, und sie mit einem Pausenkaffee von Starbucks zwischen Frosch und Kaninchen zu überraschen…

### Die Lösung

Die Lösung besteht darin, alle Namen, die auf der Liste auftauchen, in ein anderes Format umzuwandeln, welches den Sprachklang der Wörter wiedergibt und nicht ihre Schreibung. Ein Algorithmus, der

für diesen Zweck geschaffen wurde, ist die sogenannte Kölner Phonetik (vgl. Postel 1969). Die Kölner Phonetik wandelt Wörter der deutschen Sprache in einen *phonetischen Kode* um, mit dessen Hilfe auch Wörter, die unterschiedlich geschrieben werden, aber gleich klingen, verglichen werden können. So werden die beiden Namen *Christina Maier* und *Kirsten Mayr* durch den Algorithmus jeweils in die gleiche Zahlenfolge “47826-67” umgewandelt.

## Das Verfahren

Die Kölner Phonetik wandelt jeden Buchstaben eines Wortes in eine Ziffer zwischen “0” und “8” um. Bei der Umwandlung der Buchstaben wird auf bestimmte Kontextregeln Rücksicht genommen, die in der folgenden Tabelle zusammengefasst sind (vgl. `http://de.wikipedia.org/wiki/Kölner_Phonetik`):

| **Buchstabe** | **Kontext** | **Ziffer** |
|---|---|---|
| A, E, I, J, O, U, Y | | 0 |
| H | | - |
| B | | 1 |
| P | nicht vor H | |
| D, T | nicht vor C, S, Z | 2 |
| F, V, W | | 3 |
| P | vor H | |
| G, K, Q | | 4 |
| C | im Anlaut vor A, H, K, L, O, Q, R, U, X<br>vor A, H, K, O, Q, U, X außer nach S, Z | |
| X | nicht nach C, K, Q | 48 |
| L | | 5 |
| M, N | | 6 |
| R | | 7 |
| S, Z | | 8 |
| C | nach S, Z<br>im Anlaut außer vor A, H, K, L, O, Q, R, U, X<br>nicht vor A, H, K, O, Q, U, X | |
| D, T | vor C, S, Z | |
| X | nach C, K, Q | |

Für Umlaute (<ÄÖÜ>) und deutsche Sonderzeichen (bspw. <ß>) gibt es keine generellen Konventionen, jedoch können diese ohne Bedenken den jeweiligen bekannten Klassen (“o” oder “S”) zugeordnet werden.

Die Umwandlung erfolgt in der folgenden Reihenfolge:

1. Wandle jeden Buchstaben schrittweise um und beachte die Kontextregeln.
2. Reduziere alle mehrfach hintereinander auftauchenden Ziffern auf eine.
3. Entfernen die Ziffer “0” an allen Stellen des Wortes, außer am Anfang.

TableWende die Kölner Phonetik auf deinen Vor- und Nachnamen an und dokumentiere explizit, welche Schritte Du dabei durchführst. Warum ist das Verfahren komplizierter, als man am Anfang vermuten könnte?

# 3 Ein Beispiel für ein Programm: Python-Version der Kölner Phonetik

## Vorbemerkungen

Die Kölner Phonetik ist in Python von Robert Schindler implementiert worden und unter der URL `http://pypi.python.org/pypi/kph/0.1` erhältlich. Da die Umsetzung des Algorithmus in Python aufgrund der kontextsensitiven Ersetzungsregeln relativ kompliziert ist, wird darauf in diesem Zusammenhang nicht weiter eingegangen.

## Testen der Kölner Phonetik

Um die Python-Version der Kölner Phonetik testen zu können, öffnen wir zunächst eine interaktive Python-Shell, in der wir direkt Programmierbefehle eingeben und deren Ausgabe empfangen können. Wir benutzen für diese Zwecke die IDLE, welche standardmäßig mit Python installiert wird. Eine sofort ausführbare Version der Kölner Phonetik, welche nicht installiert werden muss, ist über den Link `http://lingulist.de/python/kph.py` abrufbar. Um sie zu testen, sind die folgenden Schritte notwendig:

1. Lade die Datei <kph.py> von der oben genannten Adresse herunter.
2. Öffne die IDLE Python-Shell.
3. Klicke auf **File > Open** und wähle die Datei <kph.py>.
4. Klicke im Fenster der geöffneten Datei auf **Run > Run Module**.
5. Versuche, zu ermitteln, was geschieht.

## Zum Aufbau von Python-Programmen

Wenn man sich die Datei <kph.py> genauer anschaut, so lässt sich feststellen, dass ihr ein gewisser Aufbau zugrunde liegt, der im Folgenden genauer besprochen werden soll.

### Der Header

Jede Python-Datei sollte mit einem Header beginnen. Dies ist zwar nicht zwingend erforderlich, es verhindert aber eine Vielzahl von Problemen. Der Header sollte dabei wie folgt aussehen:

```
#! /usr/bin/env python
# *-* coding:utf-8 *-*
```

Die erste Zeile des Headers ist dabei für Linux-Systeme und hat für Windows keine Relevanz: Er sagt dem Computer, wo der Python-Interpreter zu finden ist. Der zweite Header macht Angaben zur Kodierung der Datei und ist auf allen Systemen von großer Wichtigkeit, da man ohne diese Zeile Umlaute und Sonderzeichen in Python-Programmen nicht verwenden kann.

**Kommentare**

Wenn man Programme schreibt, wird generell alles, was man in ein Programm schreibt, ausgeführt. Um es möglich zu machen, Programme zu kommentieren, also etwas in ein Programm zu schreiben, dass nicht direkt ausgeführt wird, sondern lediglich der Beschreibung für denjenigen dient, der den Quellkode des Programms liest, gibt es in jeder Programmiersprache spezielle Zeichen, mit denen man Text in Programmen "auskommentieren" kann. Bei Python werden alle Zeilen, die mit dem Nummernzeichen '#' beginnen, auskommtentiert und haben für das, was das Programm tut, nicht die geringste Bedeutung, sie helfen jedoch demjenigen, der das Programm liest, besser zu verstehen, worum es geht. In der Datei <kph.py> wurde eine beispielhafte Kommentarzeile eingebaut:

```
# Dies ist eine Kommentarzeile, durch welche es möglich ist, Dinge in das
# Programm zu schreiben, die nachher nicht ausgeführt werden.
```

Es gibt noch eine weitere Möglichkeit, in Python Kommentare zu schreiben, und zwar, indem man Text in dreifache Anführungsstriche schreibt. Diese Methode wird vor allem dann verwendet, wenn mehr geschrieben werden muss. So werden Anwendungsbeispiele zum Beispiel sehr häufig auf diese Art geschrieben.

```
"""
This module implements the 'Koelner Phonetik'.

Examples of usage:"
>>> encode("Moritz Müller")
'678657'
>>> encode("Moriz Müler")
'678657'
>>> encode("Laura Mayer")
'5767'
>>> encode("Laura Meier")
'5767'
"""
```

**Einbinden von Modulen**

Programmierer sind faul und haben meist keine Lust, eine Operation, die einmal funktioniert hat, immer wieder zu verwenden. Aus diesem Grunde macht man Gebrauch von sogenannten *Modulen*. Dies sind einzelne Funktionen und ganze Bibliotheken, welche, sofern sie benötigt werden, abgerufen und in einem Programm verwendet werden können. Im Skript <kph.py> wird das Modul *re* eingebunden, welches die Möglichkeit biete, mit Hilfe regulärer Ausdrücke allgemeine Ersetzungsoperationen durchzuführen. Das Einbinden von Modulen ist in Python denkbar einfach:

```
import re
```

Sobald dies geschehen ist, kann man die Funktionen, die das Modul bereitstellt, verwenden, indem man den Modulnamen und die Funktion mit einem Punkt verbindet, wie im folgenden Beispiel, das man in der IDLE-Konsole eingeben kann:

```
>>> import re
>>> re. findall ('m','mama')
```

Die Funktion re. findall () macht nichts weiter, als jede Zeichenkette, die in einer Zeichenkette auftritt, zu suchen. Die Ausgabe ist folglich:

```
['m', 'm']
```

**Das print-Statement**

Eines der wichtigsten Wörter, das wir in Python noch sehr oft benutzen werden, ist das **print**-Statement, durch welches wir den Computer anweisen, eine Zeichenkette auf dem Terminal auszugeben. Um das Programm <kph.py> zu testen, wurden am Ende die folgenden **print**-Befehle eingefügt:

```
print encode("Christine"),encode("Maier")
print encode("Kirsten"),encode("Maier")
```

Der Computer wird mit Hilfe dieser Anweisungen angewiesen, die Funktion encode(), welche die Kölner Phonetik implementiert, auf die in die Klammern geschriebene Zeichenkette anzuwenden. Will man mehrere Werte auf einer Zeile ausgeben lassen, so müssen die Werte durch Kommata getrennt werden.

## 4 Übungen

1. Schau Dir das Skript <kph.py> genau an und vergleiche es mit der allgemeinen Beschreibung des Algorithmus.
2. Versuche das Skript so zu ändern, dass es den Kölner-Phonetik-Kode für Deinen Namen ausgibt.
3. Experimentiere mit dem Skript: Welche Ausgabe erhältst Du, wenn Du
   a) eine Zahl in Anführungsstrichen,
   b) eine Zahl ohne Anführungsstriche, oder
   c) Buchstaben mit diakritischen Zeichen (<áťň>) eingibst?
4. Gibt man statt encode("Kirsten"),encode("Maier") die Zeile encode("Kirsten␣Maier") ein, verändert sich die Ausgabe. Woran liegt das?
5. Schreibe das Programm so um, dass die folgende Ausgabe zustande kommt, wobei die doppelten Fragezeichen durch einen Wert ersetzt werden sollen.

Guten Tag, verehrte Freunde der gepflegten Namenssuche.
Wir demonstrieren heute den zweifellosen Höhepunkt des gepflegten Wortvergleichs:
Die Kölner Phonetik! In Köln ist Herr Maier die Nummer ?? ! Die Spinnen, die Kölner ...

## Literatur

Brassard, Gilles, & Paul Bratley. 1993. *Algorithmik. Theorie und Praxis*. Prentice Hall: Wolfram's Verlag.

Kluge, Friedrich, & Elmar Seebold. 2002. *Etymologisches Wörterbuch der deutschen Sprache*. Berlin: de Gruyter, 24th edition.

Postel, Hans Joachim. 1969. Die Kölner Phonetik. Ein Verfahren zur Identifizierung von Personennamen auf der Grundlage der Gestaltanalyse. *IBM-Nachrichten* 19.925–931.

# Datentypen und Variablen

## 1 Was ist eine Variable?

### Begriffserklärung

- *Variable* ist Substantiv zu *variabel*, welches auf Latein *variabilis* “veränderbar” zurückgeht.
- Eine *Variable* ist also etwas, das änderbar ist.
- Johnny Depp ist ein Beispiel für eine *Variable*, weil er sehr änderbar ist, wie man an seinen Filmen sehen kann.

### Gängige Definitionen

> In computer programming, a variable is a symbolic name given to some known or unknown quantity or information, for the purpose of allowing the name to be used independently of the information it represents. A variable name in computer source code is usually associated with a data storage location and thus also its contents, and these may change during the course of program execution. (Aus: `http://en.wikipedia.org/wiki/variable_(programming)`)

> Bei jeder Art von Datenverarbeitung beziehen sich die Anweisungen auf direkte Daten, die eingegeben werden oder auf Variable, die man sich vorstellen kann als mit Namen versehen Behälter für Daten. Jede Zuweisung an eine Variable füllt Daten in diesen Behälter, und der Wert einer Variablen ist der Inhalt dieses Behälters. So ist z. B. der Effekt einer Anweisung $A := 3 + 7$: Bilde mit den direkt eingegebenen Daten 3 und 7 die Summe 10 und weise diesen Wert der Variablen mit Namen A zu. Ein [sic!] anschließende Anweisung ”drucke A” druckt den Inhalt des Behälters A aus, den Wert der Variablen A, hier 10. (Puttkamer 1990, 188)

### Vereinfachte Begriffserklärung

Wenn wir ein Programm, also eine Folge von Anweisungen programmieren, dann werden wir zwangsläufig mit Werten arbeiten, die wir gerne *variabel* halten wollen, sei es, dass sie sich im Verlaufe des Programmes verändern, oder dass sie zu Beginn des Programms variabel festgelegt werden sollen. Wenn wir beispielsweise ein Programm schreiben wollen, das irgendeine Ganzzahl mit sich selbst multipliziert, das Ergebnis dieser Multiplikation wieder mit sich selbst multipliziert und das Endergebnis auf dem Terminal ausgibt, so müssen wir im Programm irgendwie auf diese Ganzzahl, die wir noch nicht kennen, zugreifen und sie verändern können. Dies wird beim Programmieren durch *Variablen* bewerkstelligt. Eine *Variable* ist dabei ein Platzhalter, der, wenn wir das Programm ausführen, mit einem Wert gefüllt wird. In diesem Zusammenhang spricht man davon, dass eine Variable *deklariert* wird.

## 2 Variablen in Python

### Allgemeines zu Deklaration von Variablen in Python

Das Deklarieren von Variablen in Python ist denkbar einfach und geschieht stets mit Hilfe des *Zuweisungsoperators* $=$, der nicht mit dem *Äquivalenzoperator* $==$ verwechselt werden darf. Der Zu-

weisungsoperator weist der variablen einen bestimmten Wert zu, welches sich allgemein wie folgt darstellen lässt,

```
>>> VARIABLE = VALUE
>>> VAR_1,VAR_2 = VAL_1,VAL_2
```

wobei VARIABLE für einen beliebigen Namen steht, der der von Python für Variablen geforderten Namensstruktur entspricht und VALUE für einen beliebigen Datentypen, der in Python zur Verfügung steht. Möchte man mehrere Variablen gleichzeitig deklarieren, so lässt sich das in einem einzigen Deklarationsschritt erledigen, wobei die Variablen und die Werte hier durch ein Komma getrennt voneinander eingegeben werden müssen:

### Struktur von Variablennamen

Genauso, wie dies für natürliche Sprachen auch gilt, in denen bestimmte Lautkombinationen nicht zur Schaffung von Wörtern verwendet werden können, weil sie nicht gebräuchlich, oder einfach unaussprechbar sind, sind auch in Python die Namen für Variablen nicht beliebig wählbar. Erlaubt sind grundsätzlich alle Zeichen des englischen Alphabets in Groß- und Kleinschreibung (<ABC...XYZabc...xyz>), sowie der Unterstrich (<_>). Zusätzlich können Ziffern als Variablennamen verwendet werden, jedoch nur in Kombination mit nicht-numerischen Zeichen und nicht zu Beginn eines Variablennamens, wie die folgenden Beispiele zeigen:

```
>>> Name = 1
>>> Name_von_mir = 1
>>> Name2 = 1
>>> 2Name = 1
SyntaxError: invalid syntax
```

### Programmbeispiele

Die folgenden Kodeblöcke weisen einer Variable mit dem Namen *Variable*, als Inhalt den String “Variable” zu.

```
>>> print 'Variable'
Variable
>>> Variable = 'Variable'
>>> print Variable
Variable
>>> Variable
'Variable'
>>> Variable == Variable
True
>>> Variable == 'Variable'
True
Variable1,Variable2 = 'Variable1','Variable2'
```

Was geschieht in den oben gegebenen Kodebeispielen? Worin besteht der Unterschied zwischen den beiden Befehlen, bei welchen jeweils der Äquivalenzoperator == verwendet wird?

### Fehlermeldungen

Wir haben inzwischen bereits gemerkt, dass Python immer dann, wenn ein Fehler im Programm auftaucht, eine Fehlermeldung ausgibt. Diese mag zu Beginn noch ein wenig kompliziert anmuten, jedoch wird sie sich für jeden, der ein wenig länger mit der Sprache programmiert hat, als äußerst hilfreich erweisen, insbesondere im Vergleich zu anderen Programmiersprachen (wie Perl, PHP und C++), die sich hinsichtlich ihrer Fehlermeldungen meist sehr bedeckt halten. In Python werden alle möglichen Fehler in Klassen eingeteilt, die auch vom Programmierer selbst definiert und erweitert werden können. Es gibt jedoch eine Menge bereits vorgefertigter und wichtiger Fehlerklassen (*Builtin*-Fehlerklassen), die immer dann aktiviert werden, wenn ein Fehler im Programm auftaucht, der einer dieser Klassen zugeordnet werden kann.

