

Martin Weißer

Python- Programmierung für Germanist:innen

Ein Lehr- und Arbeitsbuch

narr STUDIENBÜCHER

narr
ranck
e\atte
mpto



Dr. habil. Martin Weißer war bis Juni 2020 Professor für Fremdsprachenlinguistik an der Guangdong University of Foreign Studies und ist zurzeit Privatdozent an der Universität Bayreuth.

Martin Weißer

Python-Programmierung für Germanist:innen

Ein Lehr- und Arbeitsbuch

narr/f
ranck
e/atte
mpto

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.dnb.de> abrufbar.

DOI: <https://doi.org/10.24053/9783823394563>

© 2022 · Narr Francke Attempto Verlag GmbH + Co. KG

Dischingerweg 5 · D-72070 Tübingen

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Alle Informationen in diesem Buch wurden mit großer Sorgfalt erstellt. Fehler können dennoch nicht völlig ausgeschlossen werden. Weder Verlag noch Autor:innen oder Herausgeber:innen übernehmen deshalb eine Gewährleistung für die Korrektheit des Inhaltes und haften nicht für fehlerhafte Angaben und deren Folgen. Diese Publikation enthält gegebenenfalls Links zu externen Inhalten Dritter, auf die weder Verlag noch Autor:innen oder Herausgeber:innen Einfluss haben. Für die Inhalte der verlinkten Seiten sind stets die jeweiligen Anbieter oder Betreibenden der Seiten verantwortlich.

Internet: www.narr.de

eMail: info@narr.de

CPI books GmbH, Leck

ISSN 0941-8105

ISBN 978-3-8233-8456-4 (Print)

ISBN 978-3-8233-9456-3 (ePDF)

ISBN 978-3-8233-0315-2 (ePub)



Inhalt

1	Einleitung	9
1.1	Warum Python?	9
1.2	Kursüberblick und -ziele	10
1.3	Darstellungskonventionen	12
1.4	Python installieren	12
1.5	Einführung in Kommandozeile/Konsole/Terminal	14
1.6	Dateisysteme verstehen und navigieren	16
1.7	Editoren und IDEs	18
1.8	WingIDE Personal installieren und einrichten	19
1.9	„Sonderzeichen“ eingeben	19
1.10	Lösungen zu den Aufgaben	20
2	Grundlagen der Programmierung I – Anweisungen, Funktionen und einfache Datentypen	23
2.1	Anweisungen und Funktionen	23
2.2	Variablen	24
2.3	Datentypen – Übersicht	25
2.4	Zeichenketten und Zeichenkodierung	26
2.5	Zahlen	29
2.6	Operatoren	30
2.6.1	Mathematische Operatoren	30
2.6.2	Zeichenkettenoperatoren	31
2.6.3	Logische Operatoren	32
2.7	Skripte/Programme erstellen	33
2.8	Code Kommentieren	35
2.9	Lösungen zu den Aufgaben	36
3	Grundlagen der Programmierung II – Zusammengesetzte Datentypen, Interaktion und Kontrollfluss	43
3.1	Zusammengesetzte Datentypen	43
3.1.1	Listen	44
3.2	Einfache Interaktion mit Programmen und Benutzern	45
3.3	Lösungsstrategien und Schadensbegrenzung	46
3.4	Bedingungsabfragen	47

3.5	Schleifen	49
3.5.1	<code>while</code> -Schleifen	50
3.5.2	<code>for</code> -Schleifen	51
3.6	Lösungen zu den Aufgaben	51
4	Grundlagen der Zeichenkettenverarbeitung	59
4.1	Zeichenketten	59
4.2	Zeichenketten bereinigen	60
4.3	Mit Sequenzen arbeiten	62
4.3.1	Allgemeine Sequenzen und Listen	62
4.3.2	Tupel	64
4.4	Zeichenketten extrahieren	65
4.4.1	Zeichenketten effizient zusammenfügen	66
4.4.2	Groß- und Kleinschreibung handhaben	67
4.5	Zeichenketten formatieren	68
4.5.1	Verwendung des <code>%</code> -Operators	68
4.5.2	Die <code>format()</code> -Methode	68
4.5.3	Verwendung von f-strings	70
4.6	Lösungen zu den Aufgaben	70
5	Mit gespeicherten Daten arbeiten	75
5.1	Gespeicherte Daten	75
5.1.1	Dateien öffnen und schließen	75
5.1.2	Dateiinhalte lesen	77
5.1.3	Fehlerbehandlung	78
5.1.4	In Dateien schreiben	81
5.2	Mit Verzeichnissen und Pfaden arbeiten	82
5.2.1	Das <code>os</code> -Modul	82
5.2.2	Das <code>Path</code> -Objekt des <code>pathlib</code> -Moduls	84
5.3	Lösungen zu den Aufgaben	86
6	Sprachmuster erkennen und bearbeiten	93
6.1	Reguläre Ausdrücke	93
6.2	Allgemeine Suchsyntax	94
6.3	Mit dem <code>Match</code> -Objekt arbeiten	95
6.4	Zeichenklassen	96
6.5	Quantifizierung und Begrenzung	98
6.6	Maskieren und Verwendung von Sonderzeichen	99
6.7	Regex-Fehlerbehandlung	100
6.8	Verankerung, Gruppen und Alternation	101

6.9	Weitere Treffereingrenzungen	103
6.10	Kompilierungsflags	104
6.11	Lösungen zu den Aufgaben	106
7	Modularisierung und Objektorientierung	111
7.1	Dictionaries	111
7.2	Modularisierung	112
7.3	Benutzerdefinierte Funktionen	113
7.4	Module verstehen	113
7.5	Mit Modulen arbeiten	116
7.5.1	Module testen	116
7.5.2	Externe Module installieren	117
7.6	Klassen und Objekte	118
7.6.1	Methoden	119
7.6.2	Klassenschema	120
7.7	Lösungen zu den Aufgaben	121
8	Wortlisten, Frequenzen und Grundlagen der Sortierung	133
8.1	Wortlisten	133
8.1.1	Wortlisten generieren	134
8.1.2	Grundlagen der Sortierung	134
8.2	Einfache Wortfrequenzlisten generieren	136
8.3	Lambda-Funktionen	137
8.4	Relative Frequenzen	139
8.5	Lösungen zu den Aufgaben	140
9	Einfache grafische Benutzeroberflächen	149
9.1	Grafische Benutzeroberflächen	149
9.1.1	PyQt-Grundlagen	150
9.2	Allgemeiner Ansatz zur Entwicklung von GUI-Programmen	151
9.2.1	Nützliche PyQt-Steuerelemente	151
9.2.2	Ein minimales PyQt-Programm	153
9.2.3	Ableitung eines Hauptfensters	155
9.3	Mit Layouts arbeiten	157
9.4	Steuerelemente definieren und Layouts zuordnen	158
9.5	Eigenschaften, Methoden und Signale von Steuerelementen	159
9.6	Interaktive Funktionen hinzufügen	160
9.7	Aktionen	162
9.7.1	Menüs, Knopf- und Statuszeilen erstellen	162
9.8	Mit Dateien und Verzeichnissen in PyQt arbeiten	164
9.9	Lösungen zu den Aufgaben	166

10	Webdaten und Annotationen	171
10.1	Webdaten und Annotierungen	171
10.2	Auszeichnungssprachen	172
10.3	HTML Kurzeinführung	172
10.4	Webseiten herunterladen	174
10.5	List und Dictionary comprehension	176
10.6	Kurzeinführung in XML	178
10.7	Ersetzen mit Funktionen und globale Variablen	179
10.8	Text zu XML konvertieren	179
10.9	Lösungen zu den Aufgaben	181
11	Schlusswort	189
12	Appendix – Python-Programme	191
	Register	217
	Abbildungsverzeichnis	223
	Tabellenverzeichnis	224

1 Einleitung

Dieses Buch vermittelt Ihnen anhand von vielen praxisnahen Beispielen einen Überblick über die wichtigsten Konzepte in der Programmierung mit Python. Die Beispiele stammen hauptsächlich aus dem Bereich der Sprachwissenschaft, wobei viele der hier besprochenen Konzepte sich auch für die Analyse literarischer Werke oder im Rahmen der text-basierten Arbeit in den *Digital Humanities* eignen. Das Buch ist als Einführung konzipiert, so dass Sie keinerlei Grundkenntnisse in der Programmierung haben müssen. Alle zusätzlich benötigten Konzepte im Umgang mit Dateien etc. werden nach und nach eingeführt, ohne dabei ein tieferes Verständnis für Mathematik oder Informatik vorauszusetzen. Bevor ich Ihnen einen Überblick über die Struktur des Buches gebe, wollen wir jedoch ein paar kurze Überlegungen dazu anstellen, warum Sie überhaupt als Germanist:innen lernen sollten, in Python zu programmieren.

1.1 Warum Python?

Bei der Analyse von sprachlichen Phänomenen trifft man häufig auf Probleme, die sich nicht mittels existierender Programme lösen lassen, da diese schlicht und einfach nicht alle möglichen Optionen abdecken können. Um überhaupt fortgeschrittene Analysen durchführen zu können, ist man oft gezwungen, Daten in mehreren Schritten und mit mehreren Programmen relativ umständlich und zeitaufwendig so lange aufzubereiten, bis man diese Analysen letztendlich durchführen kann. Zum Glück kann man aber auch eine geeignete Programmiersprache lernen, um die Dinge selbst in die Hand zu nehmen und sich dann mehr auf die Lösung wichtiger Probleme konzentrieren zu können.

Python ist eine moderne Programmiersprache, deren Grundlagen relativ einfach zu erlernen sind, und die für alle gängigen Plattformen verfügbar ist. Letzteres ist ein sehr wichtiger Punkt, da man Programme ja nicht immer nur für eigene Zwecke erstellt, sondern oft auch mit Kommilitonen oder Kollegen teilen will, die vielleicht nicht dasselbe Betriebssystem verwenden, und es – je nach Betriebssystem – große Unterschiede gibt, welche Programme darauf laufen. Im Prinzip werden die meisten sprachlich interessierten Geisteswissenschaftler wahrscheinlich entweder Windows oder MacOS verwenden, da Linux eher von Computerlinguisten angewandt wird. Aber bei (gut geschriebenen) Python-Programmen spielt dies eigentlich keine Rolle, da sie gleichermaßen auf allen drei Plattformen laufen sollten.

Ein weiterer Vorteil von Python ist, dass darin geschriebene Programme ohne Kompilierung sofort ausführbar sind. Das heißt, man muss nicht erst einen (längeren)



Prozess durchlaufen, bei dem das Programm aus seinem Quellcode in ein komplettes lauffähiges Programm übersetzt wird wie z.B. bei C++ etc., sondern Python interpretiert einfach den Code, den Sie geschrieben haben, und startet Ihr Programm direkt. Dies erleichtert wiederum die Portabilität, vorausgesetzt, dass Python auch auf dem anderen Computer, wo das Programm laufen soll, installiert ist und alle eventuell benötigten Zusatzmodule vorhanden sind.

Als moderne Programmiersprache erlaubt uns Python, auf fortgeschrittenem Niveau objektorientiert zu arbeiten, wohingegen weniger versierte Anwender trotzdem noch rein prozedural arbeiten können, also nur die erforderlichen Programmschritte angeben müssen, ohne vorher Objekte definieren zu müssen. Die Objektorientierung bietet jedoch erfahreneren Programmierern große Vorteile bezüglich der Modularisierung von Programmen. Im Gegensatz zu anderen objektorientierten Sprachen, wie z.B. Java, ist dies ein erheblicher Vorteil, da man nicht von vornherein verstehen muss, wie Objekte aufgebaut sind und auch nicht für alle Zwecke gleich explizit ein Objekt anlegen, was wiederum den Arbeitsaufwand verringert und die Struktur von Programmen vereinfacht.

Außerdem gibt es für Python sehr viele Zusatzmodule, die die Arbeit mit speziellen Problemen erleichtern, so dass man nicht immer ‚das Rad neu erfinden‘ muss. Allerdings ist es dabei wichtig, zu verstehen, was diese Module bieten können und wo unter Umständen ihre Schwächen liegen, damit man bei ihrer Verwendung nicht unnötige Fehler in seinem eigenen Programm verursacht. Zu guter Letzt wird Python auch bei Sprachwissenschaftlerinnen und Computerlinguisten immer populärer, so dass damit nicht nur die Wahrscheinlichkeit steigt, dass man spezielle Module finden, sondern auch, dass man gut mit anderen bei der Erstellung neuer Module oder Programme zusammenarbeiten oder sich von erfahreneren Programmierern beraten lassen kann.