Im Zusammenhang mit der Deklaration von Variablen gibt es einige häufig auftauchende Fehler, die im folgenden Beispiel bewusst gemacht werden, um auf die entsprechenden Fehlertypen hinzuweisen:

```
>>> test = 1,2
>>> test = bla
Traceback (most recent call last):
File "<pyshell#13>", line 1, in <module>
    test = bla
NameError: name 'bla' is not defined
>>> print test1
Traceback (most recent call last):
File "<pyshell#14>", line 1, in <module>
    print test1
NameError: name 'test1' is not defined
>>> test1 = 2
>>> test2 = '2'
>>> test1 + test2
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    test1 + test2
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Schau Dir die Befehle genau an und teste selbst Varianten der Variablendeklaration. Wie lassen sich die Fehler erklären? Welche allgemeinen Fehler verbergen sich hinter den im Beispiel gegebenen Fehlern?

## 3 Was ist ein Datentyp?

### Gängige Definitionen

> Formal bezeichnet ein Datentyp in der Informatik die Zusammenfassung von Objektmengen mit den darauf definierten Operationen. Dabei werden durch den Datentyp des Datensatzes unter Verwendung einer so genannten Signatur ausschließlich die Namen dieser

> Objekt- und Operationsmengen spezifiziert. Ein so spezifizierter Datentyp besitzt noch keine Semantik. Die weitaus häufiger verwendete, aber speziellere Bedeutung des Begriffs Datentyp stammt aus dem Umfeld der Programmiersprachen und bezeichnet die Zusammenfassung konkreter Wertebereiche und darauf definierten Operationen zu einer Einheit. Zur Unterscheidung wird für diese Datentypen in der Literatur auch der Begriff Konkreter Datentyp verwendet. Für eine Diskussion, wie Programmiersprachen mit Datentypen umgehen, siehe Typisierung. (Aus: `http:de.wikipedia.org/wiki/Datentyp`)

### Vereinfachte Begriffserklärung

Wir haben oben festgestellt, dass eine Variable ein Platzhalter ist, der beim konkreten Ablauf des Programms mit einem bestimmten *Wert* gefüllt wird. Worum es sich bei diesem *Wert* genau handelt, wurde in der bisherigen Darstellung bewusst offen gelassen, es ist jedoch sicherlich einleuchtend, dass für verschiedene Operationen unterschiedliche Typen von Werten (*Datentypen*) zugrunde gelegt werden müssen. Wenn wir bspw. eine einfache Rechenoperation ausführen wollen, so sind es Zahlen (Datentyp *integer* oder *float*), die uns als Werte interessieren. Wenn wir mit Text arbeiten wollen, so bestehen unsere Werte aus Zeichenketten (Datentyp *string*), und wenn wir mit einer ganzen Reihe von Zahlen oder Zeichenketten arbeiten wollen (die bspw. sortiert werden sollen), dann brauchen wir zusammengesetzte Datentypen (Datentyp: *list*). Wenn wir Variablen deklarieren, müssen wir somit immer auch festlegen, was für einen Datentyp diese Variablen aufweisen sollen, obwohl dies nicht in allen Programmiersprachen (wie bspw. in Python) explizit getan werden muss. Das Festlegen von Datentypen bewerkstelligen wir dabei auch in der Umgangssprache ganz explizit, wenn man an Sätze wie “Multiplizier mal 1 und 1” denkt, bei denen wir automatisch wissen, dass es sich bei “1” um eine Zahl handelt und nicht um eine Schulnote, da man Zahlen sehr wohl, Schulnoten jedoch nicht miteinander multiplizieren kann.

## 4 Datentypen in Python

### Allgemeines zu Datentypen in Python

In Python werden Datentypen bei der Variablendeklaration nicht explizit angegeben, sondern aufgrund der Struktur der Werte, die den Variablen zugewiesen werden, automatisch bestimmt. Python weist eine Vielzahl von Datentypen auf und ermöglicht aufgrund seiner objektorientierten Ausrichtung auch die Erstellung eigener komplexer Datentypen. Zu den wichtigsten Datentypen zählen die folgenden:

- *integer*: fasst ganzzahlige Werte (2,3,−5,0).
- *float*: fasst Fließkommawerte (1.0,0.0,−0.5,2.333).
- *string*: fasst Zeichenketten (’Mattis’, ’List’).
- *list*: Der Datentyp “Liste” fasst jede Art von anderen Datentypen in linearer Anordnung ( [’Mattis’,1,1.0,[’List’]] ).
- *dict*: Der Datentyp “Dictionary” fasst jede Art von anderen Datentypen als Sammlung von Key-Value-Paaren ( {1:1.0,2:’Mattis’} ).

### Überprüfen von Datentypen

Um zu überprüfen, welchen Datentypen eine bestimmte Variable zu einem bestimmten Zeitpunkt aufweist, gibt es eine spezielle Typenbibliothek in Python, deren Verwendung recht unkompliziert ist. Es

wird zwar gemeinhin davon abgeraten, beim Programmieren auf derartige Typechecks zurückzugreifen, da man als Schöpfer seines Programms ja stets wissen sollte, was für Datentypen dieses produziert, es gibt jedoch gewisse Fälle, in denen man um einen Typecheck nicht umhinkommt. Ferner ist der Typecheck hilfreich, um sich klarzumachen, wie sich die verschiedenen Datentypen in Python verhalten. Die folgenden Befehle geben Beispiele für die Verwendung der Typenbibliothek in Python:

```
>>> from types import * # importiert alle Typen aus der Type-Bibliothek
>>> var1 = '3'
>>> var2 = 3
>>> var3 = 2
>>> var4 = 2.0
>>> type(var1)
<type 'str'>
>>> type(var2)
<type 'int'>
>>> type(var3)
<type 'int'>
>>> type(var4)
<type 'float'>
```

### Programmbeispiele

Die folgenden Kodeblöcke zeigen Anwendungsbeispiele für die Datentypen *integer*, *float* und *string*.

```
>>> var1 = 'ICH'
>>> var1 = var1 + var1
>>> print var1
ICHICH
>>> var2 = 1
>>> var2 = var2 + var2
>>> print var2
2
>>> var3 = 1.5
>>> var3 = var3 + var3
>>> print var3
3.0
>>> var4 = var1 / 3
>>> print var4
0
>>> var4 = var1 / var3
>>> print var4
0.66666666666666663
>>> var5 = var2 * var1
>>> print var5
ICHICHICHICH
```

Schau Dir die Befehle genau an und versuche selbst, sie zu variieren. Achte dabei insbesondere auf mögliche Fehlermeldungen. Wie lassen sich die zuweilen vom Datentypen abhängigen unterschiedlichen Ergebnisse erklären? Was lässt sich in Bezug auf die Bedeutung der arithmetischen Operatoren (+, −, *, /) im Verhältnis zu den jeweiligen Datentypen sagen?

# 5 Übung: Variablen im varen Leben: die Dreiballkaskade

## Was ist die Kaskade?

Bei der *Dreiballkaskade* handelt es sich um einen Algorithmus, mit dessen Hilfe sich drei Gegenstände von einem Menschen mit zwei beweglichen Armen und Händen jonglieren lassen können, Wikipedia beschreibt die Kaskade wie folgt:

> Als Kaskade wird das am einfachsten zu erlernende Jongliermuster mit einer ungeraden Anzahl von Gegenständen (Zum Beispiel: Bällen, Keulen oder Ringen) bezeichnet. Dabei wird mit zwei Gegenständen in einer Hand und einem in der anderen Hand angefangen. Der erste Wurf wird durch die Hand ausgeführt, in der zwei Gegenstände sind. Wenn der Gegenstand den höchsten Punkt erreicht, wird der Gegenstand aus der anderen Hand losgeworfen (und zwar unter dem zuvor geworfenen Gegenstand hindurch). Dadurch ist diese Hand frei, um den ersten Gegenstand zu fangen. Wenn der zweite Gegenstand am höchsten Punkt angekommen ist, wird der dritte Gegenstand losgeworfen (mit der Hand, die auch den ersten Gegenstand geworfen hat) und so weiter. Es wird also immer im Wechsel mit der rechten und der linken Hand geworfen. Die Kaskade ist für Einsteiger geeignet, weil gleichartige Wurf und Fangbewegungen stattfinden und die Hände feste Positionen haben, im Gegensatz zu Mustern wie Mills Mess oder Shower. (Aus: `http:de.wikipedia.org/wiki/Kaskade_(Jonglieren)`)

Warum kann man die Kaskade als einen Algorithmus bezeichnen?

## Wie man die Dreiballkaskade in Python modelliert

Ziel dieses Beispiels ist es, die Dreiballkaskade auf möglichst einfache Art mit Hilfe eines Computerprogramms zu simulieren. Dieses soll auf Terminalbasis visualisieren, wie die Bälle beim Jonglieren hin- und herwandern. Das Skript ist unter der Adresse `http://lingulist.de/python/kaskade1.txt` abrufbar.

Schau dir das Programm zur Simulation der Kaskade (<kaskade1.py>) genau an und versuche zu verstehen, wie es funktioniert. Ignoriere dabei Dinge, die Dir unbekannt sind, bzw. frage bewusst den Lehrenden, Kommilitonen, oder Wikipedia danach. Versuche dann, folgende Aufgaben zu lösen:

1. Wir wollen nicht mit Zahlen, sondern mit Smilies (:-), :-(, :-|) jonglieren. Wandle das Programm so um, das dies möglich ist.
2. Wir wollen das Jongliermuster wahlweise breiter oder höher gestalten. Wandle das Programm so um, dass nun doppelt so hoch oder doppelt so breit geworfen werden muss.
3. (für Fortgeschrittene) Wir wollen, dass das Programm nicht irgendwann aufhört zu laufen, sondern immer weiter läuft, wie lässt sich das bewerkstelligen?
4. (für Fortgeschrittene) Wir wollen, dass die Flugbahn der Bälle genau umgekehrt verläuft, wie lässt sich das bewerkstelligen?
5. Man könnte sich, wenn man das erste Mal von Variablen hört, fragen, wozu man diese überhaupt beim Programmieren benötigt. Inwiefern zeigt das kaskade1-Programm, dass man es ohne Variablen sehr schwer hätte?

## Literatur

Puttkamer, Ewald von (ed.) 1990. *Wie funktioniert das? Der Computer*. Mannheim and Wien and Zürich: Meyers Lexikonverlag.

# Operatoren

## 1 Allgemeines zu Operatoren

### Begriffserklärung

- *Operator* geht auf Latein *operātor* “Verfertiger, Arbeiter” zurück.
- Ein *Operator* ist also wer, der *irgendetwas* “verfertigt”.
- Was der *Operator* verfertigt, ist abhängig vom Kontext, in dem das Wort verwendet wird.

### 1.1 Gängige Definitionen

#### Operatoren in der Mathematik

> Ein Operator ist eine mathematische Vorschrift (ein Kalkül), durch die man aus mathematischen Objekten neue Objekte bilden kann. Er kann eine standardisierte Funktion oder eine Vorschrift über Funktionen sein. Anwendung finden die Operatoren bei Rechenoperationen, also bei manuellen oder bei maschinellen Berechnungen. (Aus: `http://de.wikipedia/wiki/Operator_(Mathematik)`)

#### Operatoren in der Logik

> Ein Logischer Operator ist eine Funktion, die einen Wahrheitswert liefert. Bei der zweiwertigen, booleschen Logik liefert er also wahr oder falsch, bei einer mehrwertigen Logik können auch entsprechend andere Werte geliefert werden. (Aus: `http://de.wikipedia/wiki/Operator_(Logik)`)

### 1.2 Vereinfachte Begriffserklärung

- Wenn wir uns den Operator als einen Verfertiger von Objekten vorstellen, dann heißt das, dass ein Operator *irgendetwas* mit *irgendetwas* anstellt.
- Entscheidend sind hierbei die Fragen,
  a) *was* der Operator tut, und
  b) *womit* der Operator etwas tut.
- Ein Operator wandelt also eines oder mehrere Objekte in neue Objekte um.
- Das, was umgewandelt wird, nennen wir die *Operanden* eines Operators.
- Das, was der Operator mit den Operanden tut, nennen wir die *Operation*.
- Der Plus-Operator wandelt bspw. zwei Zahlen in eine Zahl um, indem er die Operation *Addition* ausführt.

### 1.3 Klassifikation von Operatoren

- Operatoren können nach verschiedenen Kriterien klassifiziert werden.
- Die grundlegendste Unterscheidung besteht in der Anzahl der Operanden, auf die ein Operator die Operation anwendet.
- Hierbei unterscheiden wir zwischen *unären* und *binären* Operatoren.
  - *Unäre* Operatoren haben nur einen Operanden.
  - *Binäre* Operatoren haben zwei Operanden.
- Ferner kann zwischen *arithmetischen*, *logischen* und *relationalen* Operatoren unterschieden werden.
  - *Arithmentische* Operatoren führen arithmetische Operationen aus (Addition, Division, etc.).
  - *Logische* Operatoren liefern Aussagen über Wahrheitswerte (wahr oder falsch).
  - *Relationale* Operatoren liefern Aussagen über die Beziehungen zwischen Objekten (Identität, Zugehörigkeit).

Gib je ein Beispiel für

1. einen unären Operator,
2. einen binären Operator,
3. einen arithmetischen Operator,
4. einen logischen Operator und
5. einen relationalen Operator.

### 1.4 Überladen von Operatoren

- Normalerweise werden Operatoren immer nur für spezifische Datentypen definiert.
- Der Additionsoperator $+$ nimmt als Operanden bspw. gewöhnlich nur Zahlen.
- In vielen Programmiersprachen ist es jedoch üblich, Operatoren, je nach Datentyp, auf den sie angewendet werden, unterschiedliche Operationen zuzuschreiben.
- Diesen Vorgang nennt man das *Überladen* von Operatoren.
- Wird der Additionsoperator $+$ bswp. auf den Datentyp *string* angewendet, so bewirkt er eine Verkettung von Strings (Konkatenation).
- Das Überladen von Operatoren ermöglicht es, sehr kompakt und flexibel zu programmieren.

## 2 Operatoren in Python

### 2.1 Allgemeines zu Operatoren in Python

- Die Operatoren in Python ähneln denen vieler Programmiersprachen.

- Neben den durch mathematische Zeichen dargestellten Operatoren ($+$, $-$, $*$) sind einige Operatoren auch als *Namen* (**is**, **in**) definiert.
- Die Überladung von Operatoren ist ein entscheidender Wesenszug der Sprache. Viele Operatoren sind auf viele Datentypen anwendbar.

## 2.2 Arithmetische Operatoren

### Allgemeines

| Operator | Klass. | Name | Erläuterung | Beispiel |
|---|---|---|---|---|
| + | unär | ? | verändert den Operanden nicht | +x |
| − | unär | Negation | verändert den Operanden | -x |
| + | binär | Addition | liefert die Summe der Operanden | x + y |
| − | binär | Subtraktion | liefert die Differenz der Operanden | |
| * | binär | Multiplikation | liefert das Produkt der Operanden | x * y |
| / | binär | Division | liefert den Quotienten der Operanden als Gleitkommazahl | x / y |
| // | binär | Division | liefert den Quotienten der Operanden als Ganzzahl | x // y |
| % | binär | Modulo | liefert den Rest der Division der Operanden | x % y |
| ** | binär | Potenz | liefert die Potenz der Operanden | x ** y |

### Beispiele

```
>>> x,y = 5,2
>>> +x
5
>>> -x
-5
>>> x + y
7
>>> x - y
3
>>> x * y
10
>>> x / y
2
>>> x / float(y)
2.5
>>> x // float(y)
2.0
>>> x % y
1
>>> x ** y
25
```

Überprüfe, welche Oparationen die arithmetischen Operatoren an Strings durchführen können, stelle also heraus, ob und wenn ja wie die Operatoren überladen sind.

## 2.3 Logische Operatoren

### Allgemeines

- Die logischen Operatoren dienen dem Verarbeiten von Wahrheitswerten.
- Jeder Wert bekommt in Python automatisch einen Wahrheitswert zugewiesen (**True** oder **False**).
- Da die Wahrheitswerte selbst Werte sind, kann ein Wert auch nur ein Wahrheitswert sein, dieser wird dann dem Datentyp *bool* zugeschrieben.
- Negative Zahlen einschließlich der Zahl $0$ besitzen den Wahrheitswert **False**.
- Alle "leeren" Werte (der leere String "" oder die leere Liste []) besitzen ebenfalls den Wahrheitswert **False**.
- Die logischen Operatoren prüfen den Wahrheitswert von Werten und liefern als Ergebnis einen Wahrheitswert zurück.

| **Operator** | **Klass.** | **Name** | **Erläuterung** | **Beispiel** |
|---|---|---|---|---|
| `not` | unär | Negation | kehrt den Wahrheitswert des Operanden um | not x |
| `and` | binär | Konjunktion | verknüpft die Wahrheitswerte der Operanden als *UND*-Verbindung | x and y |
| `or` | binär | Disjunktion | verknüpft die Wahrheitswerte der Operanden als *ODER*-Verbindung | x or y |

Die Verknüpfung der Wahrheitswerte richtet sich nach der folgenden Wahrheitstafel (vgl. Weigend (2008:104)):

| a | b | not a | a and b | a or b |
|---|---|---|---|---|
| False | False | True | False | False |
| False | True | True | False | True |
| True | False | False | False | True |
| True | True | False | True | True |

Wenn andere Werte als der Datentyp *bool* mit den Operatoren **and** und **or** verknüpft werden, so wird immer einer der beiden Operanden zurückgegeben, wobei die Bedingungen für die Rückgabe wie folgt sind:

**and**
- Wenn der erste Operand falsch ist, wird dieser zurückgegeben.
- Wenn der erste Operand wahr ist, wird der zweite Operand zurückgegeben.

**or**
- Wenn der erste Operand wahr ist, wird dieser zurückgegeben.
- Wenn der erste Operand falsch ist, wird der zweite Operand zurückgegeben.

**Beispiele**

```
>>> x,y = True,False
>>> not x
False
>>> not y
True
>>> x and y
False
>>> x or y
True
>>> x,y = False,False
>>> not x and not y
True
>>> x and y
False
>>> x or y
False
>>> x,y = 'mattis','list'
>>> x and y
'list'
>>> x or y
'mattis'
>>> x = False
>>> x and y
False
>>> x or y
'list'
```

Gegeben ist die Aussage `Immer wenn die Sonne scheint oder es regnet, und egal ob es stürmt oder windstill ist geht Herr Maier spazieren.' Überprüfe, ob Herr Maier spazieren geht, wenn

1. die Sonne scheint, es regnet und stürmt,
2. es regnet und windstill ist, und
3. es stürmt.

## 2.4 Relationale Operatoren

### Allgemeines

- Relationale Operatoren liefern immer einen Wahrheitswert in Bezug auf die Relation zurück, die überprüft werden soll.
- Bei den relationalen Operatoren kann zwischen Vergleichs-, Element- und Identitätsoperatoren unterschieden werden.
  - Vergleichsoperatoren dienen dem Vergleich von Operanden hinsichtlich ihrer Werte.

- Zugehörigkeitsoperatoren dienen der Ermittlung der Zugehörigkeit eines Operanden zu anderen Operanden.
- Identitätsoperatoren dienen der Ermittlung von Identitätsverhältnissen, wobei Identität von Gleichheit ähnlich abgegrenzt wird wie dies im Deutschen mit Hilfe der Wörter *dasselbe* vs. *das gleiche* getan wird.

| Operator | Klass. | Name | Erläuterung | Beispiel |
|---|---|---|---|---|
| `<` | binär | kleiner | prüft, ob ein Operand kleiner als der andere ist | x > y |
| `>` | binär | größer | prüft, ob ein Operand größer als der andere ist | x > y |
| `==` | binär | gleich | prüft, ob ein Operand gleich dem anderen ist | x == y |
| `!=` | binär | ungleich | prüft, ob ein Operand ungleich dem anderen ist | x != y |
| `<=` | binär | kleiner-gleich | prüft, ob ein Operand kleiner oder gleich dem anderen ist | x <= y |
| `>=` | binär | größer-gleich | prüft, ob ein Operand größer oder gleich dem anderen ist | x >= y |
| `in` | binär | Element von | prüft, ob ein Operand Element eines anderen ist | x in y |
| `not in` | binär | nicht Element von | prüft, ob ein Operand nicht das Element eines anderen ist | x not in y |
| `is` | binär | Identisch | prüft, ob ein Operand denselben Speicherort wie ein anderer besitzt | x is y |
| `is not` | binär | nicht Identisch | prüft, ob ein Operand nicht denselben Speicherort wie ein anderer besitzt | x is not y |

Angenommen, die Operatoren `<=`, `>=`, `!=` stünden in Python nicht zur Verfügung. Wie ließen sich mit Hilfe der Kombination der bisher vorgestellten Operatoren dennoch den beiden Operationen äquivalente Operationen durchführen?

**Beispiele für die Verwendung von Vergleichsoperatoren**

```
>>> x,y = 5,10
>>> x > y
False
>>> x < y
True
>>> x == y
False
>>> x != y
True
>>> x <= y
True
>>> x >= y
False
```

**Beispiele für die Verwendung von Zugehörigkeitsoperatoren**

```
>>> x,y = 'mattis','is'
>>> x in y
False
>>> y in x
True
>>> x not in y
True
```

**Beispiele für die Verwendung von Identitätsoperatoren**

```
>>> x = 1,2
>>> y = x
>>> x is y
True
>>> y = 1,2
>>> x is y
False
```

## 3 Übung: Dreiballkaskade in zwei Varianten

Die Dreiballkaskade von der letzten Sitzung wurde erweitert und liegt nun als das Programm <kaskade2.py> vor. Lade die Datei herunter, schau sie Dir an, und versuche zu ermitteln, was das Programm macht, indem Du die Angaben im Quellkode genau verfolgst. Versuche dann, folgende Aufgaben zu lösen:

1. Erweitere die Anzahl der SnapShots, die von dem Jongliervorgang gemacht werden.
2. In der letzten Kontrollstruktur im Programm werden Aussagen mit Hilfe des logischen Operators **not** in Klammern gesetzt. Überprüfe mit Hilfe der Konsole, ob dies notwendig ist, und überlege, wenn dies zutreffen sollte, warum dies der Fall ist.
3. Denke Dir alternative Möglichkeiten aus, das Vor- oder Zurückspulen des Programms zu kontrollieren.

## Literatur

Weigend, Michael. 2008. *Python ge-packt*. Heidelberg: mitP, 4 edition.

# Kontrollstrukturen

## 1 Allgemeines zu Kontrollstrukturen

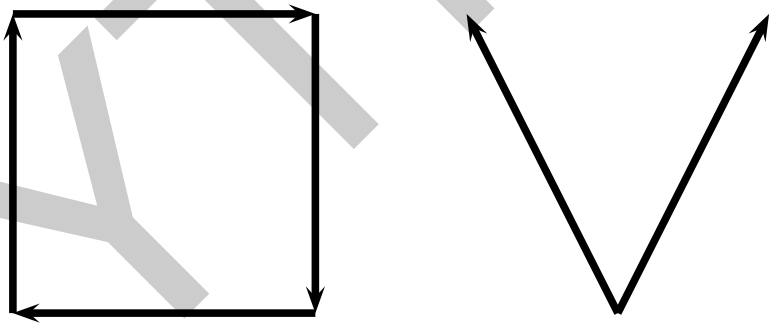
### Gängige Begriffserklärung

Kontrollstrukturen (Steuerkonstrukte) werden in imperativen Programmiersprachen verwendet, um den Ablauf eines Computerprogramms zu steuern. Eine Kontrollstruktur gehört entweder zur Gruppe der Verzweigungen oder der Schleifen. Meist wird ihre Ausführung über logische Ausdrücke der booleschen Algebra beeinflusst. (Aus: `http://de.wikipedia.org/wiki/Kontrollstruktur`)

Kontrollstrukturen legen fest, in welcher Reihenfolge und unter welchen Bedingungen die Anweisungen eines Programms abgearbeitet werden. (Weigend 2008:127)

### Typen von Kontrollstrukturen

- **Verzweigungen** kontrollieren den Fluss eines Programms, indem sie dieses, abhängig von bestimmten Gegebenheiten, in unterschiedliche Bahnen lenken
- **Schleifen** kontrollieren den Fluss eines Programms, indem sie Operationen wiederholt auf Objekte anwenden





## 2 Kontrollstrukturen in Python

- Python kennt insgesamt vier verschiedene Kontrollstrukturen: zwei Verzweigungsstrukturen (if,try) und zwei Schleifenstrukturen (for, while).
- Das besondere an den Kontrollstrukturen in Python ist deren enge Anbindung an logische Ausdrücke, welche einen sehr kompakten, sehr leicht verständlichen Kode erlauben.

## 2.1 Verzweigungen: if, elif und else

### Allgemeines

- Bei der **if**-Verzweigung wird ein Programmabschnitt unter einer bestimmten Bedingung ausgeführt.
- Dabei wird eine Bedingung auf ihren Wahrheitswert getestet. Trifft sie zu, wird das Programm entsprechend weitergeführt. Trifft sie nicht zu, unterbleibt der Abschnitt.
- Es können auch mehrere Bedingungen auf ihren Wahrheitswert überprüft und entsprechend mehrere verschiedene Programmabschnitte aktiviert werden.
- Da die Überprüfung in Python über den Wahrheitswert von Objekten oder Aussagen vorgenommen wird, können die Bedingungen sehr komplex aber auch sehr kompakt gestaltet werden.
- Es gibt in Python auch die Möglichkeit, *nichts* zu tun, wenn eine Bedingung zutrifft. Dies wird durch das Schlüsselwort **pass** bewerkstelligt. Auf diese Weise ist es möglich, nur beim Nichtzutreffen einer Bedingung eine bestimmte Operation auszuführen (vgl. die Beispiele).

### Beispiele

```
>>> x,y,yes,no = 10,0,"yes","no"
>>> if x: print yes
yes
>>> if y: print yes
yes
>>> if not x: print yes
    elif y: print no
no
>>> if 1 > 10: print "Eins ist groesser als zehn."
    elif 1 < 10: print "Eins ist kleiner als zehn."
Eins ist kleiner als zehn.
>>> if x or y: print "Einer von beiden Werten ist wahr."
    else: print "Keiner von beiden Werten ist wahr."
Einer von beiden Werten ist wahr.
>>> if x == 10: pass
    else: print "Alles wird gut (Nina Ruge)"
```

## 2.2 Verzweigungen: try und except

### Allgemeines

- Eine sehr wichtige und nützliche Verzweigunsstruktur bietet Python mit **try** und **except**.
- Hierbei wird nicht eine Bedingung auf ihren Wahrheitswert überprüft, sondern getestet, ob ein Statement eine Fehlermeldung hervorruft.
- Auf diese Weise kann man gezielt auf mögliche Fehler reagieren.
- Fehler können dabei gezielt entsprechend ihrer Klasse eingeordnet werden.

- Auf diese Weise können gezielt bestimmte Fehler angesprochen werden, die vom Programm ignoriert werden sollen.

**Aufgabe**

Schreibe ein kleines Programm, das zwei Integer addiert, und bei Eingabe von Werten, die keine Integer sind, auf eine fehlerhafte Eingabe hinweist.

**Beispiele**

```
>>> try: x = int(raw_input())
    except: print "Der Wert, den Sie eingegeben haben, ist kein Integer!"
2
>>> try: x = int(raw_input())
    except: print "Der Wert, den Sie eingegeben haben, ist kein Integer!"
2.0
Der Wert, den Sie eingegeben haben, ist kein Integer!"
>>> x = raw_input()
10
>>> y = raw_input()
0
>>> try: x / y
    except ZeroDivisionError: print "Durch Null kann man nicht teilen!"
Durch Null kann man nicht teilen.
>>> try: x = int(raw_input())
    except ValueError: print "Falscher Wert für Integer!"
10.0
Falscher Wert für Integer!
```

**Aufgabe**

Schreibe ein kleines Programm, das zwei Integer addiert und bei Floats oder Strings in der Eingabe auf falsche Eingabewerte hinweist. Überlege, worin der große Vorteil der except-Anweisung gegenüber einfachen if-Verzweigungen besteht.

## 2.3 Schleifen: while

**Allgemeines**

- Mit Hilfe von Schleifenkonstruktionen werden Blöcke von Anweisungen in Programmen wiederholt ausgeführt, bis eine Abbruchsbedingung eintritt, bzw. so lange eine Bedingung zutrifft.
- Wie für Verzweigunsstrukturen werden auch die Bedingungen in Schleifen auf Grundlage der Auswertung von Wahrheitswerten errechnet.
- Der Abbruch einer Schleife kann in Python bewusst durch das Schlüsselwort **break** gesteuert werden.

**Beispiele**

```
>>> a = 10
>>> while a > 0:
        print a," ist groesser als 0."
        a -= 1 # wir lassen der Wert gegen 0 laufen

10 ist groesser als 0
9 ist groesser als 0
8 ist groesser als 0
7 ist groesser als 0
6 ist groesser als 0
5 ist groesser als 0
4 ist groesser als 0
3 ist groesser als 0
2 ist groesser als 0
1 ist groesser als 0
>>> while True:
        b = raw_input("Sag was: ")
        print b
        if not b:
            print "Aaaaaabbbbbrrrrruuuuccccccch!"
            break
Sag was: bla
bla
Sag was: blu
blu
Sag was: bloed
bloed
Aaaaaabbbbbrrrrruuuuccccccch!
```

**Aufgabe**

Schreibe ein Programm, dass die Zahlen von 1 bis 50 miteinander addiert (1 + 1, 2 + 2) und das Ergebnis auf dem Terminal ausgibt. Wie lässt sich ein solches Programm entsprechend erweitern, dass der Benutzer bestimmen kann, wie viele Werte miteinander addiert werden sollen (also Werte von 0 bis 100, 0 bis 1000 etc.).

## 2.4 Schleifen: for

**Allgemeines**

- Während bei der **while**-Schleife die Abbruchsbedingung immer explizit genannt werden muss, gilt dies nicht für die **for**-Schleife.
- Hier wird explizit und im Voraus festgelegt, wie oft, bzw. in Bezug auf welche Objekte eine bestimmte Operation ausgeführt werden soll.

- Eine **for**-Schleife bietet sich also immer dann an, wenn man weiß, welche Objekte behandelt werden sollen.
- In Fällen, in denen der Bezugsrahmen nicht klar ist, muss man auf die **while**-Schleife zurückgreifen.
- Der Bereich, in dem in Python eine Operation mit Hilfe der **for**-Schleife ausgeführt wird, wird zumeist mit Hilfe der Funktion **range()** angegeben, welche automatisch eine Liste von Integern erzeugt, über die dann iteriert wird (vgl. die Beispiele).

**Beispiele**

```
>>> a = range(5)
>>> a
[0, 1, 2, 3, 4]
>>> for i in range(5): print i
0
1
2
3
4
>>> einkaufsliste = ['mehl','butter','zitronen','bier']
>>> for element in einkaufsliste: print element
mehl
butter
zitronen
bier
>>> namen = ['peter','frank','karl','jan']
>>> for name in namen: print name
peter
frank
karl
jan
>>> woerter = ['haus','kaffee','getraenk','wasser','flasche']
>>> for wort in woerter: print wort
haus
kaffee
getraenk
wasser
flasche
```

**Aufgabe**

Schreibe ein Programm, dass die Zahlen von 1 bis 50 miteinander addiert (1 + 1, 2 + 2) und das Ergebnis auf dem Terminal ausgibt, wobei statt einer **while**-Schleife, nun eine **for**-Schleife verwendet werden soll. Diskutiere die Vor- und Nachteile von **for**-Schleifen gegenüber **while**-Schleifen.

## 3 Übung

Die Dreiballkaskade der letzten Sitzung wurde wieder erweitert und liegt nun als das Programm <kaskade3.py> vor. Das Programm erlaubt es diesmal zusätzlich, zwischen verschiedenen Tricks zu wählen, die jongliert werden sollen. Schau Dir das Programm genau an, und setze dich mit den folgenden Fragen auseinander:

- Welche Rolle erfüllen die Verzweigungen und Schleifen?
- Die Jonglierpositionen werden durch eine Eingabe des Benutzers bestimmt. Falls dieser fehlerhafte Eingaben liefert, werden diese abgefangen. Wie lässt sich das Programm so ändern, dass bei einer falschen Eingabe einfach ein Standardmuster jongliert wird, anstatt dass sich die Eingabeschleife wiederholt?
- Der Shower ist das erste Jongliermuster, das man wählen kann. Dabei werden die Bälle alle im Kreis geworfen. Wie muss das Programm geändert werden, um einen Shower in umgekehrter Richtung als vierte Option für einen Jongliertrick zu ermöglichen?

## Literatur

Weigend, Michael. 2008. *Python ge-packt*. Heidelberg: mitP, 4 edition.

# Funktionen

## 1 Allgemeines zu Funktionen

### Gängige Begriffserklärung

> In der Mathematik ist eine Funktion oder Abbildung eine Beziehung zwischen zwei Mengen, die jedem Element der einen Menge (Funktionsargument, unabhängige Variable, x-Wert) genau ein Element der anderen Menge (Funktionswert, abhängige Variable, y-Wert) zuordnet. Das Konzept der Funktion oder Abbildung nimmt in der modernen Mathematik eine zentrale Stellung ein; es enthält als Spezialfälle unter anderem parametrische Kurven, Skalar- und Vektorfelder, Transformationen, Operationen, Operatoren und vieles mehr. (aus: `http://de.wikipedia.org/wiki/Funktion_(Mathematik)`)

> Eine Funktion (engl.: function, subroutine) ist in der Informatik die Bezeichnung eines Programmierkonzeptes, das große Ähnlichkeit zum Konzept der Prozedur hat. Hauptmerkmal einer Funktion ist es, dass sie ein Resultat zurückliefert und deshalb im Inneren von Ausdrücken verwendet werden kann. Durch diese Eigenschaft grenzt sie sich von einer Prozedur ab, die nach ihrem Aufruf kein Ergebnis/Resultat zurück liefert. Die genaue Bezeichnung und Details ihrer Ausprägung ist in verschiedenen Programmiersprachen durchaus unterschiedlich. (aus: `http://de.wikipedia.org/wiki/Funktion_(Programmierung)`)

### Typen von Funktionen

Da entscheidend für Funktionen der Eingabewert und der Rückgabewert sind, können wir ganz grob die folgenden speziellen Typen von Funktionen identifizieren:

- Funktionen ohne Rückgabewert (Prozedur)
- Funktionen ohne Eingabewert
- Funktionen mit einer festen Anzahl von Eingabewerten
- Funktionen mit einer beliebigen Anzahl von Eingabewerten

## 2 Funktionen in Python

### 2.1 Allgemeines zu Funktionen in Python

In Python sind Funktionen spezielle Datentypen. Sie unterscheiden sich von anderen Datentypen wie **strings** oder **integers** dadurch, dass sie zusätzlich aufgerufen werden können (*callable types*).

### Allgemeine Struktur

```
def functionName(
        parameterA,
        ...,
        keywordA='defaultA',
        ...,
        ):
    """
    docString
    """
    functionBody
    return functionValue
```

### Beispiel

```
#! /usr/bin/env python
# *-* coding:utf-8 *-*

def stoneAgeCalculator(intA,intB,calc = '+'):
    """
    This is the famous stoneAgeCalculator, a program written by the very first
    men on earth who brought the fire to us and were the first to dance naked
    on the first of May.
    """
    # check for inconsistencies in the input for keyword calc
    if calc not in ['+','-','*','/']:
        raise ValueError('The operation you chose is not defined.')