1.2 Kursüberblick und -ziele

Dieses Lehrbuch soll Ihnen die wichtigsten Grundbegriffe der Programmierung für Sprachanalysen vermitteln und Ihnen ermöglichen, Analysen durchzuführen, die nicht mithilfe existierender Programme durchgeführt werden können. Um dies zu erreichen, biete ich hier zunächst einen Überblick über einige wichtige Grundlagen der Arbeit mit dem Computer, da sich heutzutage allzu oft die Erfahrung vieler Benutzer mit Programmen auf die reine Verwendung vorinstallierter ‚Alltagsprogramme‘ beschränkt, und insbesondere ein Verständnis für den hierarchischen Aufbau von Speicherstrukturen und -orten auf dem Computer fehlt. Außerdem wird hier besprochen, was bei der Installation und Konfiguration der für den Kurs verwendeten Software zu beachten ist.

Sobald die Grundlagen im Umgang mit dem Computer erläutert sind, können wir in den Kapiteln 2 und 3 zu den elementaren Grundlagen der Programmierung (Anweisungen, Variablen, Kontrollstrukturen etc.) übergehen und langsam ein Verständnis dafür entwickeln, wie etwaige Lösungsstrategien für linguistische Fragen aussehen könnten.

Das Kapitel 4 vermittelt Ihnen dann die wichtigsten Grundlagen der Zeichenkettenverarbeitung. Da Sprachdaten ja im Prinzip aus Zeichenketten verschiedener Länge bestehen, von einzelnen Buchstaben, über Wörter, bis hin zu ganzen Texten, ist dieser Datentyp essenziell für die Analyse sprachliche Phänomene. Wir benötigen deshalb ein tiefgreifendes Verständnis dafür, wie man damit umgehen kann und in welcher Form solche Daten überhaupt auf dem Computer repräsentiert sind.

Weil das manuelle Eingeben von Daten in unserem Programmcode uns im Prinzip nur erlaubt, sehr einfache Programme zu schreiben, benötigen wir ebenfalls ein Wissen darüber, wie man auf die eigentlichen Sprachdaten zugreifen kann, die wir letztendlich verarbeiten wollen. Deshalb lernen wir in Kapitel 5, mit gespeicherten Daten zu arbeiten, um zunächst darauf lesend zuzugreifen, und später unsere Analyseergebnisse auch abspeichern zu können. In diesem Zusammenhang werden wir auch eruieren, wie man mit etwaigen Fehlern bei der Ein- und Ausgabe, wie z.B. der Angabe von falschen Dateinamen oder Speicherpfaden, umgehen kann, ohne dass unsere Programme gleich abstürzen, sowie Möglichkeiten kennenlernen, auf einfache Art und Weise mit Benutzern oder den Programmen zu interagieren.

Viele, wenn nicht sogar die meisten Sprachphänomene stellen Muster dar, die von einfach bis relativ komplex reichen. In Kapitel 6 erforschen wir, wie man mithilfe von sogenannten ‚regulären Ausdrücken‘ solche Sprachmuster effizient erfassen und bearbeiten kann.

Um Programme effizient aufbauen zu können, muss man in der Lage sein, sie in funktionale und wiederkehrende Untereinheiten zu unterteilen und diese ebenso effizient abzuspeichern, so dass sie später auch in andere Programme importiert werden können. In Kapitel 7 lernen wir deswegen, wie Modularisierung und Objektorientierung in Python funktionieren und unsere Arbeit erleichtern.

Sprachphänomene zu erkennen bringt uns zwar schon relativ weit in unseren Analysen, aber sie danach auch quantifizieren zu können, ist mindestens ebenso wichtig oder sogar fast noch wichtiger. Deshalb lernen wir in Kapitel 8 Optionen kennen, um Wort- und Frequenzlisten zu generieren und sinnvoll darzustellen. Um Letzteres zu erreichen, müssen wir uns auch mit den Grundlagen der Sortierung in Python befassen. Oft führt erst die Erstellung solcher Listen und Quantifizierung dazu, dass uns bewusst wird, welche Phänomene überhaupt interessant und es wert sind, weiter analysiert und beschrieben zu werden.

Grafische Benutzeroberflächenerleichtern (GUIs) ermöglichen oder erleichtern den effizienten Umgang mit Programmdaten in Bezug auf Darstellung, Generierung, Editierbarkeit etc. Deshalb erlernen wir in Kapitel 9 die wichtigsten Konzepte für die Erstellung solcher GUIs mithilfe der plattformübergreifend verwendbaren Bibliothek PyQt.

Die Arbeit mit Webdaten und Annotationen für Sprachanalysen gewinnt heutzutage immer mehr an Bedeutung. Aus diesem Grund gewinnen wir im Kapitel 10 einen Überblick über die Auszeichnungssprachen HTML und XML und lernen, wie man Daten mit Python aus dem Web herunterladen und verarbeiten sowie Texte von Rohtext zu XML wandeln kann. Alle Programme, die wir im Laufe der einzelnen Kapitel

entwickeln, können Sie unter <https://meta.narr.de/9783823384564/Zusatzmaterial.zip> herunterladen (vgl. Kasten auf p. 20).

1.3 Darstellungskonventionen

In diesem Buch verwende ich verschiedene Konventionen, um es Ihnen zum einen zu erleichtern, zwischen Phänomenen auf verschiedenen linguistischen Ebenen, aber auch zwischen beschreibendem Text und Instruktionen für Eingaben, insbesondere Programmcode, zu unterscheiden.

Beispielwörter oder Textpassagen im Text sind kursiv geschrieben. Um es Ihnen zu erleichtern, sich Fachtermini einzuprägen sind diese durch Fettdruck hervorgehoben, z.B. **Funktion**. Stellen solche Wörter Expansionen für Abkürzungen dar, so sind die Buchstaben, die Teile der Abkürzungen darstellen, zusätzlich kursiv gedruckt, wie z.B. in **eXtensible Markup Language**. Einfache Anführungszeichen markieren eine Abweichung vom normalen Sprachgebrauch, z.B., wenn sich etwas ‚kolloquialer‘ besser zum Ausdruck bringen lässt und nicht buchstäblich zu verstehen ist. Auf linguistischer Ebene werden geschweifte {...} oder spitze <...> Klammern dazu verwendet, um Morpheme und Grapheme von einfachen Buchstaben oder Wortteilen unterscheiden und somit ihre sprachlichen Funktionen besser zu verdeutlichen. Spitze Klammern sind allerdings auch ein Teil der Auszeichnungssprachen HTML und XML, wo sie natürlich eine andere Bedeutung haben, wie wir in Kapitel 10 sehen werden.

Damit Sie besser erkennen können, welche Instruktionen Sie selbst eingeben sollen, oder wenn wir Konstrukte in Python-Code besprechen, erscheinen diese in diesem Font. Variable Teile im Programmcode oder Syntaxbeschreibungen erscheinen hierbei kursiv. Zudem sind Syntaxzusammenfassungen durch Umrahmungen gekennzeichnet. Zum Teil sind andere Programmierbeispiele im Text auch freigestellt, d.h. sie erscheinen auf einer separaten Zeile, selbst wenn sie eigentlich in den Kontext eingebettet sind. In solchen Fällen kann es der Fall sein, dass orthografisch notwendige Interpunktionszeichen weggelassen werden, damit sie nicht als Teil des Programmcodes erscheinen und zu Verwirrung führen.

1.4 Python installieren

Die Installation von Python ist im Prinzip recht einfach, wenn man dabei ein paar wichtige Dinge beachtet. Für diesen Kurs verwenden wir Python 3, und die Installationspakete für verschiedene Plattformen sind direkt von <https://www.python.org/> herunterladbar. Allerdings ist unter Linux und MacOS Python meist schon vorinstalliert, normalerweise jedoch nur in Version 2, die auch vom Betriebssystem benötigt wird und deshalb nicht ersetzt werden darf! Deshalb muss man dort eine Parallelinstallation von Version 3 durchführen und seine Programme durch das Setzen der korrekten *Shebang-Zeile* (mehr

dazu später) ebenfalls als Python 3-Programme für das Betriebssystem bzw. den passenden Python-Interpreter kennzeichnen.

Als erste Übung führen wir jetzt die Installation von Python durch.

Übung 1 - Python installieren

Gehen Sie auf <https://www.python.org/>.

Finden Sie die aktuellste Python-3-Version, die für Ihr Betriebssystem geeignet ist. Unter 64-bit Windows kann entweder die 64-bit oder 32-bit Version installiert werden, wohingegen unter einer 32-bit Version von Windows nur ein 32-bit Python lauffähig ist.

Laden Sie die für Sie geeignete Version herunter und installieren Sie sie, wobei Sie auf jeden Fall die Option, ‚Python zum PATH hinzuzufügen‘ (Abbildung 1), aktivieren sollten, um die spätere Ausführbarkeit Ihrer Programme zu vereinfachen.

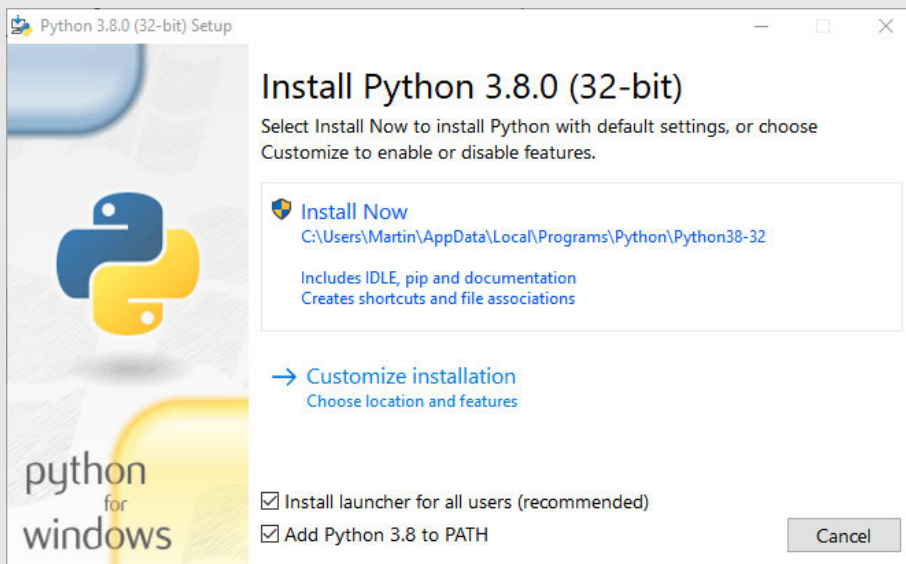


Abb. 1: Python Installation unter Windows

Für MacOS laden Sie ebenfalls ein Installationspaket herunter und installieren es per Doppelklick, parallel zu einer wahrscheinlich schon vorhandenen Version von Python 2. Allerdings müssen Sie zur Installation ein Administratorpasswort eingeben.



Unter Linux ist die Sache komplizierter: Je nach Distribution ist entweder schon eine Version 3 installiert (z.B. Ubuntu ab Version 17.10) oder Sie müssen eventuell parallel zu einer Version 2 eine Version 3 installieren. Das genaue Verfahren hängt von Ihrer Linux-Version ab, so dass es hier nicht einfach beschrieben werden kann.

Wie wir die Installation testen können, erfahren Sie in Kürze.



1.5 Einführung in Kommandozeile/Konsole/Terminal

Die meisten von Ihnen werden eher daran gewöhnt sein, bei Ihrer Arbeit auf dem Computer alles in Programmen auszuführen, die eine grafische Benutzeroberfläche haben und deshalb bequem zu bedienen sind. Die Programme, die wir jedoch entwickeln werden, bis wir es selber lernen, solche GUIs zu schreiben, laufen normalerweise nicht direkt über einen Doppelklick. Stattdessen werden sie aus einer Umgebung gestartet, die unter Windows als Eingabeaufforderung/Kommandozeile, unter Linux als Konsole/Terminal und auf dem Mac als Terminal bezeichnet wird, die wir ab sofort der Einfachheit halber immer als **Kommandozeile** bezeichnen werden. Diese Umgebung(en) ermöglichen die Eingabe von text-basierten Befehlen über eine sogenannte **Eingabeaufforderung** oder auch **Prompt**.

Unter Windows erfolgt der Aufruf der Eingabeaufforderung entweder durch Drücken von  + r, Eingabe des Befehls `cmd` und Drücken der Enter-Taste (↵), oder dadurch, dass Sie die -Taste drücken, dann den Buchstaben `c` tippen und dann die Option ‚Eingabeaufforderung‘ (oder ‚Command prompt‘ bei internationaler Einstellung) auswählen. Um die Kommandozeile als Administrator aufzurufen, z.B. um Python-Module für alle Benutzer zu installieren, können Sie bei der ersten Variante anstelle von Enter `Strg + Umschalt + Enter` drücken, bei der zweiten Variante einfach im Startmenü ‚Als Administrator ausführen‘ unter dem Eintrag für die Eingabeaufforderung auswählen.