    # start the calculation, catch errors from input with a simple
    # try-except-structure
    try:
        if calc == '+':
            return intA + intB
        elif calc == '-':
            return intA - intB
        elif calc == '*':
            return intA * intB
        else:
            return intA / intB
    except:
        raise ValueError('No way to operate on your input.')

if __name__ == '__main__':
    calculation = stoneAgeCalculator(2,2)
    print "Das Ergebnis ist:",calculation
    raw_input("Druecken Sie ENTER, um das Programm zu beenden.")
```



## 2.2 Aufrufen und Ausführen

### Allgemeines

Eine Funktion wird aufgerufen, indem der Name der Funktion angegeben wird, gefolgt von einer Liste aktueller Parameter in Klammern. (Weigend 2008:144)

**Beispiel**

```
>>> a = range(5)
>>> a = int('1')
>>> range(5)
[0, 1, 2, 3, 4]
>>> int('1')
1
>>> float(1)
1.0
>>> print('Hallo Welt!')
Hallo Welt!
>>> list('hallo')
['h', 'a', 'l', 'l', 'o']
```

## 2.3 Die Funktionsdokumentation

### Allgemeines

Für jede Funktion, die in Python definiert wird, steht eine automatische Dokumentation bereit, welche mit Hilfe der Parameter und der Angaben im Docstring generiert wird. Die Dokumentation einer Funktion wird mit Hilfe von help(function) aufgerufen. Mit Hilfe der Funktionsdokumentation lässt sich leicht ermitteln, welche Eingabewerte eine Funktion benötigt, wie sie verwendet wird, und welche Werte sie zurückliefert. Docstrings lassen sich am besten interaktiv einsehen.

**Beispiel**

```
>>> help(range)
"""
Help on built-in function range in module __builtin__:

range(...)
    range([start,] stop[, step]) -> list of integers

    Return a list containing an arithmetic progression of integers.
    range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!) defaults to 0.
    When step is given, it specifies the increment (or decrement).
    For example, range(4) returns [0, 1, 2, 3]. The end point is omitted!
    These are exactly the valid indices for a list of 4 elements.
"""
```

**Aufgabe**

Der Logarithmus wird in Python mit Hilfe des Moduls math berechnet. Die entsprechende Funktion heißt log. Informiere Dich mit Hilfe des Docstrings der Funktion über ihre Arbeitsweise und berechne entsprechend den dekadischen Logarithmus, den natürlichen und den dualen Logarithmus der Zahl 10.

## 2.4 Parameter

### Allgemeines

Mit Hilfe von Funktionen werden Eingabewerte (Parameter) in Ausgabewerte umgewandelt. Bezüglich der Datentypen von Eingabewerten gibt es in Python keine Grenzen. Alle Datentypen (also auch Funktionen selbst) können daher als Eingabewerte für Funktionen verwendet werden. Gleichzeitig ist es möglich, Funktionen ohne Eingabewerte oder ohne Ausgabewerte zu definieren. Will man eine beliebige Anzahl an Parametern als Eingabe einer Funktion verwenden, so erreicht man dies, indem man der Parameterbezeichnung einen Stern (*) voranstellt. Die Parameter von Eingabeseite werden innerhalb der Funktion dann automatisch in eine Liste umgewandelt, auf die mit den normalen Listenoperationen zugegriffen werden kann.

### Beispiele

```
>>> def leer():
...     print "leer"
>>> def empty():
...     pass
>>> def gruss1(wort):
...     print wort
>>> def gruss2(wort):
...     return wort
>>> def eins():
...     """Gibt die Zahl 1 aus."""
...     return 1
>>> leer()
leer
>>> empty
... <function empty at 0xb76ef374>
>>> empty()
>>> type(empty)
<type 'function'>
>>> a = eins()
>>> a
1
>>> print bla.func_doc
Gibt die Zahl 1 aus.
>>> def printWords(*words):
...     for word in words:
...         print word
...
>>> printWords('mama','papa','oma',opa')
mama
papa
oma
opa
```

## 2.5 Schlüsselwörter

### Allgemeines

Als Schlüsselwörter bezeichnet man Funktionsparameter, die mit einem Standardwert belegt sind. Werden diese beim Funktionsaufruf nicht angegeben, gibt es keine Fehlermeldung, sondern anstelle der fehlenden Werte wird einfach auf den Standardwert zurückgegriffen. Gleichzeitig braucht man

beim Aufrufen nicht auf die Reihenfolge der Parameter zu achten, da diese ja durch die Anbindung der Schlüsselwörter vordefiniert ist. Weist eine Funktion hingegen Schlüsselwörter und Parameter auf, so müssen die Parameter den Schlüsselwörtern immer vorangehen.

**Beispiel**

```
>>> def wind(country,season='summer')
...     """Return the typical wind conditions for a given country."""
...     if country == Germany and season == 'summer':
...         print "There's strong wind in",country
...     else:
...         print "This part hasn't been programmed yet."
...
>>> wind('Germany'):
There's strong wind in Germany.
>>> wind('Germany,season='winter')
This part hasn't been programmed yet.
```

**Aufgabe**

Schreibe das Skript <stoneAgeCalculator.py> so um, dass die Funktion nur Schlüsselwörter aufweist.

## 2.6 Prozeduren

### Allgemeines

Prozeduren sind Funktionen, die den Wert **None** zurückgeben. Fehlt in einer funktionsdefinition die **return**-Anweisung, wird automatisch der Wert **None** zurückgegeben. Es ist kein Fehler, wenn auf der reichten Seite einer Zuweisung ein Prozeduraufruf steht. (Weigend 2008:151)

**Beispiel**

```
>>> def quak(frosch=''):
...     print "quak"
...     print "quak"
...     print "quak"
...
>>> quak()
quak
quak
quak
>>> a = quak()
quak
quak
quak
>>> type(a)
<type 'NoneType'>
```

**Aufgabe**

Schreibe das Skript <stoneAgeCalculator.py> so um, dass die Funktion keinen Rückgabewert mehr aufweist, ohne, dass sich deren Ausgabe ändert.

## 2.7 Rekursive Funktionen

### Allgemeines

Rekursive Funktionen sind solche, die sich selbst aufrufen. (Weigend 2008:151)

### Beispiel

```
>>> def factorial(number):
...     """
...     Aus: http://www.dreamincode.net/code/snippet2800.htm
...     """
...     if number == 0:
...         return 1
...     else:
...         value = factorial(n-1)
...         result = n * value
...         return result
...
>>> factorial(4)
24
```

**Aufgabe**

Schreibe eine Funktion, die eine beliebige Ganzzahl (positiver integer) so lange durch zwei teilt und das Ergebnis ausgibt, bis dies nicht mehr möglich ist.

## 2.8 Globale und Lokale Variablen

### Allgemeines

Es muss bei Funktionen zwischen lokalen und globalen Variablen unterschieden werden. Lokale Variablen haben nur innerhalb einer Funktion ihren Gültigkeitsbereich. Globale Variablen hingegen gelten auch außerhalb der Funktion. Will man mit Hilfe einer Funktion eine Variable, die global, also außerhalb der Funktion deklariert wurde, verändern, so muss man ihr das *Keyword* **global** voranstellen. Dies sollte man jedoch nach Möglichkeit vermeiden, da dies schnell zu Fehlern im Programmablauf führen kann. Lokale und globale Variablen sollten nach Möglichkeit getrennt werden.

### Beispiel

```
>>> s = 'globaler string'
>>> def f1():
...     print s
...
>>> def f2():
...     s = 'lokaler string'
...     print s
...
>>> f1()
globaler string
>>> f2()
lokaler string
>>> def f3():
...     global s
...     s = 'lokaler string'
...     print s
>>> s = 'globaler string'
>>> f3()
lokaler string
>>> print s
lokaler string
```

## 2.9 Lokale Funktionen

### Allgemeines

Funktionen können selbst Funktionen enthalten. Diese sind dann lokal definiert und haben nur im Funktionskörper selbst ihre Gültigkeit.

### Beispiel

```
>>> def digitSum(*numbers):
...     """
...     Aus: Weigend (2008: 153f)
...     """
...     def dsum(number):
...         # turn number into a list of digits
...         digits = list(str(number))
...         sum = 0
...         for digit in digits:
...             sum += int(digit)
...         return sum
...     sum = 0
...     for number in numbers:
...         sum += dsum(number)
...     return sum
...
>>> digitSum(1,12,10)
5
```

### 2.10 Lambda-Formen

#### Allgemeines

Mit Lambda-Formen können kleine anonyme Funktionen definiert werden. (Weigend 2008:158)

#### Beispiel

```
>>> [i for i in range(5)]
[0, 1, 2, 3, 4, 5]
>>> list(str(12))
['1', '2']
>>> [int(i) for i in list(str(12))]
[1, 2]
>>> sum([1,2])
3
>>> f = lambda x: sum([i for i in list(str(x))])
>>> f(20)
2
>>> f(1)
1
>>> f(222)
6
```

## 3 Übung

Gegeben ist das Programm <kaskade4.py>, welches eine Erweiterung der vorherigen Versionen des beliebten und berühmten Jongliersimulators darstellt. Schau Dir das Skript an und versuche, die folgenden Fragen zu beantworten:

- Wie funktioniert das Programm allgemein (wie lässt es sich aufrufen, welche Parameter kann man verwenden, etc.)?
- Welche Funktion haben die Zeilen 14 und 15?
- Handelt es sich bei der cascade-Funktion um eine Prozedur?
- Wie lässt sich das Programm verändern, wenn man es dem Nutzer ermöglichen möchte, die Anzahl der Durchgänge zu bestimmen?

## Literatur

Weigend, Michael. 2008. *Python ge-packt*. Heidelberg: mitP, 4 edition.

# Sequenzen (1)

## 1 Was sind Sequenzen?

- Eine Sequenz ist eine geordnete Liste von Objekten.
- Im Gegensatz zu Mengen, die eine Sammlung von ungeordneten Objekten darstellt, die jeweils nur einmal vorkommen können, können dieselben Objekte in Listen mehrmals auftauchen.
- Die Distinktivität der Objekte von Sequenzen wird durch ihre Ordnung gesichert: Jedes Objekt wird definiert durch seine Gestalt und seine Position.
- Viele Objekte unseres täglichen Lebens können als Sequenzen modelliert werden: Musik stellt Sequenzen von Schallwellen dar, Filme sind Sequenzen von Bildern und Rezepte sind Sequenzen von Instruktionen.

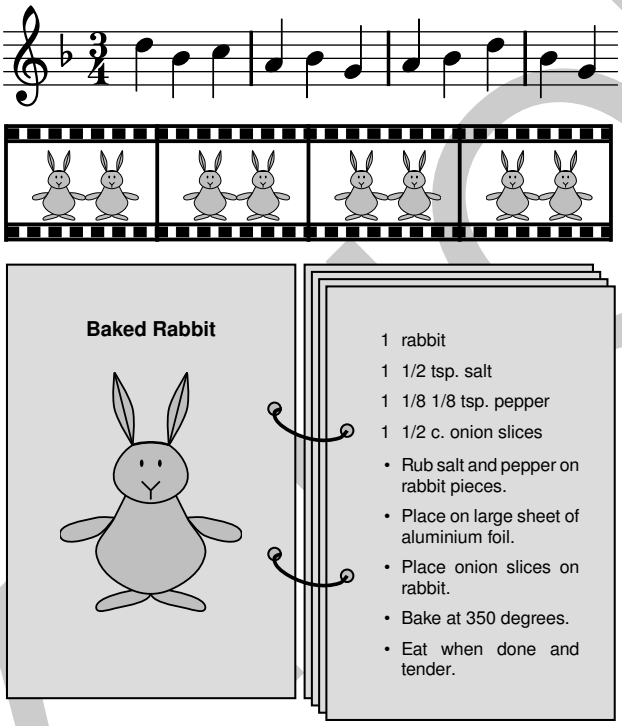



Abbildung 1: Musik, Filme, Rezepte: Sequenzen in unserem täglichen Leben

## 2 Allgemeines zu sequentiellen Datentypen in Python

- In Python gibt es eine Reihe verschiedener sequentieller Datentypen.
- Die wichtigsten Datentypen sind dabei:
  - Listen (<type 'list'>), die in der Form [a, b, …, z] deklariert werden und *veränderbare Sequenzen von Objekten* darstellen. Objekte, die keine Listen sind, können mit Hilfe der Funktion list(s) in Listen umgeformt werden.
  - Strings (<type 'str'> und <type 'unicode'>), die *unveränderbare Sequenzen von Zeichen* darstellen, und durch einfache, doppelte, dreifache einfache oder dreifache doppelte Anführungszeichen deklariert werden. Objekte, die keine Strings sind, können mit Hilfe der Funktionen str(s) oder unicode(s) in Strings umgeformt werden.
  - Tuples (<type 'tuple'>), die in der Form (a, b, …, z) deklariert werden und *unveränderbare Sequenzen von Objekten* darstellen. Objekte, die keine Tuple sind, können mit Hilfe der Funktion tuples(s) in Tuple umgeformt werden.
- Alle diese Datentypen haben gemein, dass sie sequentiell organisiert sind, dass sie also eine geordnete Struktur aufweisen.
- Für alle Datentypen sind ebenfalls einheitliche Operationen und Funktionen definiert, die im Folgenden genauer behandelt werden.

## 2.1 Allgemeine Sequenzoperationen

### Allgemeines

| Operation | Resultat |
|---|---|
| x in s | **True** oder **False** |
| x not in s | **True** oder **False** |
| len(s) | Länge von s |
| s + t | Konkatenation von s und t |
| s * n | Konkatenation von n flachen Kopien von s |
| min(s) | kleinstes Element von s |
| max(s) | größtes Element von s |
| s.index(i) | index des ersten Auftauchens von i in s |
| s.count(i) | Anzahl des Auftauchens von i in s |

### Beispiele

```
>>> s = 'blablabla'
>>> 'b' in s
True
>>> 'c' not in s
True
>>> len(s)
9
>>> s + s
'blablablablablabla'
>>> 2 * s
'blablablablablabla'
>>> min(s)
'a'
>>> max(s)
'l'
>>> s.index('b')
0
>>> s.count('b')
3
```

## 2.2 Indizierung

### Allgemeines

- Die Indizierung ermöglicht es, einzelne Elemente von Sequenzen ausgeben zu lassen.
- Welches Element ausgegeben wird, wird dabei durch die Position bestimmt.
- Bei der Indizierung fügt man allgemein den Index in eckigen Klammern an die Sequenzvariable an, also in der Form **s[i]**.
- Indizes müssen selbstverständlich Ganzzahlen sein.
- Die Zählung der Elemente beginnt in Python immer mit 0!
- Wählt man einen negativen Index, wird umgekehrt, also vom Ende der Sequenz her indiziert.

### Beispiele

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

```
3
>>> s[10]
Traceback (most recent call last)
IndexError: list index out of range
>>> s[-1]
6
>>> s[s.index(2)]
2
```

## 2.3 Slicing

- Das *Slicing* ermöglicht es, einzelne Subsequenzen aus einer Sequenz herauszuschneiden.
- Dabei wird mit Indizes operiert, die in Form eckiger Klammern, getrennt von einem Doppelpunkt an die Sequenzvariable angefügt werden, also in der Form **s[i:j]**.
- Ein Slice einer Sequenz von i nach j ist definiert als die Sequenz von Elementen mit dem Index k, wobei gilt: i <= k < j.
- Wird kein Wert für i oder j angegeben, so wird der Anfang bzw. das Ende der Sequenz als Standardwert genommen.
- Will man zusätzlich festlegen, dass bestimmte Elemente in regelmäßigen Abständen aus einer Sequenz herausgeschnitten werden, so kann man dies tun, indem man einen dritten Wert (ebenfalls durch einen Doppelpunkt separiert) angibt, also in der Form **s[i:j:k]**.
- Ist der Wert für das schrittweise Ausschneiden negativ, so wird umgekehrt, also vom Ende der Sequenz ausgehend, ausgeschnitte.
- Der Rückgabewert der Slice-Operation ist immer *ebenfalls* eine Sequenz, für die alle gängigen Sequenzoperationen *ebenfalls* gelten.

**Beispiele**