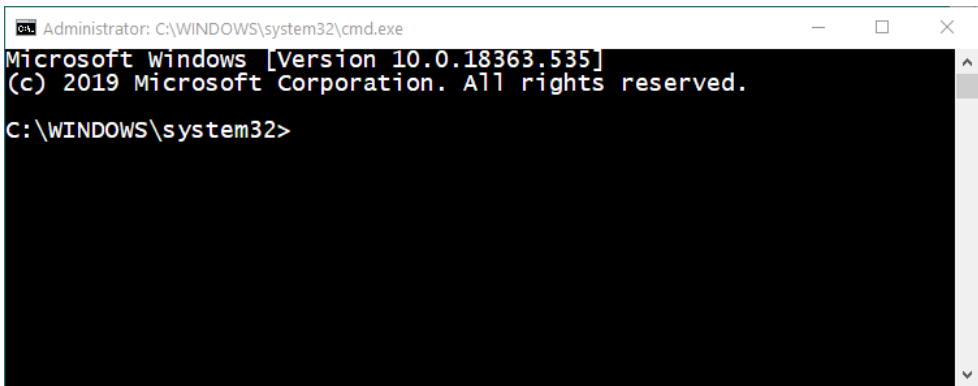


Abb. 2: Eingabeaufforderung als Administrator

Seit Kurzem gibt es unter Windows auch noch eine weitere Möglichkeit, die Kommandozeile aufzurufen, indem man in der Adresszeile des Windows Explorer einfach `cmd` eintippt. Dies öffnet die Kommandozeile dann direkt für das Verzeichnis, in dem man sich momentan befindet, so dass man nicht umständlich dorthin navigieren muss. Allerdings kann man auf diese Weise keine Kommandozeile als Administrator öffnen.

Auf dem Mac oder unter Linux muss man entweder das ‚Launchpad‘ starten (Mac) oder den Startknopf klicken (Linux) und dort nach ‚Terminal‘ (oder ähnlichem Namen) suchen und dies ausführen.



Abb. 3: Terminal unter MacOS

Befehle auf der Kommandozeile werden mit Befehlsname und möglichen Argumenten, jeweils durch Leerzeichen voneinander getrennt, eingegeben, und mit ↵ abgeschlossen und ausgelöst. Falls in den Argumenten Leerzeichen auftreten, was später bei unseren Programmen gut der Fall sein kann, dann müssen diese in paarweisen Anführungszeichen eingeschlossen werden. Dabei erwartet Windows doppelte Anführungszeichen und Mac OS oder Linux normalerweise einfache.

Übung 2 - Befehle auf der Kommandozeile eingeben

Öffnen Sie, wie oben besprochen, die Kommandozeile für Ihr Betriebssystem.

Geben Sie zunächst unter Windows den Befehl `python --version` oder unter Mac/Linux `python3 --version` ein und drücken Sie die Enter-Taste. Damit können Sie testen, ob Python richtig installiert ist.

Sollte keine Version mit Versionsnummer 3 ausgegeben werden, dann müssen Sie noch einmal versuchen, Python 3 zu installieren.

Lassen Sie die Kommandozeile geöffnet.



Wir werden im nächsten Abschnitt noch einige mehr solcher Befehle kennenlernen. Zum Schließen kann jeweils entweder der entsprechende Knopf am Fenster geklickt oder unter Windows der Befehl `exit` eingegeben werden. Auf dem Mac/unter Linux verwendet man `Strg + d`, wobei dann auf dem Mac auch noch das Fenster geschlossen werden muss. Alternativ kann man dort allerdings auch nur auf den roten Knopf zum Schließen klicken.

1.6 Dateisysteme verstehen und navigieren

Um mit Daten auf dem Computer umgehen zu können, muss man als Programmierer(in) etwas mehr Verständnis als normale Benutzer dafür haben, wie Daten auf dem Computer strukturiert abgelegt werden und ein Programm später auf sie zugreifen kann. Deshalb wollen wir dies jetzt hier kurz besprechen. **Dateisysteme** stellen die Verzeichnis- und Dateistrukturen auf dem Computer dar und haben eine hierarchische Anordnung als Baum mit Verzweigungen verschiedener Tiefe, dessen Wurzel normalerweise ein Laufwerk in Form einer Festplatte, CD, USB-Stick, oder Netzwerkressource ist. Allerdings haben Informatiker, ebenso wie Sprachwissenschaftler, im Gegensatz zu Normalsterblichen eher eine etwas verdrehte Weltansicht, weshalb dort die Bäume nach unten wachsen.

Windows verwendet Laufwerksbuchstaben (C, D, E, ...) zur Kennzeichnung von Laufwerken. Die **Wurzel**, unter der das Betriebssystem installiert ist, ist dort normalerweise die `C:\`-Platte oder `C:\`-Partition. Als Trennzeichen zwischen den Verzeichnisebenen dient hier ein `\`, wie z.B. in `C:\Temp`, wobei der `\` auch eine Abkürzung für die jeweilige Wurzel auf dem Laufwerk ist. Auf dem Mac/unter Linux gibt es keine Laufwerksbuchstaben, sondern nur **mount points**, das heißt Orte, wo Laufwerke in die Verzeichnishierarchie ‚eingehängt‘ werden. Die Wurzel ist hier `/`. Wichtig zu wissen ist auch, dass einfache Benutzerinnen – also nicht-Administratoren – meist nicht für alle Verzeichnisse Lese- oder Schreibberechtigungen haben, außer die Verzeichnisse gehören Ihnen oder wurden von Ihnen erstellt. Deshalb können auch die Programme, die wir selbst schreiben, nicht beliebig irgendwohin schreiben.

Um Verzeichnisse zu ermitteln oder deren Inhalt aufzulisten, bieten die verschiedenen Betriebssysteme ähnliche Methoden, aber teils andere Befehle, die wir hier kurz besprechen, aber auch bald üben. Unter Windows muss man `dir` (kurz für engl. *directory* = Verzeichnis) eintippen und `↵` drücken und erhält dann eine Liste der Dateien und Verzeichnisse innerhalb des aktuellen Verzeichnisses. Wird der Befehl ohne zusätzliche Argumente eingegeben, erhält man jedoch viele Zusatzinformationen, die man unter Umständen gar nicht benötigt. Um eine vereinfachte Darstellung mit weniger Informationen zu erhalten, kann man aber `dir /b` verwenden. Das `b` steht für *bare* und der Schrägstrich davor zeigt an, dass es sich um ein spezielles Argument handelt, welches den Befehl selbst modifiziert und nicht eine mögliche Form eines Dateinamensmusters. Letzteres kann ebenfalls als Argument mithilfe von sogenannten

Wildcards angegeben werden, wie z.B. in `dir /b *.txt`, womit nur Dateien, die die Endung `txt` haben, aufgelistet werden.

Unter Windows ist normalerweise das Startverzeichnis für die Kommandozeile für einfache Benutzer(innen) `C:\Users\Benutzername`, wobei `Benutzername` hier der Login-Name ist. Für Kommandozeilen, die als Administrator ausgeführt werden, ist es `C:\WINDOWS\System32`. Zum Wechseln zur Wurzel kann man das Kommando `cd \` eingeben.

Auf dem Mac oder unter Linux verwendet man `ls` (*listing* = Auflistung), um den Inhalt des aktuellen Verzeichnisses auszugeben. Im Gegensatz zu Windows liefert der Standardbefehl ohne Modifikation aber nur sehr wenige Zusatzinformationen. Falls man Zusatzinformationen, wie z.B. über Berechtigungen benötigt, kann man `ls -al` verwenden, wobei hier der Bindestrich den Beginn einer Modifikation anzeigt, `a` für ‚alle‘ (Dateien) und `l` für ‚lang‘, also ‚ausführlich‘, steht.

Unter MacOS oder Linux zeigt die Kommandozeile beim direkten Start normalerweise das Benutzerverzeichnis an (MacOS: `/Users/Benutzername`; Linux: `/home/Benutzername`). Die Kommandozeile kann normalerweise unter Linux auch direkt aus dem jeweiligen Dateimanager heraus über einen rechten Mausklick gestartet werden.

Um unter allen drei Betriebssystemen neue Verzeichnisse zu erstellen, gibt es sehr ähnliche Befehle. Der Befehl `mkdir Verzeichnisname` funktioniert unter Windows, Mac und Linux gleichermaßen, wobei unter Windows auch die Abkürzung `md Verzeichnisname` existiert. Auch können hier mithilfe von `m(k)d(ir) Verzeichnis\Unterverzeichnis` direkt Zwischenverzeichnisse automatisch erstellt werden, was bei den anderen Betriebssystemen nur mithilfe der Modifikation `-p` möglich ist.

Alle drei Betriebssysteme verwenden dasselbe Kommando, `cd Verzeichnispfad`, um in bestimmte Verzeichnisse zu wechseln, wobei `cd` für *change directory* steht. Unter Windows funktioniert dies aber nur auf der aktuellen Festplatte/Partition. Zum Wechsel in ein Verzeichnis auf einem anderen Laufwerk muss dieses zunächst durch Eingabe des Laufwerksbuchstabens, gefolgt von `:`, aktiviert werden. Da unter Mac/Linux alle Pfade ins Betriebssystem integriert sind, kann man jederzeit in alle Verzeichnisse wechseln.

Falls man nicht auf dem eigenen Computer arbeitet, wo man für die meisten Speicherorte Schreibrechte besitzt, bietet es sich an, seine Programme in einem Verzeichnis innerhalb des Benutzerzeichnisses abzulegen. Zum Wechsel ins Benutzerverzeichnis kann man unter Windows `cd %homepath%` verwenden und unter Mac/Linux entweder nur `cd` oder `cd ~`.

Um in das Wurzelverzeichnis zu wechseln, verwendet man unter Windows `cd \` und unter Mac/Linux: `cd /`. Komplexere Dateipfade werden aus Verzeichnis- und/oder Dateinamen zusammengesetzt, z.B.

`C:\temp\texte\text01.txt` (Windows) oder
`/temp/texte/text01.txt` (Mac/Linux),

um auf die Datei `text01.txt` im Unterverzeichnis `texte` im Verzeichnis `temp` zuzugreifen.



Übung 3 - Mit Verzeichnissen arbeiten

Kehren Sie zurück zur Kommandozeile.

Überprüfen Sie anhand des Pfades vor der Eingabeaufforderung, ob Sie sich in Ihrem Benutzerverzeichnis befinden.

Falls, nicht, wechseln Sie dort hin, um sicherzustellen, dass Sie Schreibrechte haben. Lassen Sie sich den Inhalt des Verzeichnisses jeweils in kurzer oder ausführlicher Form anzeigen und achten Sie dabei darauf, was genau Ihnen angezeigt wird. Verstehen Sie schon alles?

Legen Sie danach ein Verzeichnis `test` mit Unterverzeichnis `texte` an.

Lassen Sie sich den Inhalt des Verzeichnisses anzeigen.

Wechseln Sie dann in das `texte`-Unterverzeichnis.

Pfade können absolut oder relativ angegeben werden. Bei absoluten Pfaden muss immer eine komplette Pfadangabe erfolgen, z.B.

`C:\Users\Benutzername\test\texte` unter Windows oder
`/home/Benutzername/test/texte` auf dem Mac oder unter Linux.

Relative Pfade hingegen verwenden eine verkürzte Pfadangabe, die immer relativ zum aktuellen Verzeichnis ist. Dabei symbolisieren ein Punkt (.) das aktuelle Verzeichnis und zwei Punkte (..) ein jeweils übergeordnetes Verzeichnis.



Übung 4 - Mit relativen Pfaden arbeiten

Lassen Sie sich den Inhalt des nächsthöheren Verzeichnisses zunächst mit absolutem Pfad anzeigen und dann mit relativem Pfad.

Wechseln Sie dann mithilfe einer relativen Pfadangabe in das Verzeichnis zwei Ebenen über dem `texte`-Verzeichnis.

1.7 Editoren und IDEs

Wer noch nie programmiert hat, ist wahrscheinlich eher daran gewöhnt, Texte mit Textverarbeitungsprogrammen zu verarbeiten, die ihre Daten in proprietären, also speziell für die Anwendung optimierten, Formaten abspeichern, hat aber selten Grund dafür, einen reinen Editor zu verwenden. Editoren sind Programme, die zum Anzeigen und Bearbeiten von Rohtext-Dateien verwendet werden, in denen also Text ohne jegliche Formatierung, außer vielleicht Zeilenumbrüchen oder Einrückungen, auftritt. Die meisten Betriebssysteme bringen solche Editoren schon mit, weil sie sehr nützlich sind, um mit sehr einfachen Texten zu arbeiten. Zu ihnen zählen z.B. der Editor (engl. *Notepad*) unter Windows, TextEdit auf dem Mac, oder Gedit oder KWrite

unter Linux. Manche dieser Editoren bieten auch schon teilweise Unterstützung für Programmiersprachen, meist durch Syntaxhervorhebung, oder die Erstellung von Webseiten. Ein gutes Beispiel dafür unter Windows ist das kostenlose, aber sehr leistungsfähige, Notepad++.