```
>>> s = [1,2,3,4,5,6]
s[1:5]
[2, 3, 4, 5]
>>> s[:]
[1, 2, 3, 4, 5, 6]
>>> s[::]
[1, 2, 3, 4, 5, 6]
>>> s[1:]
[2, 3, 4, 5, 6]
>>> s[:4]
[1, 2, 3, 4]
>>> s[-1:]
[6]
>>> s[-4:]
[3, 4, 5, 6]
>>> s[::-1]
[6, 5, 4, 3, 2, 1]
>>> s[::2]
[1, 3, 5]
>>> s[:4] + s[4:]
[1, 2, 3, 4, 5, 6]
```

## 2.4 Iteration

**Allgemeines**

- Alle Sequenzen sind iterierbare Objekte.
- Sequenzen sind ein Untertyp der iterierbaren Datentypen.

- Wenn ein Datentyp iterierbar ist, so heißt das, das Schleifenoperationen (genauer: for-Schleifen) auf ihn angewendet werden können.
- Beim Anwenden von Schleifenoperationen wird eine Variable auf die Elemente einer Sequenz mit Hilfe des Ausdrucks **for variable in sequenz** deklariert, über die dann sukzessive iteriert wird.
- Da Sequenzen geordnete Datentypen sind, beginnt die Iteration über eine Sequenz immer beim ersten Element.
- Die Funktion range(s) wurde bereits angesprochen. Sie erzeugt eine Liste von Integern. Es ist üblich, mit Hilfe dieser Funktion über Sequenzen zu iterieren.
- Die Funktion enumerate(s) erzeugt einen Iterator von Tuples, dessen erstes Element den Index der Sequenz darstellt, während das zweite Element das Element der Sequenz selbst ist. Ein Iterator-Objekt ist keine Liste, er kann bei der Iteration jedoch genauso wie eine Liste verwendet werden. Die Funktion ist ebenfalls sehr praktisch, wenn man über Sequenzen iterieren möchte.

**Beispiele**

```
>>> s = list('HhAaUuSs')
>>> for i in range(len(s)):
...     if i % 2 == 0:
...         print s[i]
...
H
A
U
S
>>> for i in enumerate(s[0:4]): print i
...
(0, 'H')
(1, 'h')
(2, 'A')
(3, 'a')
>>> for i,j in enumerate(s):
...     if i % 2 == 0:
...         print j
...
H
A
U
S
>>> for i in s:
...     j = s.index(i)
...     if j % 2 == 0:
...         print s[j]
...
H
A
U
S
```

### 2.5 Weitere Operationen

## 3 Strings

- Ein String ist ein sequentieller Datentyp, der aus Strings der Länge 1 besteht.
- Im Gegensatz zu Listen sind Strings unveränderbar. Das heißt, dass man, wenn man einen String verändern möchte, immer eine durch Operationen veränderte Kopie des Strings erzeugen muss (bzw. automatisch erzeugt).

### 3.1 Byte-Strings und Unicode-Strings

- Zu unterscheiden sind in Python zwei Typen von Strings: Byte-Strings und Unicode-Strings: 'byte strings contain bytes while Unicode is intended for text' (Bassi 2010:40).
- Unicode-Strings stellen alle einwertigen Zeichen, die in der Unicodedatenbank definiert sind als einwertige Zeichen dar.
- Bytestrings hingegen speichern Zeichen als Byte-Sequenzen ab, was dazu führen kann, dass einzelne Zeichen durch zwei Byte-Characters abgespeichert werden. Dies kann bei Textoperationen zu Problemen führen, weshalb es sich anbietet, in den meisten Fällen mit Unicode-Strings zu arbeiten.
- Strings werden, wie bereits behandelt, mit Hilfe von Anführungszeichen deklariert.
- Beim Deklarieren von Unicode-Strings wird den Anführungszeichen ein **u** vorangestellt (also **u'bla'**).

#### Beispiele

```
>>> a = 'bla'
>>> type(a)
<type 'str'>
>>> a = u'bla'
>>> type(a)
<type 'unicode'>
>>> a.encode('utf-8')
'bla'
>>> a = a.encode('utf-8')
>>> type(a)
<type 'str'>
>>> a = a.decode('utf-8')
>>> type(a)
<type 'unicode'>
>>> a = u'\u03b8'
>>> print a
θ
>>> len(a)
1
>>> a = a.encode('utf-8')
>>> len(a)
2
```

### 3.2 Stringfunktionen

- Es gibt eine große Anzahl von Funktionen und Operationen, die für Strings vordefiniert sind und hier nicht alle explizit behandelt werden können.
- Unter den Stringfunktionen in Python können grob die unterschieden werden, die Wahrheitswerte über Strings zurückliefern, solche, die einen veränderten String zurückgeben, solche, die einen formatierten String zurückgeben und solche, die andere Objekte als Rückgabewerte enthalten.
- Es sollte noch einmal betont werden, dass alle Funktionen und Operationen, die auf Strings angewendet werden, nie den String, der als Eingabewert dient, selbst zurückliefern, da Strings unveränderbare Objekte sind (im Gegensatz zu Listen).

### 3.3 Funktionen, die Wahrheitswerte liefern

#### Allgemeines

| Operation | Resultat |
|---|---|
| s.startswith(x) | **True**, wenn s mit x beginnt, sonst **False** |
| s.endswith(x) | **True**, wenn s mit x endet, sonst **False** |
| s.isalpha() | **True**, wenn s alphabetisch ist, sonst **False** |
| s.isdigit() | **True**, wenn s nur aus Zahlen besteht, sonst **False** |

#### 3.3.1 Beispiele

```
>>> a = 'bla'
>>> a.startswith('b')
True
>>> a.endswith('b')
False
>>> a.endswith('a')
True
>>> a.isalpha()
True
>>> a = '012'
>>> a.isdigit()
True
```

### 3.4 Funktionen, die veränderte Strings liefern

**Allgemeines**

| Operation | Resultat |
|---|---|
| s.lower() | Liefert einen String, in dem alle Großbuchstaben in s durch Kleinbuchstaben ersetzt wurden |
| s.upper() | Liefert einen String, in dem alle Kleinbuchstaben in s durch Großbuchstaben ersetzt wurden |
| s.swapcase() | Liefert einen String, in dem alle Großbuchstaben mit und Kleinbuchstaben vertauscht wurden |
| s.replace(x,y) | Liefert einen String, in dem alle x in s durch y ersetzt wurden |
| s.find(x) | Liefert den Index des ersten Aufkommens von x in s |

**Beispiele**

```
>>> a = 'string'
>>> a.isalpha()
True
>>> a.upper()
'STRING'
>>> a = a.upper()
>>> a
'STRING'
>>> a.lower()
'string'
>>> a = '0123'
>>> a.isdigit()
True
>>> a.find(0)
0
>>> a.replace('0','1')
'1123'
```

### 3.5 Funktionen, die formatierte Strings zurückgeben

**Allgemeines**

| Operation | Resultat |
|---|---|
| s.strip([x]) | Liefert einen String, in dem alle x zu Beginn und zu Ende des Strings entfernt wurden. Wenn kein Argument angegeben wird, so wird Whitespace entfernt |
| s.center(i,[fillchar]) | Liefert einen String der Länge i, der s als Zentrum hat. Die übrigen Zeichen können durch fillchar vorgegeben werden, ansonsten wird mit Leerzeichen aufgefüllt. |
| s.format(*args, **kwargs) | Eine komplexe Funktion mit eigener spezifischer Syntax, die das Formatieren von Strings erlaubt und in diesem Zusammenhang nicht eingehender beleuchtet werden kann |

**Beispiele**

```
>>> a = '   STRING   '
>>> a
'   STRING   '
>>> a = a.strip()
>>> a
'STRING'
>>> a.center(20)
'       STRING       '
>>> len(a.center(20)
... )
20
>>> a = "Das Ergebnis von {0} + {1} ist {2}."
>>> a.format(2,4,2+4)
'Das Ergebnis von 2 + 4 ist 6.'
```

### 3.6 Funktionen, die andere Objekte als Strings liefern

| Operation | Resultat |
|---|---|
| s.split(x) | Liefert eine Liste von Strings, indem s an jeder Stelle, wo x auftaucht, gespalten wird |
| s.splitlines() | Liefert eine Liste von Strings, indem s an allen Stellen, wo ein Newline-Zeichen auftaucht, gespalten wird. |
| s.partition(x) | Liefert ein ähnliches Resultat wie s.split(), mit dem Unterschied, dass das Trennzeichen x auch in der Liste auftaucht. |

**Beispiele**

```
>>> a = 'Ich habe gerade irgendwie keinen Bock mehr zu arbeiten.'
>>> b = a.split(' ')
>>> for word in b: print word
...
Ich
habe
gerade
irgendwie
keinen
Bock
mehr
zu
arbeiten.
>>> b = a.partition('g')
```

```
>>> for irgendwas in b: print irgendwas
...
Ich habe
g
erade irgendwie keinen Bock mehr zu arbeiten.
>>> a = """
... Ich bin irgendwie muede.
... Aber ich muss noch arbeiten.
... Mann ist das bloede!
... """
>>> b = a.splitlines()
>>> for sentence in b: print sentence
...

Ich bin irgendwie muede.
Aber ich muss noch arbeiten.
Mann ist das bloede!
```

## 4 Übung

Es wäre irgendwie praktisch, eine erweiterte Index-Funktion in Python zu haben, die nicht nur den ersten, sondern alle Indizes für einen bestimmten Substring liefert. Das erstellen einer solchen Funktion dürfte eigentlich gar nicht so schwer sein, denn die index-Funktion in Python erlaubt es ja auch, nur einen Teilstring zu untersuchen, indem man die Indizes dieses Teilstrings als Parameter angibt. Man kann also im String s = "blablabla" nur das "bla" in der Mitte, welches durch die Indizes (3,6) erfasst werden kann, auf einen bestimmten Substring untersuchen, für den dann der richtige Index geliefert wird. Das mittlere "l" könnte man dann mit s.index('l',3,6) erreichen. Die Werte für die Indizes müssten dann noch irgendwo gespeichert werden. Dafür böte sich eine Liste an, die man mit dem Kommando l.append(stuff) ja praktisch erweitern kann.

- Schreibe eine Funktion, die alle Indizes für einen bestimmten Substring in einer Liste zurückliefert.

## Literatur

Bassi, Sebastian. 2010. *Python for bioinformatics*. Boca Raton and London and New York: CRC Press.

# Sequenzen (2)

## 1 Allgemeines zu Listen

> Lists are one of the most versatile object types in Python. A list is an ordered collection of objects. It is represented by elements separated by commas and enclosed between square brackets. (Bassi 2010:44)

## 2 Erstellen von Listen (allgemein)

- Definieren von Listen als Variable
- Erzeugen von Listen mit Hilfe von Funktionen
- Erstellen von Listen mit Hilfe der *Listenkomprehension*

```
1  >>> a = [1,2,3,4,5]
2  >>> a
3  [1, 2, 3, 4, 5]
4  >>> type(a)
5  <type 'list'>
6  >>> a = range(1,6)
7  >>> a
8  [1, 2, 3, 4, 5]
9  >>> type(a)
10 <type 'list'>
11 >>> a = [i for i in range(1,6]
12 >>> a
13 [1, 2, 3, 4, 5]
14 >>> type(a)
15 <type 'list'>
```

## 3 Erstellen von Listen mit Hilfe der Listenkomprehension

- Die Listenkomprehension ermöglicht es, Listen flexibel mit Hilfe von allen möglichen Funktionen zu erstellen, welche Sequenzen zurückgeben.
- Eine Liste wird somit mit Hilfe einer anderen Liste erzeugt.
- Eine Liste wird dabei auf die gleiche Weise erzeugt, wie auch die for-Schleife in Python Operationen reihenweise ausführt: die 'Formel' zur Erstellung von Listen lautet dabei [ x **for** x **in** s ], wobei x auf die Variable verweist, mit der die Liste sukzessive gefüllt werden soll, während s die Sequenz darstellt, über deren Werte iteriert wird.
- Die Listenkomprehension ist eine *der* Spezialitäten von Python und ermöglicht es in vielen Fällen, schnelle, intuitiv verständliche Kodeschnipsel zu verfassen.

```
>>> A = [0,1,2,3,4,5,6,7,8,9,10]
>>> [i for i in A]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> [2 * i for i in A]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
>>> [2 * i for i in A if i % 2 == 0]
[0,2,4,6,7,10]
>>> [i ** 2 for i in A]
[1, 1, 4, 27, 256, 3125, 46656, 823543, 16777216, 387420489, 10000000000L]
>>> from random import randint
>>> A = [randint(0,10) for i in range(10)]
>>> B = [randint(0,10) for i in range(10)]
>>> A
[10, 1, 9, 1, 7, 10, 0, 2, 3, 0]
>>> B
[6, 5, 3, 2, 1, 0, 5, 9, 6, 4]
>>> [i for i in A if i in B]
[1, 9, 1, 0, 2, 3, 0]
>>> [i for i in A if i not in B]
[10, 7, 10]
```

## 4 Allgemeine Listen-Funktionen

| Operation | Resultat |
|---|---|
| s.reverse( ) | Kehrt die Reihenfolge der Listenelemente um |
| s.append(x) | Fügt das Element x an s an |
| s.extend(y) | Fügt die Sequenz y an s an |
| s.insert(i,x) | Fügt das Element x an Stelle i in s ein |
| s.pop([i]) | Liefert und entfernt das i-te Element von s. Wenn i nicht definiert ist, wird das letzte Element entfernt und ausgegeben |
| sorted(s,key=k) | Liefert eine Liste, die entsprechend dem Key k sortiert wurde |
| sum(s) | Liefert die Summe aller Elemente in s, falls s nur Zahlen enthält |
| x.join(s) | Liefert einen String, der die Elemente von s getrennt durch den Separator x enthält. Hierbei müssen alle Elemente von s ebenfallst Strings sein. |

```
>>> A = [randint(0,20) for i in range(20)]
>>> A
[10, 9, 15, 3, 18, 4, 11, 16, 18, 5, 0, 4, 12, 18, 3, 4, 6, 12, 20, 4]
>>> B = [i for i in B if B.count(i) == 1]
>>> B
[10, 9, 15, 11, 16, 5, 0, 6, 20]
>>> B.reverse()
>>> B
[20, 6, 0, 5, 16, 11, 15, 9, 10]
>>> B.reverse()
>>> B
[10, 9, 15, 11, 16, 5, 0, 6, 20]
```

```
>>> B.append(40)
>>> B
[10, 9, 15, 11, 16, 5, 0, 6, 20, 40]
>>> B.extend([100,20])
>>> B
[10, 9, 15, 11, 16, 5, 0, 6, 20, 40, 100, 20]
>>> B.insert(1,'bla')
>>> B
[10, 'bla', 9, 15, 11, 16, 5, 0, 6, 20, 40, 100, 20]
>>> B.pop(1)
'bla'
>>> sorted(B)
[0, 5, 6, 9, 10, 11, 15, 16, 20, 20, 40, 100]
>>> sum(B)
252
```

### 4.1 Arbeiten mit Listenelementen

- Im Gegensatz zu Strings sind Listen *veränderbare* Objekte.
- Das bedeutet, das die Elemente von Listen flexibel verändert werden können.
- Dies ist eine wichtige Eigenschaft von Listen, die sie zu einem flexiblen und leicht zu handhabenden Tool in der Pythonprogrammierung macht.
- Gleichzeitig gilt es aber immer auch zu beachten, dass Listen tatsächlich veränderbar sind, da Fehler oftmals dadurch entstehen, dass dies Vergessen wird.

```
>>> s = 'ich trinke abends gerne apfelschohrle'.split(' ')
>>> s
['ich', 'trinke', 'abends', 'gerne', 'apfelschohrle']
>>> s[1]
'trinke'
>>> s[4]
'apfelschohrle'
>>> s[4].replace('schohrle','schorle')
'apfelschorle'
>>> s[4]
'apfelschohrle'
>>> s[4] = s[4].replace('schohrle','schorle')
>>> s[4]
'apfelschorle'
>>> s[0],s[1] = 'trinkst','du'
>>> s
['trinkst', 'du', 'abends', 'gerne', 'apfelschorle']
>>> s[0],s[1] = s[1],s[0]
>>> s
['du', 'trinkst', 'abends', 'gerne', 'apfelschorle']
>>> print ' '.join(s)
```

```
du trinkst abends gerne apfelschorle
>>> print 'http://www.'+'/'.join(s)+'.de'
http://www.du/trinkst/abends/gerne/apfelschorle.de
```

### 4.2 Verschachtelte Listen

- Listen können als Objekte selbst wiederum Listen enthalten.
- Sie eignens ich damit hervorragend für Operationen, die auf zweidimensionalen Datenstrukturen basieren.
- In Bezug auf verschachtelte Listen muss man jedoch vorsichtig sein, wenn man mit Kopien von Listen arbeitet: Da Python vorrangig flache Kopien von Listen erstellt, können hier dumme Fehler passieren, wenn man Listen nicht richtig deklariert.