Für ernsthafte Programmierprojekte sind solche Editoren allerdings meist nicht optimal, weshalb es besser ist, eine sogenannte Integrierte Entwicklungsumgebung (engl. **Integrated Development Environment**; **IDE**) zu verwenden, die dann weitere Unterstützung bei der Programmierung, in Form von Fehlersuche (engl. **debugging**), Syntaxhilfe und -vervollständigung, Setzen von Lesezeichen etc., bieten kann.

1.8 WingIDE Personal installieren und einrichten

Für diesen Kurs wollen wir die WingIDE Personal, die kostenlose Version einer professionellen IDE für Windows, Mac und Linux verwenden. Diese ist speziell für Python optimiert, unterstützt aber auch PyQt, HTML, XML etc. Heruntergeladen kann diese von <https://wingware.com/downloads/wing-personal> werden. Die genaue Installationsroutine ist abhängig vom Betriebssystem, aber gut beschrieben, so dass wir hier nicht näher darauf eingehen.

Das Programm ist ursprünglich auf Englisch, kann aber einfach für Deutsch lokalisiert werden, indem man über die Menüeinträge ‚Edit → Preferences → User Interface → Display Language‘ auswählt, und einen Neustart durchführt, woraufhin fast alle Einstellungen tatsächlich auch auf Deutsch vorliegen. Da die Standardkodierung für die erstellten Python-Dateien normalerweise die Einstellung des Betriebssystems übernimmt, empfiehlt es sich ebenfalls, diese dann über ‚Bearbeiten → Einstellungen → Standard-Kodierung‘ auf UTF-8 umzustellen, um die Ausführbarkeit auch auf Computern in anderen Ländern zu gewährleisten. Wir werden später noch mehr über die Bedeutung von UTF-8 für unsere Arbeit erfahren.

Übung 5 - WingIDE Personal Installieren

Laden Sie die Ihrem Betriebssystem entsprechende Version der WingIDE Personal herunter und installieren Sie sie.

Ändern Sie danach die Einstellungen für die Lokalisierung und Kodierung.


Machen Sie sich danach ein wenig mit dem Programm vertraut, indem Sie versuchen, die Menüeinträge und Komponenten der IDE zu verstehen.

1.9 ‚Sonderzeichen‘ eingeben

Wer immer nur in deutschsprachigen Ländern und mit einer deutschen Tastatur gearbeitet hat, musste sich wahrscheinlich nie Gedanken darüber machen, wie man denn



speziell deutsche Buchstaben wie die Umlaute eingeben kann. Wenn man allerdings Germanistik in einem anderen Land betreibt oder sich im Ausland einen Computer mit einer anderen Tastaturbelegung gekauft hat, dürfte einem die Eingabe solcher Zeichen durchaus als Problem bewusst geworden sein. Idealerweise sollte man in so einem Fall auch versuchen, eine deutsche Tastatur zu erwerben und verwenden. Falls dies nicht einfach möglich ist, kann man als Alternative eine Bildschirmtastatur einrichten.

Um dies zu tun, muss im Allgemeinen Deutsch als Sprachoption installiert sein, was unter Windows (10) meist relativ einfach zu erreichen ist, indem man die Einstellungen durch Drücken von  + i öffnet, unter ‚Uhrzeit & Sprache‘ ‚Sprache‘ auswählt, und dann Deutsch als neue Sprache unter den bevorzugten Sprachen hinzufügt. Danach kann man dann einfach auf der Anwendungsleiste per rechtem Mausklick ein Kontextmenü öffnen, dort die Option ‚Bildschirmtastatur anzeigen (Schaltfläche)‘ auswählen, und dadurch das Symbol für die Bildschirmtastatur zur Anwendungsleiste hinzufügen. Die Bildschirmtastatur kann dann jederzeit bei Bedarf durch Klicken auf dieses Symbol aktiviert werden. Auf dem Mac muss man nach Hinzufügen der Sprachoption in den Systemeinstellungen unter dem Tastatur-Eintrag ‚Tastatur- und Zeichenübersichten in der Menüleiste anzeigen‘ auswählen, woraufhin dann die Zeichenübersicht über das Tastatur-Icon in der Menüleiste ausgewählt werden kann. Unter Linux ist die genaue Einrichtung leider wieder abhängig von der Betriebssystemversion, so dass ich hier keine konkreten Angaben machen kann.

Nachdem Sie jetzt wissen, welche Vorteile Python Ihnen für Ihre Forschung bietet, wie Sie mit der Kommandozeile umgehen, und hoffentlich Python und die WingIDE erfolgreich installiert haben, können wir uns mit den Grundlagen der Python-Programmierung selbst vertraut machen.

Die Programme, die Sie in den Übungen erstellen werden, finden Sie im Appendix vollständig in einem für den Druck angepassten Format, teils mit zusätzlichen Annotierungen. Diese können auch unter <https://meta.narr.de/9783823384564/Zusatzmaterial.zip> zum Austesten und ‚Herumspielen‘ herunterladen.


Zudem finden Sie auf dieser Webseite Instruktionen, wie Sie die für einige Übungen verwendeten Textdateien mit literarischen Texten aus dem Internet herunterladen können und in eine geeignete Form bringen.



1.10 Lösungen zu den Aufgaben

Lösung 1 - Python installieren

Bei dieser Übung kann eigentlich nicht viel schiefgehen, es sei denn, Sie vergessen bei der Installation unter Windows auch gleich Python zum Pfad (PATH) hinzufügen zu lassen. Im Normalfall erfolgt die Installation ab Python 3.8 auch ins Benutzerverzeichnis, so dass Sie später eigentlich immer Berechtigungen zum Installieren

zusätzlicher Module haben sollten. Falls Sie den Installationsort manuell verändern, sollten Sie auf jeden Fall sicherstellen, dass Sie später auch für das entsprechende Verzeichnis Schreibberechtigung haben. Sollten Sie vergessen haben, Python zum PATH hinzuzufügen, drücken Sie  + i, um die Systemsteuerung zu öffnen, tippen Sie in der Suche *Umgebungsvariablen* ein und wählen Sie *Umgebungsvariablen* für dieses Konto bearbeiten. Doppelklicken Sie dann unter ‚Benutzervariablen für *Benutzername*‘ auf den ‚Path‘-Eintrag und auf ‚Bearbeiten‘ und tragen Sie `C:\Users\Benutzername\AppData\Local\Programs\Python\` ein, gefolgt von dem Verzeichnis, in dem die tatsächliche Python-Version liegt. Hierbei müssen Sie natürlich *Benutzername* durch Ihren eigenen ersetzen. Sobald Sie alle Dialoge bestätigt und geschlossen haben, müssten sich Ihre Python-Programme direkt starten lassen.

Unter Linux könnte es sein, dass Sie auf einem von mehreren Benutzern verwendeten Computer keine Administratorrechte haben, um Python 3 zu installieren. In diesem Fall sollten Sie idealerweise Ihren Administrator bitten, Python 3 für Sie zu installieren oder sich im Internet darüber informieren, wie Sie gegebenenfalls eine Version 3 nur für sich selbst einrichten können.

Lösung 2 - Befehle auf der Kommandozeile eingeben

Insofern Sie es geschafft haben, die Kommandozeile zu öffnen, kann eigentlich nicht mehr viel schiefgehen, es sei denn, Python ist nicht richtig installiert oder im Pfad eingetragen, oder Sie haben vergessen zwischen dem Aufruf des Python-Interpreters mittels `python` und dem Argument `--version` ein Leerzeichen einzugeben oder einen der zwei Bindestriche einzutippen. Im ersteren Fall müssten Sie eine Fehlermeldung erhalten, dass das Programm nicht gefunden wurde und im letzteren, dass Sie eine falsche Option angegeben haben.

Lösung 3 - Mit Verzeichnissen arbeiten

Falls an der Eingabeaufforderung nicht `C:\Users\Benutzername` (Windows), `/Users/Benutzername` (MacOS) oder `/home/Benutzername` (Linux) erscheint, müssen Sie jetzt entweder `cd` (Mac/Linux) oder `cd %homepath%` eingeben und die Eingabetaste drücken.

Lösung 4 - Mit relativen Pfaden arbeiten

Da wir zuletzt im `texte`-Unterverzeichnis des `test`-Verzeichnisses innerhalb des Benutzerverzeichnisses waren, hätten Sie zum Anzeigen des Verzeichnisses unter Windows `dir C:\Users\Benutzername\test`, oder auf dem Mac oder unter Linux `ls /home/Benutzername/test` eingeben müssen. Dabei hätten Sie umständlicherweise ziemlich viel schreiben müssen, während Sie viel einfacher mit den relativen Angaben `dir ..` oder `ls ..` zum Ziel kämen.

Um sich das Verzeichnis zwei Ebenen über dem `texte`-Verzeichnis anzeigen zu lassen, müssen Sie einfach `dir ../..` oder `ls ../../` verwenden. Zu beachten ist hierbei eigentlich nur, dass Sie den richtigen Verzeichnistrenner für Ihr Betriebssystem

verwenden und dass tatsächlich auch ein solches Verzeichnis existiert, was aber der Fall sein müsste, falls Sie die Verzeichnisse vorher richtig erstellt hatten und sich auch im richtigen Verzeichnis befinden.

Lösung 5 - WingIDE Personal Installieren

Insofern Sie Administratorrechte auf dem von Ihnen verwendeten Computer haben, kann bei dieser Übung eigentlich nichts schiefgehen. Falls Sie keine Administratorrechte besitzen, dann müssten Sie Ihren Administrator bitten, die IDE für Sie einzurichten.

2 Grundlagen der Programmierung I - Anweisungen, Funktionen und einfache Datentypen



In diesem Kapitel stelle ich Ihnen erste grundlegende Konzepte der Programmierung vor. Sie entwickeln ein Verständnis dafür, wie Python instruiert wird, einfache Befehle in Form von Anweisungen oder Funktionen auszuführen, was es mit Variablen auf sich hat, wie diese erzeugt und mit Daten ‚befüllt‘ werden sowie welche einfachen Datentypen überhaupt zur Verfügung stehen. Weiterhin lernen Sie, wie Zeichen auf dem Computer kodiert werden, und mithilfe von Operatoren verschiedene Arten von Aktionen mit Daten ausgeführt werden können. Zudem besprechen wir, wie Sie eigene Programme erstellen und den Code darin sinnvoll kommentieren können.

2.1 Anweisungen und Funktionen

Programminstruktionen in Python werden in Form von **Anweisungen** gegeben, wobei normalerweise eine Anweisung auf einer Zeile steht. So z.B. gibt

```
print('Guten Morgen!')
```

diese Begrüßung auf der Kommandozeile aus. Seltener schreibt man auch mehrere, durch Strichpunkte voneinander getrennte, Anweisungen in dieselbe Zeile, z.B.:

```
print('Guten Morgen!'); print("Oder ist es schon Abend?")
```

Funktionen (mehr dazu später) sind besondere Formen von Anweisungen, die meist ein oder mehrere **Argumente** in runden Klammern übergeben bekommen. Argumente sind dabei Daten, mit denen die Funktion etwas tun soll, wie z.B. diese zu verändern oder, wie im Falle der oben gezeigten `print()`-Funktion, einfach anzuzeigen. Falls einer Funktion mehr als ein Argument übergeben wird, so werden die Argumente im einfachsten Fall als durch Kommas getrennte Listen angegeben, wobei hier die Bedeutung der einzelnen Argumente durch ihre Position bestimmt ist. Es ist allerdings manchmal auch möglich, **Argument-Werte-Paare** mit namentlich angegebenen **Schlüsselwort-Argumenten** zu übergeben, wobei die Argumente und Werte miteinander über das `=`-Zeichen verknüpft und somit einander zugeordnet werden. Bei dieser Form muss man sich also nicht die Reihenfolge der Argumente merken, was manchmal die Anwendung komplexerer Funktionen vereinfachen kann.

Wie wir schon gesehen haben, gibt die `print()`-Funktion auf dem Bildschirm eine Nachricht als Argument in Form einer **Zeichenkette** – hier in einfachen

Anführungszeichen angegeben – aus, wobei normalerweise automatisch auch ein Zeilenumbruch am Ende angefügt wird. Letzteres ist anders als bei vielen anderen Programmiersprachen, wo bei ähnlichen Funktionen üblicherweise der Programmierer selbst einen Zeilenumbruch innerhalb der Zeichenkette angeben muss. Die Ausgabe des Zeilenumbruchs lässt sich jedoch auch mittels eines zusätzlichen Schlüsselwort-Arguments `end` unterdrücken oder verändern, was wir später noch sehen werden.

Wiederum anders als bei den meisten Programmiersprachen können Anweisungen in Python auch interaktiv in der sogenannten **Python-Shell**, einer speziellen Umgebung zur Ausführung von Python, getestet werden. Dies kann unter Umständen sehr nützlich sein, um schnell etwas auszutesten, ohne es gleich in ein größeres Programm integrieren zu müssen. Die Python-Shell wird normalerweise über die Kommandozeile aufgerufen, ist aber auch in die WingIDE integriert, so dass man sie dort noch komfortabler verwenden kann. Dies probieren wir in der folgenden Übung gleich einmal aus.



Übung 6 – Anweisungen in der Python-Shell testen

Starten Sie die WingIDE.

Finden Sie das Unterfenster für die Python-Shell.

Geben Sie die Anweisungen von oben im Text nacheinander in die Python-Shell ein und sehen Sie sich die Ergebnisse genau an.

Modifizieren Sie die Zeichenketten, also das, was jeweils zwischen den einfachen Anführungszeichen steht, nach Belieben, und testen Sie die Ergebnisse.

Versuchen Sie auch, die zwei `print()`-Anweisungen als eine auszugeben, indem Sie zwei Argumente in derselben Klammer angeben, so wie dies oben beschrieben ist.

Wie unterscheidet sich diese Ausgabe von der mit den zwei getrennten `print()`-Anweisungen?

2.2 Variablen

Programme ‚berechnen‘ und verändern Werte. Um diese veränderlichen Werte speichern zu können, benötigt man Speicherorte als ‚Platzhalter‘, genannt **Variablen**. Diese Variablen müssen durch **Deklaration** einen möglichst sprechenden Namen zugewiesen bekommen, um sie später leicht ansprechen bzw. referenzieren zu können. Dies geschieht normalerweise durch **Initialisierung** in Form einer **Zuweisung**, z.B.

```
nachricht = 'Guten Morgen'
```

Hierbei erfolgt die Zuweisung durch `=`, wobei der Wert oder Ausdruck rechter Hand vom `=`-Zeichen der Variable auf der linken Seite zugewiesen wird, z.B.

```
c = a + b
```

vorausgesetzt, dass `a` und `b` auch initialisiert sind, da sonst ein Fehler auftritt. Mit einer solchen Zuweisung kann auch einer Variablen der Inhalt einer anderen zugewiesen werden, z.B.

```
nachricht1 = nachricht2
```

Bei erneuter Zuweisung wird der ursprüngliche Inhalt des Platzhalters überschrieben, bzw. teilweise, je nach Datentyp, auch eine komplett neue Variable angelegt und die vorherige ‚gelöscht‘. Sobald eine Variable einmal initialisiert ist, kann sie auch in anderen Arten von Anweisungen verwendet werden, z.B.

```
print(nachricht)
```

wo der Inhalt der Variablen, in dem Fall der Text *Guten Morgen*, mithilfe der `print()`-Funktion ausgegeben wird.

Variablenamen können aus Buchstaben, Unterstrichen und Ziffern bestehen, dürfen aber nicht mit einer Ziffer beginnen. Sie sollten möglichst auch nicht mit einem Unterstrich beginnen, da Python-interne Variablen dies tun. Umlaute und andere Sonderzeichen sollten im Allgemeinen ebenfalls vermieden werden. Variablen mit sprechenden Namen können aus mehreren Wörtern bestehen, wobei es zwei verschiedene Varianten gibt, um diese Wörter zu einem Variablennamen zu verknüpfen.

- `erstes_wort, wort_1` (Unterstrichform)
- `zweitesWort, wortZwei` (**camel case**-Form)

Welche der beiden Varianten Sie dabei verwenden, oder ob Sie sie mischen, bleibt im Allgemeinen Ihnen überlassen, zumindest insofern nur Sie Ihre eigenen Programme lesen, verstehen und bearbeiten müssen. Sollten Sie aber mit anderen im Team arbeiten, dann ist es am besten, sich auf bestimmte Konventionen zu einigen.

2.3 Datentypen - Übersicht

Python stellt standardmäßig schon verschiedene einfache und ‚zusammengesetzte‘ Datentypen zur Verfügung. Datentypen in Python sind Objekte (mehr dazu in Kapitel 7) mit zugehörigen **Methoden**, ähnlich den Funktionen, die wir oben kennengelernt haben. Diese ‚gehören‘ jedoch zu dem Objekt und werden über *Objektvariablenname.Methodenname()* aufgerufen, wie wir später noch sehr oft sehen werden.

Ich biete hier zunächst nur eine Kurzübersicht über die für uns bei der Arbeit mit Sprache wichtigsten Datentypen, die wir nach und nach noch im Detail besprechen werden. Dabei stehen jeweils die Typenbezeichnungen, und teilweise auch möglichen Werte, in Klammern hinter den deutschen Bezeichnungen.

Datentyp	Verwendungszweck
Zeichenketten (<code>str</code>)	für Wörter/Wortteile (engl. <i>strings</i>)
Zahlen (<code>int</code> , <code>float</code>)	ganze (Integer) und Fließkommazahlen
Binärwerte (<code>bool</code>)	wahr (<code>True</code>) oder falsch (<code>False</code>)
Listen (<code>list</code>)	ursprünglich unsortierte Kombinationen aus anderen Datentypen; Mehrfachvorkommen von Werten möglich
Tupel (<code>tuple</code>)	geordnete, unveränderliche Liste an Werten
Sets (<code>set</code>)	ungeordnete Kombinationen aus anderen Datentypen; Mehrfachvorkommen von Werten nicht möglich
Dictionaries (<code>dict</code>)	Kombinationen von Schlüssel- und Wertepaaren

Tabelle 1: Für linguistische Zwecke wichtigste Datentypen

Alle dieser Datentypen haben besondere Formen, um eine leere Variable ihres Typs anzulegen, welche wir später noch für die einzelnen Datentypen besprechen werden. Zusätzlich gibt es auch meist bestimmte Funktionen, die dies tun. Um jedoch eine leere Variable anzulegen, deren Datentyp entweder noch nicht bekannt ist, oder sie bewusst als nicht-initialisiert zu markieren, kann man ihr als ‚Wert‘ auch `None` zuweisen.

2.4 Zeichenketten und Zeichenkodierung

Zeichenketten stellen den wichtigsten Datentyp für sprachbezogene Aufgaben dar, da ja Sprache aus einer bedeutungsvollen Aneinanderreihung von Zeichen oder Buchstaben besteht. Der `str`-Datentyp wird mit verschiedenen Objektmethoden zur Verarbeitung dieser Zeichen verwendet. Hier wieder nur eine kurze Übersicht der wichtigsten, wobei komplementäre Methoden parallel, durch Schrägstriche getrennt, angegeben sind.

Methode(n)	Funktionalität
<code>split() / join()</code>	spaltet Zeichenketten an festgelegtem Trenner oder fügt sie damit zusammen;
<code>splitlines()</code>	spaltet Zeichenketten, die meist aus Dateiinhalten bestehen, in Zeilen auf;
<code>find() / count()</code>	findet Index einer Zeichenkette in einer anderen oder zählt, wie oft diese darin vorkommt;
<code>startswith() / endswith()</code>	prüft, ob eine Zeichenkette mit einer anderen anfängt oder aufhört.

Tabelle 2: Nützliche Zeichenkettenmethoden

Reguläre Zeichenkettenmethoden sind aber häufig zu einfach für komplexere sprachliche Analysen, weshalb wir später noch bessere Optionen für solche Zwecke besprechen werden.

Zeichenketten werden normalerweise durch Einschließen in einfache ('...') oder doppelte ("...") Anführungszeichen angegeben, aber dreifache Anführungszeichen ('''...''') können insbesondere für längere Zeichenketten mit mehreren Zeilenumbrüchen verwendet werden, auch als sogenannte *Docstrings* zur Dokumentation von Programmen. Falls Anführungszeichen innerhalb der Zeichenkette auftreten, dürfen diese entweder nicht gleich denen zur Markierung der Zeichenkette sein, oder müssen durch einen rückwärtsgerichteten Schrägstrich (\; engl. **backslash**) maskiert werden, z.B.

```
'Sie sagten "Hallo"'
```

oder

```
"Sie sagten \"Hallo\""
```

Zeichenketten, die Sonderzeichen, wie z.B. \, beinhalten, können durch ein vorangestelltes r als roh (engl. *raw*) markiert werden, was wir später noch öfter tun werden, nachdem wir besprochen haben, wofür dies nützlich ist.

Computer speichern Text zur internen Repräsentierung als Zahlen. Um Texte, die durch diese Zahlen kodiert sind, darstellen zu können, wurden Zeichensätze definiert, die die Zeichen als ein oder mehrere Bytes repräsentieren. Anfangs gab es nur einfache ‚Einzel-byte-Zeichensätze‘, von denen der bekannteste wahrscheinlich der ASCII-Zeichensatz ist. Dieser verwendet 7 Bits und kann damit bis zu 2⁷, also 128 Zeichen, darstellen, wobei nicht alle dieser Zeichen auch Buchstaben repräsentieren, sondern auch Steuerzeichen mit einbegriffen sind, welche die Ausgabe am Bildschirm oder beim Druck steuern. Da diese Anzahl an Zeichen sich jedoch als zu beschränkt erwies, wurden später zunächst Zeichensätze, wie z.B. Latin1 (ISO 8859-1) definiert, die mit 8 Bits 256 Zeichen darstellen können. Bei diesen einfachen Zeichensätzen sind die lateinischen Zeichen mit einzelnen Ziffer kodiert, die sich an den folgenden Positionen befinden.

Zeichentypen	Kodepositionen
Steuerzeichen	1–32
Interpunktion, Zahlen etc.	33–64
Großbuchstaben	65–90
Kleinbuchstaben	97–122
höhere Positionen	variabel, abhängig von Ortsumgebung

Tabelle 3: Latin1-Kodepositionen für Zeichen

Die Variabilität innerhalb der höheren Positionen führte jedoch zu Kompatibilitätsproblemen bei solchen älteren Kodierungssystemen (engl. **legacy encodings**), selbst für westliche Sprachen. Auch ist es unmöglich, viele asiatische Sprachen, wie z.B. das Chinesische, aufgrund der Vielzahl an Schriftzeichen, in Einzelbyte-Kodierungen zu speichern. Deshalb wurden zunächst Doppelbyte-Zeichensätze entwickelt, die jedoch weitere Inkompatibilitätsprobleme hervorriefen.

Die Lösung für diese Probleme liegt in dem Versuch, mittels eines universellen Kodierungsstandards namens **Unicode** so viele Zeichen wie möglich gleichzeitig kodieren und darstellen zu können. Allerdings existieren auch für Unicode verschiedene Typen, nämlich UTF-32, UTF-16 und UTF-8, die entweder mit einer festen oder variablen Länge von Bytes Tausende von Zeichen darstellen können. Ideal für die Korpuslinguistik ist UTF-8, welches bis zu 6 Bytes verwendet, wobei auch die einfachen lateinischen Zeichen günstigerweise die gleichen Positionen wie in ASCII oder Latin 1 belegen. Zudem entwickelt sich mittlerweile UTF-8 auch zum Standard bei der Kodierung von Webseiten und ist die Standardkodierung für die Auszeichnungssprache XML, die wir später für die linguistische Annotierung verwenden werden.

Der Code für ein Zeichen kann in Python mithilfe der `ord()`-Funktion abgefragt und, umgekehrt, mit der `chr()`-Funktion produziert werden. Wie wir aber oben anhand der Positionen gesehen haben, sind Zeichenketten eigentlich immer **case-sensitive**. Es wird also prinzipiell zwischen Groß- und Kleinbuchstaben unterschieden, wobei der Code für Großbuchstaben immer um 32 niedriger liegt als für den entsprechenden Kleinbuchstaben. Dies ist vor Allem wichtig für Vergleiche zwischen Zeichenketten/Wörtern oder deren Sortierung. Was für unsere Zwecke auch extrem nützlich ist, ist dass Python intern normalerweise UTF-8 verwendet, so dass es sehr einfach ist, auch deutsche Buchstaben zu repräsentieren.

Da wir jetzt genug über Variablen und Zeichenketten wissen, können wir schon eine etwas umfangreichere Übung machen, um auszuprobieren, wie man eigene Variablen anlegt, einander zuweist und wie man den Code eines Buchstabens herausfinden kann.



Übung 7 - Mit Zeichenkettenvariablen arbeiten

Legen Sie zwei Zeichenkettenvariablen für die Wörter `hier` und `ist` in der Python-Shell an, wobei Sie nach jeder Anweisung `↵` drücken müssen, um diese ‚abzuschicken‘. Verwenden Sie dazu selbstgewählte Variablennamen, und probieren Sie auch aus, was geschieht, wenn Sie versuchen, einen Variablennamen zu verwenden, der mit einer Ziffer beginnt.

Schreiben Sie dann eine `print()`-Anweisung mit zwei Argumenten, um beide Wörter über die Variablen auszugeben, wobei Sie beim zweiten Mal die Reihenfolge umdrehen. Was haben Sie hiermit aus sprachwissenschaftlicher Sicht simuliert? Versuchen Sie jetzt, denselben Effekt zu erreichen, indem Sie die Inhalte der zwei Variablen durch Neuzuweisungen vertauschen und dann wieder über `print()` ausgeben. Welches Problem tritt hierbei auf und warum?