```
>>> a = [i for i in '1','2']
>>> b = [i for i in a]
>>> b
['1', '2']
>>> b[1] = 10
>>> a
['1', '2']
>>> b
['1', 10]
>>> c = a
>>> c[1] = 10
>>> a[1]
10
>>> d = [a,b,c]
>>> d
[['1', 10], ['1', 2], ['1', 10]]
>>> d[1]
['1', 2]
>>> d[1][1]
2
>>> a = [i for i in b]
>>> a.append(a)
>>> a
['1', 2, [...]]
>>> a[1]
2
>>> a[2]
['1', 2, [...]]
>>> a[2][2]
['1', 2, [...]]
>>> a[2][2][2]
['1', 2, [...]]
```

# 5 Anwendungsbeispiel

- Auf einer Party treffen sich viele verschiedene Menschentypen.
- Der eine spielt in seiner Freizeit Schach, der andere hat einen Faible für Meerschweinchen, und wieder einer studiert Linguistik.
- Wir wollen für fünf Personen berechnen, wie ähnlich sie einander sind, und die Ergebnisse in einer Ähnlichkeitsmatrix anlegen.
- Das Ganze soll auf einer Liste von Merkmalen beruhen, durch die Personen definiert werden können.
- Eine Person weist entweder ein bestimmtes Merkmal auf, oder nicht.
- Basierend auf der Anzahl an gemeinsamen vorhandenen Merkmalen lässt sich somit eine prozentuale Ähnlichkeit zwischen Personen errechnen.

## 5.1 Vorüberlegung

- Um die Daten ordentlich zu generieren, importieren wir zunächst die Funktionen **randint** und **sample** aus dem random-Module.
- **randint** erzeugt einen Integer für einen gewählten Bereich. **randint(1,10)** liefert bspw. eine beliebige Zahl zwischen 1 und 10.
- **sample** liefert eine Liste von zufällig aus einer anderen Liste gewählten Werten, deren Länge im Voraus festgelegt wird. **sample([0,1,2,3,4],2)** kann beispielsweise **[1,3]** liefern.

```
1 from random import randint,sample
```

## 5.2 Die Charakteristika

- Wir legen die Charakteristika als einfache Liste von Eigenschaften ab.
- Die Eigenschaften werden als Strings ausgedrückt.

```
 9 # create a list of characteristics for persons
10 characteristics = [
11         'blue eyes',
12         'brown hair',
13         'has child',
14         'is pregnant',
15         'likes football',
16         'is a linguist',
17         'is intelligent',
18         'likes chess',
19         'is a nerd',
20         'likes python'
21         ]
```

### 5.3 Erstellen von Personen

- Die Personen werden als eine Liste von Indizes erstellt.
- Wenn eine Person einen bestimmten Index aufweist, so wird ihr die Eigenschaft zugewiesen, die in der Liste **characteristics** an dieser Stelle definiert ist.
- Die Liste wird mit Hilfe der **sample**-Funktion erzeugt. Die Länge der Liste wird mit Hilfe von **randint** festgelegt.
- Dadurch können Personen eine Reihe verschiedener Eigenschaften haben.
- Mit Hilfe der Listenkomprehension wird dann eine Reihe von Personen erzeugt.

```
23 # define a function which returns a person by defining its characteristics
24 def make_person():
25     """
26     Function returns a random list of indices in range(0,10). Each index
27     defines the presence of a characteristic, the absence of an index
          defines
28     the absence.
29     """
30     return sorted(sample(range(10),randint(0,10)))
31
32 # create the persons using the function and list comprehension
33 persons = [make_person() for i in range(5)]
```

### 5.4 Erstellen der Ähnlichkeitsmatrix

- Um die Ähnlichkeitsmatrix zu erstellen, müssen wir diese zweidimensional anlegen, also eine Liste, die wiederum Listen enthält.
- Die Länge der Matrix leitet sich aus der Länge der Personen ab, die wir erzeugt haben, in unserem Falle also 5, was eine $5 \times 5$ Matrix ergibt.
- Über diese Matrix iterieren wir mit zwei for-Schleifen.
- Die durch die for-Schleifen inkrementierten Variablen **i** und **j** werden als Index für die Personenliste verwendet.
- Mit Hilfe der Listenkomprehension extrahieren wir alle Elemente, die in beiden Personenlisten auftauchen. Deren Länge, geteilt durch die Gesamtanzahl an Merkmalen (10) ergibt die Ähnlichkeit zwischen den Personen.
- Dieser Wert wird an der entsprechenden Stelle in der Matrix eingefügt.

```
35 # create a matrix which will store all values for the similarities between
      the
36 # persons
37 matrix = [[1 for i in range(5)] for j in range(5)]
38
```

```
39 # iterate over the matrix, thereby calculating the pairwise similarities
40 # between all persons
41 for i in range(len(matrix)):
42     for j in range(len(matrix[i])):
43         # we only need to calculate for i < j, since all other values are
44         # either zero or symmetric
45         if i < j:
46             # we simply calculate, how many characteristics two persons have
                   in
47             # common by using list comprehension
48             commons = [k for k in persons[i] if k in persons[j]]
49             # we now calculate the similarity by dividing the commons by the
50             # total number of possible commons (which is 10)
51             similarity = len(commons) / 10.0
52             # we now assign the similarity to matrix[i][j] and matrix[j][i],
53             # since the matrix should be symmetric
54             matrix[i][j],matrix[j][i] = similarity,similarity
```

### 5.5 Ausdrucken der kalkulierten Daten

- Zum Ausdrucken der Daten erzeugen wir einen String, den wir iterativ um neue Daten und Zahlen ergänzen.
- Um genauer zu verstehen, wie die Daten sukzessive an den String angefügt werden, kann man zum "Debuggen" die jeweiligen Teilergebnisse durch einfaches Einfügen eines Print-Kommandos in Zeile 59 anzeigen lassen.
- Die Reihe der Matrix wird durch den join-Befehl erzeugt. Hierbei muss aber vorher der Inhalt der Zellen in strings umgewandelt werden, weshalb wieder auf die Listenkomprehension zurückgegriffen wird.

```
56 # print out the matrix to the screen
57 outstring = '  \tP1\tP2\tP3\tP4\tP5\n'
58 for i in range(len(matrix)):
59     # if you want to see, what this block does, insert a print statement at
         any
60     # point of the block
61     outstring += 'P'+str(i+1)+'\t'
62     outstring += '\t'.join([str(k)[0:5] for k in matrix[i]]) + '\n'
63
64 print outstring
```

### 5.6 Ausdrucken der Ähnlichkeiten

- Das Ausdrucken der Ähnlichkeiten geschieht ähnlich wie das Ausdrucken der Matrix.
- Wieder wird ein String sukzessive erzeugt.
- Nur wird diesmal über die Liste der Ähnlichkeitsmerkmale iteriert.

- Eine if-Bedingung checkt, ob eine Ähnlichkeit für die jeweilige Person vorliegt.
- Wenn nicht, wird der String um ein Minus erweitert, wenn doch, um ein Plus.

```
66 # we now print out what the characteristics of each person are
67 outstring = 'characteristic\tP1\tP2\tP3\tP4\tP5\n'
68 for i in range(len(characteristics)):
69     outstring += characteristics[i]
70     # iterate over persons
71     for j in range(len(persons)):
72         if i in persons[j]:
73             outstring += '\t-'
74         else:
75             outstring += '\t+'
76     outstring += '\n'
77
78 print outstring
```

## Literatur

Bassi, Sebastian. 2010. *Python for bioinformatics*. Boca Raton and London and New York: CRC Press.

# Reguläre Ausdrücke

## 1 Allgemeines zu regulären Ausdrücken

### 1.1 Das obligatorische Wiki-Zitat

> In der Informatik ist ein regulärer Ausdruck (engl. regular expression, Abk. RegExp oder Regex) eine Zeichenkette, die der Beschreibung von Mengen beziehungsweise Untermengen von Zeichenketten mit Hilfe bestimmter syntaktischer Regeln dient. Reguläre Ausdrücke finden vor allem in der Softwareentwicklung Verwendung; für fast alle Programmiersprachen existieren Implementierungen. (aus: `http://de.wikipedia.org/wiki/Regul%C3%A4rer_Ausdruck`)

### 1.2 Informelle Zusammenfassung

- Reguläre Ausdrücke stellen konkret die Möglichkeit dar, einfache Such- oder Such-Ersetz-Funktionen um die Suche nach *Mustern* zu erweitern.
- Während die bekannte **replace**-Funktion in Python bspw. nur die Möglichkeit bietet, einen bestimmten Substring in einer Zeichenkette zu finden und zu ersetzen, kann mit Hilfe regulärer Ausdrücke eine ganze Klasse von Substrings gesucht und durch eine ander Klasse ersetzt werden.
- Man kann beispielsweise alle Wörter suchen, die mit Großbuchstaben anfangen.
- Genauso kann man alle Wörter suchen, die eine bestimmte Länge aufweisen.
- Oder man kann alle Wörter suchen, die mit einem Großbuchstaben beginnen, und sie durch Wörter ersetzen, in denen Groß- mit Kleinbuchstaben vertauscht wurden.
- Reguläre Ausdrücke sind das grundlegende Handwerkszeug zur automatischen Textanalyse.

### 1.3 Zeichen und Zeichenklassen

- Das Grundprinzip der regulären Ausdrücke besteht im Zusammenfassen von Zeichen zu Zeichenklassen.
- Durch das Zusammenfassen werden bei Such- und Such-Ersetz-Ausdrücken nicht nur die Zeichen in ihrer wörtlichen Bedeutung, sondern als abstrakte Klassen angesprochen.
- Während der reguläre Ausdruck ‘a’ bspw. alle Zeichen matcht, die ein (wörtliches) ‘a’ darstellen, bezeichnet der reguläre Ausdruck ‘[0-9]’ alle Ziffern von $0$ bis $9$.
- Da man auch zur Bezeichnung der Klassen in regulären Ausdrücken zwangsläufig auf Zeichen zurückgreifen muss, haben bestimmte Zeichen in regulären Ausdrücken einen anderen Wert, als der, den sie im wörtlichen Sinne hätten.
- Zur Trennung der Bedeutung von wörtlichen und abstrakten Werten in bestimmten “ambigen” Zeichen wird meist der Backslash (`\`) verwendet. Während das Zeichen ‘`.`’ als abstraktes Zeichen in regulären Ausdrücken die Klasse aller Zeichen darstellt, also auf jedes Zeichen verweist, stellt die Kombination aus Backslash und Punkt ‘`\.`’ den nicht-abstrakten, wörtlichen Punkt dar.

- In Bezug auf die Trennung abstrakter und wörtlicher Zeichen unterscheiden sich Programmiersprachen oftmals geringfügig. Es bietet sich also an, sich im Zusammenhang mit regulären Ausdrücken stets mit den besonderen Konventionen der jeweiligen Sprachen vertraut zu machen.

# 2 Reguläre Ausdrücke in Python

## 2.1 Das Modul re

- Alle regulären Ausdrücke werden in Python mit Hilfe des **re**-Moduls realisiert.
- Vor der Verwendung von regulären Ausdrücken muss also immer der Befehl **from re import *** eingegeben werden.

## 2.2 Literale und Klassenzeichen in Python

| Zeichen | Bedeutung | Zeichen | Bedeutung |
|---|---|---|---|
| `\.` | literaler Punkt | `.` | matcht jedes einzelne Zeichen, außer dem Newline-Zeichen |
| `\*` | literaler Stern | `*` | matcht keine oder mehr Wiederholungen desselben Zeichens oder Ausdrucks |
| `\+` | literales Plus | `+` | matcht eine oder mehr Wiederholungen desselben Zeichens oder Ausdrucks |
| `\$` | literales Dollarzeichen | `$` | matcht das Ende einer Zeichenkette oder den leeren String vor einem Newline-Zeichen |
| `\^` | literales hohes Bogenzeichen | `^` | matcht den Anfang einer Zeichenkette, negiert einen Ausdruck, wenn dieser in eckigen Klammern steht |
| `\n` | Newlinezeichen | `n` | literales n |
| `\( \)` | literale Klammern | `( )` | matcht alle Ausdrücke, die innerhalb der Klammern sind (wichtig für Backreferenzen) |
| `\[ \]` | literale eckige Klammern | `[ ]` | markieren Zeichenklassen |
| `\|` | literale Pipe | `\|` | markiert ein logisches Oder, welches zwei reguläre Ausdrücke miteinander verbinden kann |
| `\?` | literales Fragezeichen | `?` | matcht $0$ oder $1$ Wiederholung des vorangehenden Ausdrucks |
| `\d` | matcht alle Ziffern von 0 bis 9 | `d` | literales ‘d’ |
| `\s` | matcht alle Whitespace-Zeichen | `s` | literales ‘s’ |

```
1 >>> from re import sub,findall
2 >>> mystring = "habe nun\tach\nJuristerei"
3 >>> print mystring
4 habe nun    ach
5 Juristerei
6 >>> findall('a.',mystring)
```

```
['ab', 'ac']
>>> findall('.*e',mystring)
['habe', 'Juristere']
>>> findall('a*e',mystring)
['e', 'e', 'e']
>>> findall('b*e',mystring)
['be', 'e', 'e']
>>> findall('b+e',mystring)
['be']
findall('.+$',mystring)
['Juristerei']
>>> findall('^.....',mystring)
['habe ']
>>> findall('...\n',mystring)
['ach\n']
>>> findall('.u.|.a.',mystring)
['hab', 'nun', '\tac', 'Jur']
>>> findall('.[aeiou]',mystring)
['ha', 'be', 'nu', '\ta', 'Ju', 'ri', 'te', 're']
>>> findall('[aeiou].*?[aeiou]',mystring)
['abe', 'un\ta', 'uri', 'ere']
>>> findall('[aeiou].*[aeiou]',mystring)
['abe nun\ta', 'uristerei']
>>> from re import DOTALL
>>> findall('[aeiou].*[aeiou]',mystring,DOTALL)
['abe nun\tach\nJuristerei']
>>> findall('[^\s]+',mystring)
['habe', 'nun', 'ach', 'Juristerei']
```

### 2.3 Die sub-Funktion

- Mit Hilfe der Funktion **sub** lassen sich Such-Ersetz-Operationen mit Hilfe regulärer Ausdrücke ausführen.
- Die Syntax lautet dabe **sub(source,target,string)**.
- Ausgegeben wird ein neuer String, der entsprechend der Anweisungen verändert wurde.

```
>>> mystring = "Tina Turner"
>>> sub('[iaue]','a',mystring)
'Tana Tarnar'
>>> sub('i','a',mystring)
'Tana Turner'
```

### 2.4 Die findall-Funktion

- Mit Hilfe der Funktion **findall** werden alle Matches eines regulären Ausdrucks in einem Text gesucht und in einer Liste angelegt.

- Die Syntax lautet dabei **findall(regex,string)**.
- Wenn man komplexere reguläre Ausdrücke zugrunde legt, lassen sich mit der **findall**-Funktion bequem ganze Texte oder Teile von Texten in bestimmte Bestandteile zerlegen.

```
>>> from re import findall
>>> text = "Dies ist ein simpler Demonstrationstext."
>>> words = findall('[a-zA-Z]+',text)
>>> for word in words: print word
...
Dies
ist
ein
simpler
Demonstrationstext
```

## 3 Zeichenklassen

- Mit Hilfe der eckigen Klammern werden Zeichenklassen definiert.
- Man kann die Zeichen, die zu einer Klasse gehören sollen dabei explizit auflisten. '[aeiou]' verweist bspw. auf alle kleingeschriebenen 'normalen' Vokale.
- Gleichzeitig lassen sich auch bestimmte Bereiche von Zeichen definieren. So verweist '[a-z]' bspw. auf alle kleingeschriebenen alphabetischen Zeichen, '[0-9]' verweist auf alle Ziffern von 0 bis 9.

```
>>> text = "Beliebte Getränke: \n1: Bier\n2: Wasser\n3: Apfelsaft"
>>> print text
Beliebte Getränke:
1: Bier
2: Wasser
3: Apfelsaft
>>> text = sub('[0-9]:','- ',text)
>>> print text
Beliebte Getränke:
-  Bier
-  Wasser
-  Apfelsaft
```

## 4 Quantoren

- Als Quantoren bezeichnet man die Ausdrücke, welche Angaben über die Anzahl an Matches machen, die gewünscht ist.
- Der einfache Stern-Quantor, bswp. matcht alle Ausdrücke die kein oder mehrmals vorkommen.
- Der Plusquantor matcht alle Ausdrücke, die ein- oder mehrmals vorkommen.

- Schließlich lässt sich noch mit Hilfe eckiger Klammern, die dem Ausdruck nachgestellt werden, ein expliziter Zahlenbereich angeben, in dem gematcht werden soll.