Versuchen Sie, das Problem mithilfe einer dritten Variable namens `temp` zu lösen, wobei Sie zuerst die erste Variable wieder mit dem richtigen Wort neu instanzieren und dann den Inhalt des zweiten Wortes zunächst in `temp` zwischenspeichern. Stellen Sie zuletzt fest, welcher deutsche Buchstabe den Code 196 in UTF-8 hat.

2.5 Zahlen

Auch wenn Zeichenketten für die Arbeit mit Sprache am wichtigsten sind, so sind Zahlen immer noch nötig, um die Häufigkeiten, mit denen sprachliche Phänomene auftreten, auch zählen zu können. Bevor wir jedoch die für uns relevanten Datentypen für Zahlen besprechen, ist es wichtig, sich immer dessen bewusst zu sein, dass Python ein angelsächsisches Zahlenformat verwendet, so dass der Punkt und das Komma innerhalb von Zahlen jeweils die umgekehrte Bedeutung von der deutschen, also als Dezimal- und Tausendtrenner, haben.

Python bietet mehrere Arten von mathematischen Datentypen, von denen aber für uns nur zwei wichtig sind. Der Rest wird im Wesentlichen für naturwissenschaftliche Aufgaben benötigt. Hierbei verwenden wir den Typ `int` (Integer, Ganzzahl) zum einfachen Zählen von Auftretenshäufigkeiten sprachlicher Phänomene und `float` (Fließkommazahlen) zum Berechnen von Relationen oder relativen Frequenzen. Ein häufiger Fehler, der bei der Ausgabe von Zahlen zusammen mit Zeichenketten auftritt, ist, dass diese bei der Verknüpfung mit einer Zeichenkette nicht erst mithilfe der `str()`-Funktion in Zeichenketten umgewandelt werden. Das ist normalerweise unbedingt erforderlich, es sei denn, dass die Zahlen mithilfe spezieller Formatierungsanweisungen ausgegeben werden, auf die wir im Abschnitt 4.5 noch genauer eingehen werden.

Wie schon für die Zeichenketten, wollen wir auch hier zunächst einige wichtige Funktionen auflisten.

Funktion	Funktionalität
<code>int(zahl)</code>	wandelt Fließkommazahlen zur niedrigsten Basis (das heißt $1,9 \rightarrow 1$)
<code>round(zahl, n)</code>	rundet Fließkommazahlen zur nächsten Ganzzahl oder auf n Stellen

Tabelle 4: Wichtige Funktionen für Zahlen

Binärschalter/-Werte (`bool`) markieren, ob eine Bedingung erfüllt ist oder nicht, bzw. eine Eigenschaft eines Objektes gegeben. Bei `True` ist eine Bedingung erfüllt, bzw. liegt eine Eigenschaft vor, wohingegen bei `False` die Bedingung nicht erfüllt ist oder die Eigenschaft nicht vorliegt. Binärwerte werden normalerweise durch Bedingungsabfragen oder in Schleifen zurückgeliefert, während Binärschalter selbst gesetzt werden

können, um Zustände zu markieren, z.B. um anzuzeigen, dass etwas schon abgearbeitet wurde.

2.6 Operatoren

Operatoren ermöglichen es – wie der Name schon sagt – Operationen mit Daten auszuführen, wie z.B. die Zuweisung eines Wertes an eine Variable über den =-Operator. Sie können unter anderem mathematische, logische, oder Zeichenkettenoperationen auslösen, aber auch weitere, die wir, je nach Bedarf, später besprechen werden. Dabei sind manche Operatorsymbole ‚polysem‘, das heißt, sie können in verschiedenartigen Operationen auftreten, wobei ihre genaue Funktion dann vom Datentyp abhängt.

Tipp: Auch wenn Operatoren nicht immer von Variablen oder Werten getrennt werden müssen, bietet es sich zwecks Übersichtlichkeit an, dass man um sie herum Leerzeichen schreibt. Dadurch können die Operationen leichter erkannt werden und der Code wird besser lesbar.

2.6.1 Mathematische Operatoren

Die meisten mathematischen Operatoren dürften Ihnen aus dem Mathematikunterricht geläufig sein, wobei die Symbole in Python manchmal etwas anders sein können. Auch sind neben den regulären Operationen oft solche möglich, bei denen gleichzeitig eine Zuweisung stattfindet. Tabelle 5 listet die mathematischen Operatoren, ihre Funktion sowie kurze Beispiele dazu auf.

Operator	Funktion	Beispiel
+ (+=)	Addition (ohne/mit Zuweisung)	zahl = zahl + 3 (zahl += 3)
- (--)	Subtraktion (ohne/mit Zuweisung)	zahl = zahl- 3 (zahl -= 3)
* (*=)	Multiplikation (ohne/mit Zuweisung)	zahl = zahl * 3 (zahl *= 3)
**	Exponentialfunktion	zahl = zahl ** 2
/ (/=)	Division (ohne/mit Zuweisung)	zahl = zahl / 3 (zahl /= 3)
//	ganzzahlige Division (keine Kommastellen)	zahl = 5 // 3
%	Modulo (Rest der Division)	zahl = 5 % 3
==	Test auf Gleichheit (für Zahlenwerte)	zahl1 == zahl2
!=	Test auf Ungleichheit (für Zahlenwerte)	zahl1 != zahl2

< (<=)	kleiner (gleich)	zahl1 < zahl2 (zahl1 <= zahl2)
> (>=)	größer (gleich)	zahl1 > zahl2 (zahl1 >= zahl2)

Tabelle 5: Mathematische Operatoren

Übung 8 - Mit mathematischen Operatoren arbeiten

Testen Sie die mathematischen Operatoren, indem Sie die Beispiele aus der vorhergehenden Tabelle in der Python-Shell ausprobieren. Dabei können Sie zuerst die Variable `zahl`, bzw. später die Variablen `zahl1` und `zahl2`, nach Belieben initialisieren und ebenso beliebig oft ändern. Den jeweiligen Wert nach der Operation können Sie einfach in der Shell ausgeben, indem Sie `zahl` eintippen und die ↵-Taste drücken. Achten Sie dabei jeweils auf den **Rückgabewert** (das heißt das Ergebnis) in der Shell, und versuchen Sie immer zu verstehen, was er bedeutet.

2.6.2 Zeichenkettenoperatoren

Wie schon angesprochen, werden manche Operatoren für mehrere Arten von Operationen verwendet. Deshalb werden einige der Symbole, die wir schon für mathematische Operationen gesehen haben, auch für mehr oder weniger ähnliche Zeichenkettenoperationen angewandt. Tabelle 6 listet wieder die Operatoren, deren Funktion und kurze Beispiele auf.

Operator	Funktion	Beispiel
+	Konkatenation	nachricht = 'Guten ' + tageszeit + '!'
*	Wiederholung	print('-' * 50)
%	Einfachste formatierte Ausgabe (in allen Python-Versionen vorhanden)	print('5/3 mit 2 Nachkommastellen ist %.2f' % (5/3))
==	Vergleich auf Gleichheit	'Ab' == 'ab'
!=	Vergleich auf Ungleichheit	'Ab' != 'ab'
<	kleiner als (bezüglich Zeichencode)	'Ab' < 'ab'
>	größer als (bezüglich Zeichencode)	'Ab' > 'ab'

Tabelle 6: Zeichenkettenoperatoren



Bei der Konkatination handelt es sich um eine Verknüpfung von Zeichen, wobei in Python – anders als in anderen Programmiersprachen – eine komplett neue Kette angelegt und die ursprüngliche Variable gelöscht wird. Anders als bei der Ausgabe mit `print()`, wo automatisch ein Leerzeichen zwischen den Argumenten erzeugt wird, müssen bei der Konkatination von Wörtern die Leerzeichen innerhalb einer Kette (oder als getrennte Kette) angegeben werden, da ansonsten die Wörter einfach zusammengefügt werden.

Die Wiederholung gibt einfach dasselbe Zeichen, oder dieselbe Kette, mehrmals hintereinander aus, wodurch man z.B. Trennlinien bei der Ausgabe von Ergebnissen erzeugen kann.

Bei der formatierten Ausgabe mit dem `%`-Operator können ein oder mehrere Variablen oder, wie im obigen Beispiel, auch Ergebnisse von Ausdrücken mit speziellen Formatierungen in eine Zeichenkette interpoliert werden. Bei Zahlen erlaubt dies auch die Interpolation ohne vorherige Konversion mittels der `str()`-Funktion. Mehr dazu und zu anderen, neueren Möglichkeiten der Formatierung im Abschnitt 4.5.

Bei den Tests auf Gleichheit oder Ungleichheit werden immer ganze Ketten auf exakte Übereinstimmung überprüft und ein boolescher Wert, entweder **True** (wahr) oder **False** (falsch), zurückgeliefert. Bei Größenvergleichen werden die Ketten jeweils zeichenweise von links nach rechts miteinander anhand der Zeichencodes verglichen.



Übung 9 – Zeichenkettenoperatoren austesten

Testen Sie die Zeichenkettenoperatoren, indem Sie die Beispiele aus der vorhergehenden Tabelle in der Python-Shell ausprobieren.

Experimentieren Sie wieder, indem Sie die Beispiele selbständig verändern.

Versuchen Sie insbesondere zu verstehen, worauf die Ergebnisse bei den Vergleichen beruhen.


2.6.3 Logische Operatoren

Logische Operatoren stellen entweder Verknüpfungen oder die Negation von Bedingungen dar. Dafür werden die entsprechenden englischen Ausdrücke verwendet, da dies in Programmiersprachen so üblich ist. Tabelle 7 zeigt wieder die Optionen.

Operator	Funktion	Beispiel
and	verknüpft Bedingungen/Vergleiche, das heißt beide müssen <code>True</code> zurückliefern, sonst <code>False</code>	<code>a != 0 and b < 5</code>
or	unterscheidet zwischen Alternativen, das heißt eine von beiden muss <code>True</code> zurückliefern, sonst <code>False</code>	<code>wort1 != 'vielleicht' or wort2 == 'ja'</code>
not	negiert Bedingungen	<code>not bedingung_erfuellt</code>

Tabelle 7: Logische Operatoren

Logische Operatoren werden normalerweise mit Bedingungsabfragen verwendet. Wie genau, werden wir ab Abschnitt 3.4 anhand von realistischeren Beispielen sehen.



Übung 10 - Logische Operatoren testen

Testen Sie die logischen Operatoren, indem Sie die Beispiele aus der vorhergehenden Tabelle in der Python-Shell ausprobieren.

Bestimmen Sie dabei wieder die Werte der Variablen selbst, und verändern Sie diese so lange, bis Sie jeweils mindestens einmal ein `True` und einmal `False` als Ergebnisse erreicht haben.

Versuchen Sie dabei immer zu verstehen, wie die verschiedenen Werte sich auf die Bedingungen auswirken.

2.7 Skripte/Programme erstellen

Die Python-Shell ist zwar äußerst nützlich zum Testen von Code-Schnipseln und der Ausführung bzw. dem Testen kurzer Anweisungen, aber nicht geeignet für längere Programme, die vielleicht auch wiederholt laufen sollten. Die Lösung für dieses Problem besteht in der Erstellung speicherbarer Programme in einem Editor oder einer IDE, welche dann über den Python-Interpreter direkt ausgeführt werden können. Für solche Programme verwendet man dann normalerweise die Dateiendung (Extension) `.py`.

Unter Windows wird die Endung bei der Python-Installation normalerweise mit dem Interpreter verknüpft, bzw. Python im Pfad ausführbarer Verzeichnisse eingetragen, wie wir es unter 1.3 gesehen hatten. Unter Linux oder MacOS erfolgt diese ‚Verknüpfung‘ über die sogenannte **Shebang-Zeile**,

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

die dem Betriebssystem zeigt, mit welchem Programm die Datei ausgeführt werden soll. Allerdings müssen hier die Programme meist erst ausführbar gemacht werden,

was mithilfe der Anweisung `chmod +x Dateiname` auf der Kommandozeile oder über die entsprechenden Optionen im Dateimanager oder Finder erreicht wird.

Der **Quellcode** (engl. *source code*) für Python-Programme muss als Rohtext vorliegen, die idealerweise schon UTF-8-kodiert sind, wobei Abweichungen von diesem de-facto Standard über Angabe der Kodierung, z.B.

```
#-*- coding:iso-8859-15 -*- oder # -*- coding:latin-1 -*-
```

markiert werden sollten. Allerdings spricht eigentlich nichts dafür, dass sie überhaupt eine andere Kodierung als UTF-8 verwenden. Die Kodierung sollte auch zur Sicherheit im Editor oder der IDE eingestellt werden, so wie wir dies nach der Installation der WingIDE Personal getan haben.