```
>>> text = "Auf der Schifffahrt habe ich mich verfahren."
>>> findall('f*.',text)
['A', 'u', 'f ', 'd', 'e', 'r', ' ', 'S', 'c', 'h', 'i', 'fffa', 'h', 'r', '
   t', ' ', 'h', 'a', 'b', 'e', ' ', 'i', 'c', 'h', ' ', 'm', 'i', 'c', 'h',
    ' ', 'v', 'e', 'r', 'fa', 'h', 'r', 'e', 'n', '.']
>>> findall('f+.',text)
['f ', 'fffa', 'fa']
>>> findall('f{3}.',text)
['fffa']
```

## 5 Rückwärtsreferenzen

- Mit Hilfe von Backreferenzen lassen sich Ausdrücke während der Bearbeitung wieder aufnehmen.
- Dadurch ist es möglich, einen Ausdruck oder Teilausdruck durch sich selbst zu ersetzen.
- Ausdrücke, auf die später zugegriffen werden soll, werden in Python in normale Klammern geschrieben.
- Die Backreferenz wird mit Hilfe des Backslash, dem eine Nummer nachgestellt wird, realisiert.
- Die Nummer gibt dabei an, um welche Backreferenz es sich handelt.
- Verwendet man Backreferenzen mit der **sub**-Funktion, so sollte der Ersetzungsausdruck immer als raw-string markiert werden, indem den Anführungszeichen ein kleines ‘r’ vorangestellt wird (**r’raw’**). Auf diese Weise verhindert man, dass der Backslash falsch interpretiert wird.

```
>>> text = "Beliebte Getränke: \n1: Bier\n2: Wasser\n3: Apfelsaft"
>>> text = sub('([0-9]): ([a-zA-Z]+)',r'\2 (\1)',text)
>>> print text
Beliebte Getränke:
Bier (1)
Wasser (2)
Apfelsaft (3)
```

## 6 Ein Anwendungsbeispiel

### 6.1 Anfang

```
#! /usr/bin/env python
# *-* coding:utf-8 *-*

from re import sub,findall,DOTALL
```

```
5  from random import shuffle
6
7  # zunächst einmal brauchen wir einen Text, um damit zu arbeiten. Wir nehmen
       ein
8  # Interview aus dem Spiegel, welches einfach aus dem Internet kopiert wurde.
9  # Der String wird als Unicode deklariert, weil es ansonsten Schwierigkeiten
       mit
```

### 6.2 Der Text

```
43 ------
44 # wir schauen uns die ersten zwanzig Zeichen des Textes an
45 print "[i] Anfang des Textes: '"+text[0:20]+"...'"
```

### 6.3 Fragen und Antworten extrahieren

```
46 ------
47 # wir definieren einen regulären Ausdruck, der alle Fragen findet. Dazu
48 # verwenden wir die Backreferenz mit Hilfe der einfachen Klammern. Jede
       Frage
49 # beginnt mit der Zeichenkette "Frage: ". Der Rest (bis zum Newline-Zeichen)
50 # ist der Inhalt der Frage. Also lässt sich der Ausdruck, der alle Fragen
51 # findet, wie folgt definieren:
52 question = 'Frage: (.*)'
53
54 # wir testen das Ganze mit findall()
55 fragen = findall(question,text)
56
57 # wir schauen nach, ob der Ausdruck gewirkt hat:
58 print "[i] Ausgabe der Fragen: "
59 for frage in fragen: print '- ' + frage
60
61 # das können wir im Prinzip noch schöner ausgeben, mit gleichzeitiger
62 # Nummerierung der Fragen:
63 print "[i] Nummerierte Ausgabe der Fragen: "
64 for i,frage in enumerate(fragen): print "["+str(i+1)+"] "+frage
65
66 # nun suchen wir die Antworten. Das Prinzip ist dasselbe wie im vorherigen
67 # Fall:
68 answer = 'Schimpf: (.*)'
69 print "[i] Ausgabe der Antworten:"
70 antworten = findall(answer,text)
71 for i,antwort in enumerate(antworten): print "["+str(i+1)+"] "+antwort
```

### 6.4 Wortstatistik

```
72 ——————
73 # Wir interessieren uns nun mehr für einige Eigenschaften des Textes.
74 # Beispielsweise interessiert uns, welche Wörter wie oft in einem Text
75 # vorkommen. Ein Wort ist ein beliebiges Zeichen außer den Zahlen, dem Komma
      ,
76 # dem Semikolon, dem Fragezeichen, dem Punkt, der Klammern und einigen
      weiteren
77 # Sonderzeichen (inklusive dem Leerzeichen).
78 word = u'[^ \(\)_\. ,;\?\n\d—:!"]+'
79 woerter = findall(word,text)
80 for wort in woerter: print '- '+wort
81
82 # Nun wollen wir aber wissen, wie oft die Wörter in dem Text vorkommen.
      Hierzu
83 # bietet es sich an, alle Großbuchstaben zunächst zu Kleinbuchstaben zu
      machen,
84 # da man damit besser arbeiten kann und Satzanfänge nicht ins Gesicht fallen
85 # werden (das Ganze muss natürlich auch mit dem Text gemacht werden). Als
86 # weiteren Schritt brauchen wir die Wörter nicht zu wiederholen, wenn sie
87 # bereits einmal gezählt wurden. Wir müssen also alle in der Liste mehrmals
88 # auftauchenden Wörter entfernen:
89 unique_woerter = []
90 for wort in woerter:
91     if wort.lower() not in unique_woerter:
92         unique_woerter.append(wort.lower())
93 # Wir sortieren die Liste besser noch mal alphabetisch:
94 unique_woerter = sorted(unique_woerter)
95 # jetzt kann es losgehen
96 for i,wort in enumerate(unique_woerter):
97     anzahl = text.lower().count(wort)
98     print "["+str(i+1)+"] "+wort+": "+str(anzahl)
```

### 6.5 Verarmtes Deutsch

```
99 ——————
100 # Zum Abschluss erlauben wir uns noch einen kleinen Scherz und machen den
      Text
101 # zu einem Beispiel für den Sprachverfall, indem wir alle Vokale in ein "a"
102 # verwandeln und alle Mehrfachkonsonanzen abbauen. Dafür brauchen wir zwei
103 # reguläre Ausdrücke und die Sub-Funktion.
104 # der folgende Ausdruck matcht alle Vokale für das Deutsche:
105 vokalabbau = u'[aeiouäüöAEIOUÄÜÖy]+'
106 # Dieser Ausdruck matcht alle Konsonanten:
107 konsonanten = u'[^aeiouäüÖÄÜÖAEIOUy \(\)_\. ,;\?\n\d—:!"]'
108 # Wir kombinieren ihn mit Backreferenzen (der erste Konsonant), um die
109 # Ersetzung zu gewährleisten.
```

```
konsonantenabbau = u'('+konsonanten + ')'+konsonanten+'+'
newtext = sub(vokalabbau,'a',text)
newtext = sub(konsonantenabbau,r'\1',newtext)
print "[i] Primitives Zukunftsdeutsch: "+newtext[0:100]
```

### 6.6 Verdrehtes Deutsch

```
————————
# Nun testen wir noch in einem letzten Versuch, ob man Texte, deren Wörter
# durcheinandergewürfelt wurden, wirklich lesen kann! Wir brauchen dafür die
# list-Funktion, und die shuffle-Funktion aus dem random-Modul.
# zunächst einmal suchen wir noch mal alle Wörter im Text und speichern
   diese
# in einer Liste mit unique Wörtern ab.
unique_words = []
for wort in woerter:
    if wort not in unique_words:
        unique_words.append(wort)

# jetzt definieren wir eine kurze Funktion, die einen string
# durcheinanderwürfelt:
def shuffleString(string):
    liste = list(string)
    shuffle(liste)
    new_string = ''.join(liste)
    return new_string

# jetzt kreieren wir eine Liste aus Tuplen, deren erstes Element jeweils die
# source, und deren zweites Element das target ist
st_list = []
for word in unique_words:
    st_list.append((word,shuffleString(word)))

# jetzt wenden wir mit Hilfe von sub() diese Funktion iterativ auf den text
   an
new_text = text
for source,target in st_list:
    new_text = sub(source,target,new_text)

print "[i] Ein durcheinandergewürfelter Text:",new_text[0:100]
print "[i] Das Original:                     ",text[0:100]
```

# Dateien (1)

## 1 Dateien im Allgemeinen

### 1.1 Das obligatorische Wikizitat

Eine Datei (engl. file) ist ein strukturierter Bestand inhaltlich zusammengehöriger Daten, die auf einem beliebigen Datenträger oder Speichermedium abgelegt bzw. gespeichert werden kann. Diese Daten existieren über die Laufzeit eines Programms hinaus und werden als nicht flüchtig oder persistent bezeichnet. Das Wort Datei ist ein Kofferwort aus Daten und Kartei, geschaffen durch das Deutsche Institut für Normung (DIN).

In der elektronischen Datenverarbeitung ist der Inhalt jeder Datei zunächst eine eindimensionale Aneinanderreihung von Bits, die normalerweise in Byte-Blöcken zusammengefasst interpretiert werden. Erst der Anwender einer Datei bzw. ein Anwendungsprogramm oder das Betriebssystem selbst interpretieren diese Bit- oder Bytefolge beispielsweise als einen Text, ein ausführbares Programm, ein Bild oder eine Tonaufzeichnung.

### 1.2 Was ist ein Kofferwort?

Ein Kofferwort oder Portmanteau ist ein Kunstwort, das aus mindestens zwei Wortsegmenten besteht, die zu einem inhaltlich neuen Begriff verschmolzen sind. Der Neologismus ist im Deutschen erstmals 1983 in Hadumod Bußmanns Lexikon der Sprachwissenschaft anzutreffen. Weitere Synonyme sind: Blend, Wortkreuzung und mot-valise, das man im Französischen seit 1970 findet. Der Vorgang, durch den Kofferwörter entstehen, wird in der Wortbildungsmorphologie als Amalgamierung oder Kontamination bezeichnet. (aus: Wikipedia)

### 1.3 Dateitypen

- Ausführbare Dateien:
  - Programme in Maschinensprache
  - Programme in Skriptsprachen
- Nichtausführbare Dateien:
  - Textdateien
  - Audiodateien
  - Bilddateien
  - Datenbankdateien
  - Binärdateien
- Verzeichnisse

### 1.4 Aufgabe

Gib für jeden der Dateitypen ein Beispiel. Woran kann man die unterschiedlichen Dateitypen normalerweise erkennen?

# 2 Textdateien in Python

## 2.1 Der Datentyp file

### Allgemeines

- Das Arbeiten mit einfachen Textdateien wird in Python mit Hilfe des Datentyps **file** geregelt.
- Ein **file** wird deklariert, indem der Pfad und der Name zu der Datei, die geöffnet oder erstellt werden soll, mit Hilfe der Funktion **open** angegeben wird.
- Als zweiter Parameter wird mit Hilfe eines Strings angegeben, ob der File kreiiert werden soll (**'w'**), ob er schon existiert und nur gelesen werden soll (**'r'**), oder ob an diesen File Daten angehängt werden sollen (**'a'**).
- Der Befehl **open('datei.txt','w')** öffnet eine (zuvor nicht vorhandene) Datei und ermöglicht es, sie zu beschreiben.
- Der Befehl **open('datei.txt','r')** öffnet eine vorhandene Datei und ermöglicht es, ihren Inhalt einzulesen. Wenn die Datei nicht existiert, löst dies einen **IOError** aus.
- Files werden immer da angelegt, wo man sich gerade im Computer befindet, es sei denn, es wird ein spezieller Pfad definiert. Am einfachsten ist es, Programme zum Öffnen und Bearbeiten von Files immer im gleichen Ordner anzulegen, wo sich auch die enstprechenden Files befinden, weil dann die einfache Angabe des Filenamens genügt, um ihn zu bearbeiten.
- Der Befehl **file.close()** schließt den File, was, sofern er verändert wurde, dazu führt, dass die Veränderungen endgültig gespeichert werden.

### Beispiele

```
1  >>> infile = open('test.txt','r')
2  >>> data = infile.read()
3  >>> print data
4  Dies ist ein simpler Text in einem Textfile, der zu Probezwecken erstellt wurde.
5
6  >>> data
7  'Dies ist ein simpler Text in einem Textfile, der zu Probezwecken erstellt wurde.\n
       '
8  >>> type(data)
9  <type 'str'>
10 >>> type(infile)
11 <type 'file'>
```

### Aufgabe

Ein Spaßvogel sich in den Computer Deiner WG-Mitbewohnerin, welche sich an Schauspielschulen bewerben möchte, gehäckt und den Text für Ihre neue Rolle dergestalt geändert, dass dieser nun in komplett verkehrter Reihenfolge dasteht. Da sie morgen eine Prüfung an der Ernst-Busch-Schule in Berlin hat, für die sie den Text unbedingt noch einmal durcharbeiten muss, ist sie völlig verzweifelt und wendet sich an Dich Pythonexperten um Hilfe. Wie kannst Du ihr den Text (<reverse.txt>) so schnell wie möglich in die ursprüngliche Form zurückversetzen?

## 2.2 Der Befehl glob

**Allgemeines**

- Wie bereits erwähnt, bietet es sich an, Files immer im selben Ordner, in dem auch das Skript liegt, zu bearbeiten.
- Möchte man sich jedoch einen Überblick über Unterordner verschaffen, so ist es hilfreich, von Python aus auf diese zuzugreifen.
- Hierzu bietet sich der Befehl **glob** aus dem Modul **glob** an, welcher im Unix-Wildcard-Stil nach Dateien in Ordnern sucht, die ein bestimmtes Muster aufweisen.
- Der Wildcard-Stil basiert grundlegend darauf, dass das Sternzeichen '*' anstelle beliebiger Substrings treten kann.
- Der Befehlt **glob('*.txt')** liefert beispielsweise eine Liste aller Dateien in dem aktuellen ordner, die die Endung '.txt' aufweisen.
- Der Vorteil des glob-Befehls ist, dass eine einheitliche Pfadbenennung, die unabhängig vom jeweiligen Betriebssystem ist, verwendet werden kann. Bei dieser Benennung werden Ordner durch einen Slash '/' gekennzeichnet (vgl. das Beispiel).

**Beispiele**

```
>>> from glob import glob
>>> glob('folder/*.txt')
['folder/machen_wir.txt', 'folder/heute.txt', 'folder/quatsch.txt']
>>> sorted([i[7:] for i in glob('folder/*.txt')])
['heute.txt', 'machen_wir.txt', 'quatsch.txt']
>>> sorted([i[7:].replace('.txt','') for i in glob('folder/*.txt')])
['heute', 'machen_wir', 'quatsch']
>>> quarks = sorted([i[7:].replace('.txt','') for i in glob('folder/*.txt')])
>>> for quark in quark: print quark
heute
machen_wir
quatsch

>>> files = glob('folder/*.txt')
>>> for file in sorted(files): print file,'\t',open(file,'r').read().strip()
folder/heute.txt        Nummer 1
folder/machen_wir.txt   Nummer 2
folder/quatsch.txt      Nummer 3
```

**Aufgabe**

Dein gemeiner WG-Mitbewohner hat Dir einen Streich gespielt und Deine Lieblingsmusik umbenannt und in einem Ordner voller leerer MP3-Dateien versteckt (<mp3>). Du bist zu faul, jede Datei auf ihre Byte-Zahl zu überprüfen, geschweige denn, zu versuchen, sie in deinem Mediaplayer abzuspielen. Stattdessen öffnest Du schnell die Pythonkonsole, und schreibst ein paar kurze Zeilen in die IDLE. Welche?

## 2.3 Arbeiten mit Textdateien: Einlesen von Dateien

### Allgemeines

- Die einfachste Art, mit Textdateien, ist die Daten einzulesen.
- Während wir bisher Daten, wie kleine Textausschnitte oder Ähnliches, immer in das Skript selbst geschrieben haben, ermöglicht der Datentyp **file** es uns, die Daten in gesonderter Form abzulegen, zu erstellen, und dann mit Hilfe von Skripten oder von der Konsole aus zu bearbeiten.
- Beim Einlesen ist der einfachste Befehl der Befehl **file.read()**, welcher die Datei komplett einliest. Er wurde in den obigen Beispiele bereits verwendet.
- Noch praktischer ist jedoch, dass Python auch das Iterieren über files gestattet. So kann man mit dem Befehl **for line in file: print line** bspw. jede Zeile einer Datei einzeln ausgeben.
- Beim Iterieren von files wird jede Zeile einzeln eingelesen und ausgegeben. Sie kann dann einfach in einer Liste mit Hilfe des Befehls **liste.append(line)** gespeichert werden.
- Man sollte das zeilenweise Einlesen von Dateien immer mit dem Kommando **line.strip()** kombinieren, da dieser jeglichen unnötigen Whitespace löscht. Da Dateien in Python zunächst immer als **string** repräsentiert werden, ist es von Vorteil, wenn bspw. keine unnötigen Newlinezeichen, oder Leerzeichen am Rand einer Datei nach dem Einlesen auftauchen.

### Beispiel