Was das Layout von Programmen angeht, so werden Leerzeilen zwischen Anweisungen normalerweise ignoriert und können dafür verwendet werden, den Programmtext übersichtlicher machen. Allerdings ist Python sehr penibel, was Einrückungen angeht. Weshalb, werden wir bald noch erfahren.

Die WingIDE umbricht normalerweise auch längere Zeilen automatisch bei der Darstellung, damit Sie den kompletten Code auch bei längeren Zeilen noch lesen können, ohne scrollen zu müssen. Zur übersichtlicheren Darstellung von Code, zum Beispiel beim Ausdrucken, kann man längere Zeilen manuell umbrechen. Dabei setzt man die entsprechenden Teile am besten zur Gruppierung in runde Klammern, insofern sie es nicht sowieso schon sind, und umbricht vor Operatoren, wie im folgenden Beispiel:

```
print('Dies ist ein Zeilenumbruchtest für eine etwas längere'
      + 'Zeile, die sonst schwer zu lesen wäre.')
```

Das obige Beispiel lässt sich allerdings noch vereinfachen, da, wenn zwei Zeichenketten hintereinander stehen, diese auch automatisch konkateniert werden. Dies erlaubt es uns, längere Zeichenketten einfach auf mehreren Zeilen zu schreiben. Allerdings kann dies bei übertriebener Verwendung den Code auch undeutlicher machen, weshalb man es eher spärlich verwenden sollte, da ja keine Operatoren zu sehen ist.

Handelt es sich bei dem geklammerten Ausdruck um eine Liste mit mehreren Argumenten, so kann man die einzelnen Elemente auch untereinander schreiben. Dabei sollte jedoch das Komma zwischen den Elementen auf der unvollständigen Zeile stehenbleiben, um den Listencharakter deutlicher zum Ausdruck zu bringen.

Außer diesen drei Möglichkeiten gibt es noch eine letzte Option, die jedoch normalerweise nicht empfohlen wird, nämlich die Zeile mittels eines `\` am Ende der unvollständigen Zeile als solche zu markieren. Dabei kann es allerdings auch zu Fehlern kommen, wenn man aus Versehen noch ein Leerzeichen hinter der Markierung einfügt, was im Editor oft nicht deutlich zu sehen ist. Ich werde, um den im Buch abgedruckten Programmcode lesbarer zu machen, je nach Bedarf die geeignetste Option verwenden. Wenn Zeilen im Code vorkommen, die eindeutig und korrekt als unvollständig markiert sind, ist es auch möglich, die Folgezeilen so einzurücken, dass sie übersichtlicher

werden, auch wenn Python sonst auf Einrückungen sehr empfindlich reagiert. Auch davon werde ich zwecks Übersichtlichkeit Gebrauch machen.

Übung 11 - Variablen tauschen als Programm

Schreiben Sie die Vertauschungsoperation der zwei Zeichenkettenvariablen aus Übung 7 als ein Programm, welches Sie neu in der WingIDE erstellen.

Vergessen Sie nicht, eine Shebang-Zeile mit einzufügen, damit Ihr Programm (theoretisch) auch unter MacOS/Linux lauffähig ist, selbst wenn Sie unter Windows arbeiten.

Um erkennen zu können, welche Werte die Variablen jeweils vor und nach der Vertauschung haben, fügen Sie soweit nötig entsprechende `print()`-Anweisungen hinzu.

Nennen Sie das Programm `01_tausch.py` und speichern Sie es in dem Verzeichnis ab, das Sie für den Kurs verwenden.

Sollten Sie das Verzeichnis neu anlegen, verwenden Sie am besten keine Leerzeichen im Verzeichnisnamen.

Falls erforderlich (unter MacOS/Linux), setzen Sie die nötige Berechtigung zum Ausführen der Datei, wie oben angesprochen. Dies müssen Sie dann natürlich auch für alle weiteren Programme tun, die wir noch schreiben werden.

Öffnen Sie eine Kommandozeile in Ihrem Verzeichnis und führen Sie das Programm aus, wobei Sie nach dem Programmnamen immer die Enter-Taste drücken müssen.

Windows: `01_tausch.py`

MacOS/Linux: `python ./01_tausch.py`

2.8 Code Kommentieren

Guter Programmcode sollte normalerweise auch gut kommentiert sein! Dabei erleichtern Kommentare im Code das Verständnis für andere Programmierer, aber auch für einen selbst im Umgang mit Code, den man lange nicht bearbeitet hat. Wie dies grob funktionieren kann, haben Sie hoffentlich schon aus meinen Beschreibungen zur vorherigen Übung gesehen. Dort hatte ich die Erklärungen dazu, was im Programm vorgegangen ist, immer nach dem jeweiligen Programmtext angefügt. Das geschieht bei der Kommentierung im Code aber üblicherweise vorher, so dass man erst liest, was im nächsten Schritt passieren sollte. So gewinnt man einen besseren Überblick über die Programmstruktur und kann damit auch die verwendeten Variablen und Funktionen/Methoden einfacher verstehen.

Zu Beginn Ihrer Programmierkarriere sollten Sie sich am besten angewöhnen, eher zu viel als zu wenig zu kommentieren, um Ihr eigenes Verständnis für bestimmte Programmieraufgaben zu überprüfen bzw. für sich selbst in geeigneter Form zum



Ausdruck zu bringen. Wenn Sie etwas mehr Erfahrung haben, können Sie dies natürlich wieder reduzieren.

Für Kommentare gibt es im Wesentlichen zwei Formen. Einfache Kommentare, die im Wesentlichen nicht länger als eine Zeile sind, beginnen mit einem Rautenzeichen `#`. Das steht meist am Zeilenanfang (vgl. Shebang-Zeile oder Kodierungsangabe), oder folgt einer Anweisung, wenn der Kommentar sehr kurz ist. Dreifache Anführungszeichen leiten mehrzeilige Kommentare ein und können als **Docstrings** zur (automatisch generierten) Dokumentation von Objekten und Funktionen verwendet werden.



Übung 12 – Kommentieren üben

Kommentieren Sie das Programm, das Sie in der vorhergehenden Übung geschrieben haben, so dass deutlich wird, was genau in jedem Schritt geschieht.

Verwenden Sie dabei nur die einfache Kommentarform, und schreiben Sie die Kommentare so, wie Sie sie selbst am besten verstehen können. Aber berücksichtigen Sie idealerweise auch, inwieweit dies für Andere verständlich ist.

Fügen Sie außerdem Ihren Namen als Autor(in) zu Beginn des Programms – aber erst nach der Shebang-Zeile – als Kommentar ein und dokumentieren Sie dort auch, wann Sie das Programm erstellt und zuletzt bearbeitet haben. In diesem anfänglichen Kommentarabschnitt kann man auch eine kurze Beschreibung der beabsichtigten Funktionalität des Programms einfügen.

In diesem Kapitel haben wir die ersten Grundlagen der Programmierung in Python kennengelernt. Dabei haben wir besprochen, was es mit Anweisungen und Variablen auf sich hat, welche einfachen Datentypen es in Python gibt, wie man Operationen mit diesen Datentypen ausführt und welche, sowie ebenfalls, wie man Programme schreibt und sinnvoll kommentiert. Im nächsten Kapitel wenden wir uns etwas fortgeschritteneren, aber dennoch relativ grundlegenden, Konzepten zu.



2.9 Lösungen zu den Aufgaben

Lösung 6 – Anweisungen in der Python-Shell testen

Das Unterfenster für die eingebaute Python-Shell in der WingIDE befindet sich im unteren Arbeitsbereich auf der rechten Seite und wird über den gleichnamigen Reiter aktiviert. Daraufhin kann man dann Anweisungen hinter dem `1>>>`-Prompt eintippen, wobei sich nach jeder Eingabe die Ziffer vor den drei nach rechts gerichteten spitzen Klammern erhöht, da die Anzahl der Eingaben mitgezählt wird.

Wenn Sie die Ausgaben der `print()`-Anweisungen genau ansehen, vorausgesetzt, dass Sie beim Eintippen keinen Fehler gemacht haben, sollten Sie sehen, dass die Zeichenketten, die Sie als Argumente eingegeben haben, ohne die Anführungszeichen ausgegeben werden, da diese ja nur dazu benötigt werden, die Zeichenketten als

solche für Python zu markieren. Außerdem stellt die Shell den Funktionsnamen, die Klammern, und die Zeichenketten in unterschiedlichen Farben dar, was den Überblick erleichtert. Sobald Sie anfangen, eine Anweisung einzugeben, wird die Shell auch beginnen, Ihnen Vorschläge zur Ergänzung zu machen, bei denen Sie mithilfe der Pfeil-Tasten zum passenden Eintrag navigieren können und diesen durch Drücken der Tabulator-Taste annehmen.

Mögliche Fehler bei der Eingabe könnten sein, dass Sie den Namen der Funktion falsch schreiben, z.B. `prunt` statt `print` und dann die Fehlermeldung

```
builtins.NameError: name 'prunt' is not defined
```

bekommen, dass Sie versehentlich die schließende Runde Klammer löschen/vergessen, woraufhin die Shell Ihnen ... anzeigen wird, bis Sie die Klammer hinzugefügt haben, oder, dass Sie eines der Anführungszeichen bei der Zeichenkette vergessen haben, was zu einem Fehler wie

```
Syntax Error: EOL while scanning string literal: <string>,
line 1, pos 22
```

führen wird. Hier gibt Ihnen der Interpreter auch gleich noch die mögliche Fehlerposition am Ende der Meldung aus.

Wenn Sie beide Ausgaben mithilfe von zwei Argumenten innerhalb einer `print()`-Anweisung erzeugen, sollten Sie feststellen, dass Python nicht, wie bei der ersten Version, zwei Zeilen ausgibt, sondern die beiden Argumente hintereinander auf der gleichen Zeile, aber automatisch durch ein Leerzeichen getrennt, ausgibt, da bei jeder `print()`-Anweisung jeweils nur ein Zeilenumbruch angefügt wird.

Lösung 7 - Mit Zeichenkettenvariablen arbeiten

Am besten verwenden Sie zwei Variablen wie `wort1` und `wort2`, oder `erstesWort` und `zweitesWort`, je nachdem, welchen Stil sie bevorzugen. Im Prinzip dürfte hierbei fast nichts falsch zu machen sein, es sei denn, dass Ihnen Fehler mit den Anführungszeichen bei der Initialisierung der Zeichenketten unterlaufen.

Wenn Sie jedoch austesten was passiert, wenn Sie einen Variablennamen, der mit einer Ziffer beginnt, zu erstellen versuchen, dann sollten Sie eine der folgenden Fehlermeldung bekommen:

```
Syntax Error: invalid syntax: <string>, line 1, pos 2
```

oder

```
Syntax Error: invalid decimal literal: <string>, line 1, pos 2
```

Bei Ersterer hatte ich versucht `leswort` und bei der Zweiten `1_eswort` als Variablennamen zu deklarieren.

Bei korrekter Deklaration und Initialisierung sollten Ihre ersten zwei Zeilen in etwa wie folgt aussehen, wobei Sie jedoch nach jeder Anweisung auch `↵` drücken müssen:

```
erstes_wort='hier'
zweites_wort='ist'
```

Die Ausgabe von zwei Argumenten hatten wir schon in der letzten Übung erprobt, weshalb es Ihnen keine Probleme bereiten sollte, die nächsten zwei Zeilen so zu formulieren.

```
print(erstes_wort, zweites_wort)
print(zweites_wort, erstes_wort)
```

Wie Sie hoffentlich aus der Ausgabe unschwer erkennen können, haben wir hier eine syntaktische Inversion simuliert, da bei der ersten Ausgabe das Lokaladverb *hier* normalerweise die Subjektposition eines einfachen Deklarativsatzes einnehmen würde, und die Verbform *ist* an zweiter Stelle, wobei in der zweiten Ausgabe diese Positionen getauscht werden, so wie das normalerweise bei der entsprechenden Entscheidungsfrage der Fall wäre. Selbstverständlich läge im Normalfall ein kompletter Satz samt Interpunktion vor, wobei auch das erste Wort im Satz per Konvention großgeschrieben sein sollte. Da wir aber bisher noch nicht wissen, wie wir dies erreichen können, werden wir das Beispiel später schrittweise überarbeiten, bis wir eine vollständige Ausgabe erreicht haben, die allen Konventionen entspricht.

Ihr erster Versuch, die Variablen zu vertauschen wird wahrscheinlich daran scheitern, dass die Inhalte in den Platzhaltern für Variablen jeweils überschrieben werden, falls Sie einfach auf die folgende Art versuchen, die Variablen zu tauschen:

```
erstes_wort = zweites_wort
zweites_wort = erstes_wort
```

Hier wird leider bei der Ausgabe nur *ist ist* herauskommen, da mittels der ersten ‚Tauschanweisung‘ der Inhalt der Variable `zweites_wort`, nämlich *ist*, in der ersten Variable gespeichert wird, aber dabei der bisherige Inhalt, *hier*, auch einfach überschrieben wird. Wenn Sie dann den Inhalt von `zweites_wort` mit dem von `erstes_wort` überschreiben, beinhalten beide Variablen jetzt leider das gleiche Wort.