```
>>> infile = open('einkaufsliste.txt','r')
>>> tobuy = []
>>> for line in infile:
...     tobuy.append(line.strip())
...
>>> for line in tobuy: print line
Anzahl  Produkt
6       Bier
1       Butter
2       gebratener Hase
1       Salat ohne Ehec
4       Billigfertigpizza
1       Chips
3       Joghurt

>>> for i in range(len(tobuy)):
...     tobuy[i] = tobuy[i].split('\t')
...
>>> tobuy
[['Anzahl', 'Produkt'],
 ['6', 'Bier'],
 ['1', 'Butter'],
 ['2', 'gebratener Hase'],
 ['1', 'Salat ohne Ehec'],
 ['4', 'Billigfertigpizza'],
 ['1', 'Chips'],
 ['3', 'Joghurt']]
>>> for i,line in enumerate(tobuy):
...     tobuy[i][0] = int(tobuy[i][0])
...
>>> anzahl = [i[0] for i in tobuy[1:]]
>>> anzahl
```

```
33 [6, 1, 2, 1, 4, 1, 3]
34 >>> sum(anzahl)
35 18
36 >>> stuff = [i[1] for i in tobuy[1:]]
37 >>> print ','.join(stuff)
38 Bier,Butter,gebratener Hase,Salat ohne Ehec,Billigfertigpizza,Chips,Joghurt
39 >>> def wieviel(item):
40 ...     return anzahl[stuff.index(item)]
41 ...
42 >>> wieviel('Bier')
43 6
44 >>> wieviel('Salat ohne Ehec')
45 1
```

**Aufgabe**

Das Skript <simplePlot.py> stellt eine einfache, ASCII-basierte Methode dar, Temperaturdaten in Kurvenform zu plotten. Als Input benötigt man, nachdem die Funktion geladen wurde, ein Dictionary, welches als Key jeweils die zwölf Monatsnamen enthält, und die jeweilige Durchschnittstemperatur als Value, der ein Float sein sollte. Gegeben ist die Text-Datei <temperatures.txt>, in der monatliche Durchschnittstemperaturen in Deutschland für die Jahre 1985 – 1994 aufgelistet sind. Lies die Datei in die Pythonkonsole, oder ein Pythonskript ein und versuche, für das Jahr 1990 ein Dictionary zu erstellen, das sich als Eingabedatentyp für die **temperature_plot**-Funktion in <simplePlot.py> eignet, und teste damit die Funktion. Berechne gleichzeitig die Durchschnittstemperatur für das Jahr.

## 3 Binärdateien in Python: Das Modul pickle

# Dateien (1)

## 1 Dateien im Allgemeinen

### 1.1 Das obligatorische Wikizitat

Eine Datei (engl. file) ist ein strukturierter Bestand inhaltlich zusammengehöriger Daten, die auf einem beliebigen Datenträger oder Speichermedium abgelegt bzw. gespeichert werden kann. Diese Daten existieren über die Laufzeit eines Programms hinaus und werden als nicht flüchtig oder persistent bezeichnet. Das Wort Datei ist ein Kofferwort aus Daten und Kartei, geschaffen durch das Deutsche Institut für Normung (DIN).

In der elektronischen Datenverarbeitung ist der Inhalt jeder Datei zunächst eine eindimensionale Aneinanderreihung von Bits, die normalerweise in Byte-Blöcken zusammengefasst interpretiert werden. Erst der Anwender einer Datei bzw. ein Anwendungsprogramm oder das Betriebssystem selbst interpretieren diese Bit- oder Bytefolge beispielsweise als einen Text, ein ausführbares Programm, ein Bild oder eine Tonaufzeichnung.

### 1.2 Was ist ein Kofferwort?

Ein Kofferwort oder Portmanteau ist ein Kunstwort, das aus mindestens zwei Wortsegmenten besteht, die zu einem inhaltlich neuen Begriff verschmolzen sind. Der Neologismus ist im Deutschen erstmals 1983 in Hadumod Bußmanns Lexikon der Sprachwissenschaft anzutreffen. Weitere Synonyme sind: Blend, Wortkreuzung und mot-valise, das man im Französischen seit 1970 findet. Der Vorgang, durch den Kofferwörter entstehen, wird in der Wortbildungsmorphologie als Amalgamierung oder Kontamination bezeichnet. (aus: Wikipedia)

### 1.3 Dateitypen

- Ausführbare Dateien:
  - Programme in Maschinensprache
  - Programme in Skriptsprachen
- Nichtausführbare Dateien:
  - Textdateien
  - Audiodateien
  - Bilddateien
  - Datenbankdateien
  - Binärdateien
- Verzeichnisse

### 1.4 Aufgabe

Gib für jeden der Dateitypen ein Beispiel. Woran kann man die unterschiedlichen Dateitypen normalerweise erkennen?

# 2 Textdateien in Python

## 2.1 Der Datentyp file

### Allgemeines

- Das Arbeiten mit einfachen Textdateien wird in Python mit Hilfe des Datentyps **file** geregelt.
- Ein **file** wird deklariert, indem der Pfad und der Name zu der Datei, die geöffnet oder erstellt werden soll, mit Hilfe der Funktion **open** angegeben wird.
- Als zweiter Parameter wird mit Hilfe eines Strings angegeben, ob der File kreiiert werden soll (**'w'**), ob er schon existiert und nur gelesen werden soll (**'r'**), oder ob an diesen File Daten angehängt werden sollen (**'a'**).
- Der Befehl **open('datei.txt','w')** öffnet eine (zuvor nicht vorhandene) Datei und ermöglicht es, sie zu beschreiben.
- Der Befehl **open('datei.txt','r')** öffnet eine vorhandene Datei und ermöglicht es, ihren Inhalt einzulesen. Wenn die Datei nicht existiert, löst dies einen **IOError** aus.
- Files werden immer da angelegt, wo man sich gerade im Computer befindet, es sei denn, es wird ein spezieller Pfad definiert. Am einfachsten ist es, Programme zum Öffnen und Bearbeiten von Files immer im gleichen Ordner anzulegen, wo sich auch die enstprechenden Files befinden, weil dann die einfache Angabe des Filenamens genügt, um ihn zu bearbeiten.
- Der Befehl **file.close()** schließt den File, was, sofern er verändert wurde, dazu führt, dass die Veränderungen endgültig gespeichert werden.

### Beispiele

```
1  >>> infile = open('test.txt','r')
2  >>> data = infile.read()
3  >>> print data
4  Dies ist ein simpler Text in einem Textfile, der zu Probezwecken erstellt wurde.
5
6  >>> data
7  'Dies ist ein simpler Text in einem Textfile, der zu Probezwecken erstellt wurde.\n
       '
8  >>> type(data)
9  <type 'str'>
10 >>> type(infile)
11 <type 'file'>
```

### Aufgabe

Ein Spaßvogel sich in den Computer Deiner WG-Mitbewohnerin, welche sich an Schauspielschulen bewerben möchte, gehäckt und den Text für Ihre neue Rolle dergestalt geändert, dass dieser nun in komplett verkehrter Reihenfolge dasteht. Da sie morgen eine Prüfung an der Ernst-Busch-Schule in Berlin hat, für die sie den Text unbedingt noch einmal durcharbeiten muss, ist sie völlig verzweifelt und wendet sich an Dich Pythonexperten um Hilfe. Wie kannst Du ihr den Text (<reverse.txt>) so schnell wie möglich in die ursprüngliche Form zurückversetzen?

## 2.2 Der Befehl glob

**Allgemeines**

- Wie bereits erwähnt, bietet es sich an, Files immer im selben Ordner, in dem auch das Skript liegt, zu bearbeiten.
- Möchte man sich jedoch einen Überblick über Unterordner verschaffen, so ist es hilfreich, von Python aus auf diese zuzugreifen.
- Hierzu bietet sich der Befehl **glob** aus dem Modul **glob** an, welcher im Unix-Wildcard-Stil nach Dateien in Ordnern sucht, die ein bestimmtes Muster aufweisen.
- Der Wildcard-Stil basiert grundlegend darauf, dass das Sternzeichen '*' anstelle beliebiger Substrings treten kann.
- Der Befehlt **glob('*.txt')** liefert beispielsweise eine Liste aller Dateien in dem aktuellen ordner, die die Endung '.txt' aufweisen.
- Der Vorteil des glob-Befehls ist, dass eine einheitliche Pfadbenennung, die unabhängig vom jeweiligen Betriebssystem ist, verwendet werden kann. Bei dieser Benennung werden Ordner durch einen Slash '/' gekennzeichnet (vgl. das Beispiel).

**Beispiele**

```
>>> from glob import glob
>>> glob('folder/*.txt')
['folder/machen_wir.txt', 'folder/heute.txt', 'folder/quatsch.txt']
>>> sorted([i[7:] for i in glob('folder/*.txt')])
['heute.txt', 'machen_wir.txt', 'quatsch.txt']
>>> sorted([i[7:].replace('.txt','') for i in glob('folder/*.txt')])
['heute', 'machen_wir', 'quatsch']
>>> quarks = sorted([i[7:].replace('.txt','') for i in glob('folder/*.txt')])
>>> for quark in quark: print quark
heute
machen_wir
quatsch

>>> files = glob('folder/*.txt')
>>> for file in sorted(files): print file,'\t',open(file,'r').read().strip()
folder/heute.txt        Nummer 1
folder/machen_wir.txt   Nummer 2
folder/quatsch.txt      Nummer 3
```

**Aufgabe**

Dein gemeiner WG-Mitbewohner hat Dir einen Streich gespielt und Deine Lieblingsmusik umbenannt und in einem Ordner voller leerer MP3-Dateien versteckt (<mp3>). Du bist zu faul, jede Datei auf ihre Byte-Zahl zu überprüfen, geschweige denn, zu versuchen, sie in deinem Mediaplayer abzuspielen. Stattdessen öffnest Du schnell die Pythonkonsole, und schreibst ein paar kurze Zeilen in die IDLE. Welche?

## 2.3 Arbeiten mit Textdateien: Einlesen von Dateien

### Allgemeines

- Die einfachste Art, mit Textdateien, ist die Daten einzulesen.
- Während wir bisher Daten, wie kleine Textausschnitte oder Ähnliches, immer in das Skript selbst geschrieben haben, ermöglicht der Datentyp **file** es uns, die Daten in gesonderter Form abzulegen, zu erstellen, und dann mit Hilfe von Skripten oder von der Konsole aus zu bearbeiten.
- Beim Einlesen ist der einfachste Befehl der Befehl **file.read()**, welcher die Datei komplett einliest. Er wurde in den obigen Beispiele bereits verwendet.
- Noch praktischer ist jedoch, dass Python auch das Iterieren über files gestattet. So kann man mit dem Befehl **for line in file: print line** bspw. jede Zeile einer Datei einzeln ausgeben.
- Beim Iterieren von files wird jede Zeile einzeln eingelesen und ausgegeben. Sie kann dann einfach in einer Liste mit Hilfe des Befehls **liste.append(line)** gespeichert werden.
- Man sollte das zeilenweise Einlesen von Dateien immer mit dem Kommando **line.strip()** kombinieren, da dieser jeglichen unnötigen Whitespace löscht. Da Dateien in Python zunächst immer als **string** repräsentiert werden, ist es von Vorteil, wenn bspw. keine unnötigen Newlinezeichen, oder Leerzeichen am Rand einer Datei nach dem Einlesen auftauchen.

### Beispiel

```
>>> infile = open('einkaufsliste.txt','r')
>>> tobuy = []
>>> for line in infile:
...     tobuy.append(line.strip())
...
>>> for line in tobuy: print line
Anzahl  Produkt
6       Bier
1       Butter
2       gebratener Hase
1       Salat ohne Ehec
4       Billigfertigpizza
1       Chips
3       Joghurt

>>> for i in range(len(tobuy)):
...     tobuy[i] = tobuy[i].split('\t')
...
>>> tobuy
[['Anzahl', 'Produkt'],
 ['6', 'Bier'],
 ['1', 'Butter'],
 ['2', 'gebratener Hase'],
 ['1', 'Salat ohne Ehec'],
 ['4', 'Billigfertigpizza'],
 ['1', 'Chips'],
 ['3', 'Joghurt']]
>>> for i,line in enumerate(tobuy):
...     tobuy[i][0] = int(tobuy[i][0])
...
>>> anzahl = [i[0] for i in tobuy[1:]]
>>> anzahl
```

```
[6, 1, 2, 1, 4, 1, 3]
>>> sum(anzahl)
18
>>> stuff = [i[1] for i in tobuy[1:]]
>>> print ','.join(stuff)
Bier,Butter,gebratener Hase,Salat ohne Ehec,Billigfertigpizza,Chips,Joghurt
>>> def wieviel(item):
...     return anzahl[stuff.index(item)]
...
>>> wieviel('Bier')
6
>>> wieviel('Salat ohne Ehec')
1
```

**Aufgabe**

Das Skript <simplePlot.py> stellt eine einfache, ASCII-basierte Methode dar, Temperaturdaten in Kurvenform zu plotten. Als Input benötigt man, nachdem die Funktion geladen wurde, ein Dictionary, welches als Key jeweils die zwölf Monatsnamen enthält, und die jeweilige Durchschnittstemperatur als Value, der ein Float sein sollte. Gegeben ist die Text-Datei <temperatures.txt>, in der monatliche Durchschnittstemperaturen in Deutschland für die Jahre 1985 – 1994 aufgelistet sind. Lies die Datei in die Pythonkonsole, oder ein Pythonskript ein und versuche, für das Jahr 1990 ein Dictionary zu erstellen, das sich als Eingabedatentyp für die **temperature_plot**-Funktion in <simplePlot.py> eignet, und teste damit die Funktion. Berechne gleichzeitig die Durchschnittstemperatur für das Jahr.

# Dateien (2)

## 1 Erzeugen von Dateien

### 1.1 Allgemeines

- Genauso, wie man Informationen aus Dateien herauslesen kann, kann man sie auch in Dateien hineinschreiben.
- Dazu muss die zu beschreibende Datei mit dem speziellen Parameter 'w' (für "write") geöffnet werden.
- Die Inhalte können dann einfach als Strings in die Datei eingefügt werden.

### 1.2 Beispiel

```
>>> outfile = open('outfile.txt','w')
>>> outfile.write('lalala')
>>> outfile.close()
>>> meine_liste = [str(i) for i in range(10)]
>>> outfile = open('outfile.txt','w')
>>> for line in meine_liste:
...     outfile.write(line+'\n')
>>> outfile.close()
```

## 2 Aufpassen bei Unicode, Integern und anderen Datentypen

### 2.1 Allgemeines

- In einen File können NUR Strings geschrieben werden.
- Dies bedeutet, dass alle anderen Datentype, die man gerne in einen File hineinschreiben möchte, zuvor in Strings umgewandelt werden müssen.
- Dies funktioniert in den meisten Fällen ganz einfach mit dem Kommando str(), jedoch gibt es hier bestimmte Dinge, die man beachten muss.
- Der Datentyp Unicode wird mit Hilfe des Kommandos **unicode.encode('utf-8')** in einen String umgewandelt, der in einen File geschrieben werden kann.
- Möchte man reine, interne Python-Elemente in eine Datei schreiben, so bietet sich das Kommando **repr(x)** an, welches einen String des Datentypen erzeugt.

### 2.2 Beispiele

```
>>> outfile = open('outfile.txt','w')
>>> mychar = unicode(u'diesistunicode!')
>>> outfile.write(mychar.encode('utf-8'))
>>> outfile.close()
```

# 3 Übungen

## 3.1 Daten in Datei schreiben

Modifiziere das Programm aus der Hausaufgabe so, dass die in phonetische Umschrift kodierten Daten aus der Datei <in.txt> automatisch in die Datei <out.txt> geschrieben werden.

## 3.2 Spalten vertauschen

Gegeben ist die eine Einkaufsliste (<einkaufsliste.txt>). Erzeuge eine modifizierte Version dieses Files (<einkaufliste_mod.txt>), in der die erste und die zweite Spalte vertauscht sind.

## 3.3 Zeilen in Spalten verwandeln

Gegeben ist die ein wenig unübersichtliche Datei <transponiert.txt>. Übersichtlicher wäre diese wohl, wenn man Zeilen in Spalten umwandeln würde. Versuche dies zu bewerkstelligen und speichere die Ergebnisse in der Datei <grosstaedte.txt> ab.

## 3.4 Daten einlesen und neu sortiert ausgeben

Gegeben ist die Datei <grossstaedte.txt>. Erzeuge eine modifizierte Version dieser Datei (<grossstaedte_mod.txt>), in der die Städte alphabetisch sortiert sind. Füge als letzte Zeile die Durschschnittliche Einwohnerzahl aller Städte auf der Liste an.

## 3.5 Ein dämlicher Texteditor

Schreibe ein kleines simples Programm (<texteditor.py>), welches es dem Benutzer ermöglicht, so lange er will, Text in eine Datei zu schreiben. Hier bietet es sich an, die Datei im "append-Modus" zu öffnen, der durch ein 'a' im Anschluss an den Dateinamen eingeleitet wird.