Wie uns dieses Beispiel zeigt, ist das, was uns mithilfe der Vertauschung der Argumente in der `print()`-Anweisung und im Kopf ‚konzeptuell‘ sehr einfach gelingt, nicht immer direkt auf dem Computer möglich. Deshalb sollten wir bei jeder Zuweisung an eine Variable sicher sein, dass wir auch den gewünschten Inhalt zuweisen und nicht versehentlich einen Datenverlust verursachen.

Mithilfe der Hilfsvariable `temp`, für ‚temporär‘, können wir den Inhalt der zweiten Variable, bevor wir diese überschreiben, erst einmal zwischenspeichern und diesen dann von `temp` aus in `erstes_wort` kopieren, so dass unsere Vertauschungsaktion dann wie folgt aussehen würde:

```
temp = erstes_wort
erstes_wort = zweites_wort
zweites_wort = temp
```


Jedoch würde uns Python es auch wieder erlauben, diese Aufgabe auf einfachere und effizientere Weise zu erledigen. Dazu müssen wir allerdings etwas über Tupel lernen, zu denen wir erst im nächsten Kapitel kommen.

Um den letzten Teil der Übung auszuführen, sollten Sie selbstverständlich die `chr()`-Funktion verwenden, wobei Sie entweder einfach in der Python-Shell `chr(196)` eingeben können oder `print(chr(196))`. Im ersten Fall wird Ihnen dabei 'Ä' ausgegeben und im zweiten Ä, was den Unterschied zwischen der reinen Ausgabe des Ergebnisses der `chr()`-Funktion in der Shell und deren Verwendung als Argument für die `print()`-Funktion deutlich macht. Bei der reinen Ausgabe erhalten Sie nämlich als Rückgabewert eine Zeichenkette, die aus einem einzelnen Buchstaben besteht, und bei der Verwendung von `print()` die Ausgabe von Text. Dabei zeigt uns die zweite Form auch, dass wir nicht nur Zeichenketten oder Variablen als Argumente an Funktionen übergeben können, sondern auch wieder Funktionen, die ihre eigenen Argumente haben. Dadurch entstehen verschachtelte Anweisungen, die jeweils von rechts nach links ausgewertet werden, so dass in unserer zweiten Form zunächst die `chr()`-Funktion mit dem Argument 196 ausgewertet wird, die ihr Ergebnis dann an die `print()`-Funktion übergibt.

Lösung 8 - Mit mathematischen Operatoren arbeiten

Eigentlich sollten bei dieser Übung keine Probleme auftreten, es sei denn, Sie vergessen, ein Zeichen einzutippen oder versuchen bei einem der Divisionstypen, `/`, `//`, oder `%`, durch 0 zu teilen, was zu der Fehlermeldung

```
builtins.ZeroDivisionError: (integer) division (or modulo) by zero
```

führt, da Divisionen durch 0 nicht erlaubt/mathematisch sinnvoll sind, oder Sie vergessen haben, eine Variable zu initialisieren. Die Teile der Fehlermeldung in runden Klammern treten dabei bei ganzzahliger oder Modulo-Division auf. Bei Division durch erlaubte Werte liefert die normale Division, insofern nicht sowieso glatt geteilt werden kann, ein Ergebnis mit Kommastellen, die ganzzahlige rundet nach unten ab, und die Modulo-Operation, die Ihnen unter Umständen noch nicht bekannt war, das, was eventuell nach der Division übrig bleibt. Modulo-Operationen werden z.B. dafür verwendet, jedes *n*te Element, also zweite, dritte, vierte, fünfte etc., aus einer Liste abzuarbeiten, z.B. nur jedes zweite oder dritte Wort.

Wichtig ist auch, zu beachten, dass Operatoren, die aus zwei Zeichen bestehen, immer zusammengeschrieben werden müssen, da sonst Fehler auftreten. Sie können dies gerne austesten, indem Sie z.B. versuchen, die Exponentialfunktion zu verwenden, dabei aber ein Leerzeichen zwischen den beiden Sternchen einfügen.

Die Operationen mit Zuweisung setzen voraus, dass die Variable auf der linken Seite existiert und die Operation darauf ausgeführt wird. Dies ist also nur eine Kurzschreibung der jeweils längeren Form.

Die Tests auf Gleichheit oder Ungleichheit liefern entweder `True` oder `False` zurück, je nach Gegebenheiten. Ein häufiger Fehler beim Testen auf Gleichheit, der selbst

erfahrenen Programmierern und Programmierern noch unterläuft, ist es, eines der beiden `--`-Zeichen zu vergessen, was dann zu einer Zuweisung anstelle von einem Vergleich führt. Tritt dieser Fehler innerhalb einer Bedingungsabfrage auf, die wir im nächsten Kapitel besprechen werden, dann kann dies leicht zu Endlosschleifen führen, da dann die Bedingung u.U. immer gegeben ist!

Bei der Konkatination handelt es sich um eine Verknüpfung von Zeichen, wobei in Python – anders als in anderen Programmiersprachen – eine komplett neue Kette angelegt wird. Anders als bei der Ausgabe mit `print()`, wo automatisch ein Leerzeichen zwischen den Argumenten erzeugt wird, müssen aber bei der Konkatination von Wörtern die Leerzeichen innerhalb einer Kette, oder als getrennte Kette, eingegeben werden, da ansonsten die Wörter einfach zusammengefügt werden.

Die Wiederholung gibt einfach dasselbe Zeichen, oder dieselbe Kette, mehrmals hintereinander aus, wodurch man z.B. Trennlinien bei der Ausgabe von Ergebnissen erzeugen kann.

Bei der formatierten Ausgabe mit dem `%`-Operator können ein oder mehrere Variablen oder, wie im obigen Beispiel, auch Ergebnisse von Ausdrücken mit speziellen Formatierungen in eine Zeichenkette interpoliert werden. Bei Zahlen erlaubt dies auch die Interpolation ohne vorherige Konversion mittels der `str()`-Funktion. Mehr dazu und zu anderen Möglichkeiten der Formatierung erfahren wir später.

Bei den Tests auf Gleichheit oder Ungleichheit werden immer ganze Ketten auf exakte Übereinstimmung überprüft und ein boolescher Wert zurückgeliefert. Bei Größenvergleichen werden die Ketten jeweils zeichenweise von links nach rechts miteinander anhand der Zeichencodes verglichen.

Lösung 9 – Zeichenkettenoperatoren austesten

Beim Testen der Konkatination mit dem angegebenen Beispiel müssen Sie die Variable `tageszeit` erst einmal mit einer passenden Bezeichnung für eine Tageszeit initialisieren, bevor Sie den Ausdruck auf der rechten Seite der Variable `nachricht` zuweisen und diese ausgeben können. Sonst kann hier außer den üblichen Fehlern bei der Erzeugung von Zeichenketten eigentlich nichts schiefgehen, bis auf vielleicht, dass Sie das Leerzeichen, welches zwischen *Guten* und der Variable für die Tageszeit stehen sollte, vergessen.

Bei der Wiederholung sollten Sie mit verschiedenen Zeichen arbeiten und auch einmal ganze Ketten eingeben, um zu beobachten, wie das Ergebnis aussieht. Den Ausdruck für die formatierte Ausgabe können Sie sinnvoll anpassen, indem Sie die Anzahl der Kommastellen, die anfänglich mittels `.2f` auf zwei Kommastellen festgelegt wurde, sowie die dazu passenden anderen Informationen, verändern. Die Rechenoperation kann gleichermaßen angepasst werden.

Die Vergleiche auf Gleichheit und Größe dürften normalerweise auch keine überraschenden Ergebnisse liefern, aber Sie sollten unter Umständen einmal austesten, was passiert, wenn Sie Ketten unterschiedlicher Länge, die sonst aber die gleichen Zeichen beinhalten, verwenden.

Lösung 10 - Logische Operatoren testen

Wie schon vorher, wenn Sie mit Variablen gearbeitet haben, können Sie natürlich nicht einfach die Anweisungen aus der Tabelle eintippen, ohne vorher die Variablen auch sinnvoll belegt zu haben. Das heißt, Sie müssen die Variablen `a` und `b` jeweils mit Zahlen initialisieren, `wort1` und `wort2` mit Wörtern, und `bedingung_erfüllt` entweder mit `True` oder `False`. Ansonsten müssen Sie verstehen, was hier logisch geschieht, wenn Sie mit den vorgegebenen Werten arbeiten oder diese verändern.

Lösung 11 - Variablen tauschen als Programm

Falls Sie sich beim Eintippen des Programmnamens nicht vertippt haben, und die Berechtigungen unter Linux/auf dem Mac richtig gesetzt sind, sollte es beim Starten des Programmes keine Probleme geben, es sei denn, dass Sie noch Fehler im Programmcode haben. Das fertige Programm sollte dann in etwa wie folgt aussehen, wobei ich einzelne Programmteile immer danach getrennt kommentiere.

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

Als erste Zeile sollte die Shebang-Zeile im Programm stehen, damit unter Linux oder auf dem Mac der richtige Python-Interpreter gefunden werden kann. Unter Windows sollte das Programm, wie schon angesprochen, über die Extension mit dem Interpreter verknüpft sein, weshalb diese Zeile einfach als Kommentar interpretiert und übersprungen werden kann.

```
wort1 = 'hier'
wort2 = 'ist'
```

Die Deklaration und Initialisierung der beiden Variablen hatten wir vorher schon geübt, und wenn Sie dabei keine Tippfehler haben, sollten diese keine Probleme bereiten.

```
print('Vor Vertauschung, Anfang Deklarativsatz: Wort 1=' +
      wort1 + '; Wort 2=' + wort2)
```

Die `print()`-Anweisung gibt hier einfach die beiden Variableninhalte, zusammen mit einer passenden Beschreibung, aus.

```
temp = wort1
```

Hier wird der Inhalt von `wort1` in `temp` zwischengespeichert.

```
wort1 = wort2
```

In diesem Schritt wird Wert von `wort2` der Variable `wort1` zugewiesen, so dass der Inhalt von `wort1` jetzt `ist` ist.

```
wort2 = temp
```

Hier wird der in `temp` zwischengespeicherte ursprüngliche Wert von `wort1` dann `wort2` zugewiesen, so dass `wort2` jetzt hier ist.

```
print('Nach Vertauschung, Anfang Interrogativsatz: Wort 1=' +
      wort1 + '; Wort 2=' + wort2)
```

Zum Schluss wird noch das Ergebnis der Vertauschung mit einer geeigneten Erklärung ausgegeben, so dass die Ausgabe, die Sie zu sehen bekommen, ungefähr folgendermaßen sein sollte:

```
Vor Vertauschung, Anfang Deklarativsatz: Wort 1=hier; Wort 2=ist
Nach Vertauschung, Anfang Interrogativsatz: Wort 1=ist; Wort 2=hier
```

Lösung 12 – Kommentieren üben

Bei dieser Übung kann eigentlich fast nichts schiefgehen, es sei denn, Sie fügen irgendwo ein Kommentarzeichen an einer falschen Stelle ein, so dass Teile Ihres Codes auskommentiert werden und dann das Programm nicht mehr läuft. Falls dies nicht der Fall ist, so sollte Ihr kommentiertes Programm dann in etwa so aussehen, wobei variable Elemente, die Sie selbst einfügen sollten, kursiv gedruckt sind:

```
#!/usr/bin/env python3
# 01_tausch.py
# Autor(in): Ihr Name
# Programm zum Vertauschen zweier Variablen,
# um syntaktische Inversion zu simulieren
# erstellt: Erstellungsdatum
# zuletzt bearbeitet: Bearbeitungsdatum

# Woerter deklarieren & instanziiieren
wort1 = 'hier'
wort2 = 'ist'
print('Vor Vertauschung, Anfang Deklarativsatz: Wort 1='+
      wort1 + '; Wort 2=' + wort2)
# Inhalt von wort1 zwischenspeichern
temp = wort1
# Wert von wort2 wort1 zuweisen
wort1 = wort2 # wort1 jetzt ist
# zwischengespeicherten Wert wort2 zuweisen
wort2 = temp # wort2 ist jetzt hier
print('Nach Vertauschung, Anfang Interrogativsatz: Wort 1=' +
      wort1 + '; Wort 2=' + wort2)
```



3 Grundlagen der Programmierung II - Zusammengesetzte Datentypen, Interaktion und Kontrollfluss

Dieses Kapitel beschreibt weitere, schon etwas fortgeschrittenere, Grundkonzepte der Programmierung. Unter anderem werden hier zusammengesetzte Datentypen eingeführt, gezeigt, wie man einfache Interaktion mit Programmen und Benutzern erzielt, Lösungsstrategien entwickelt und mit ‚sich schlecht benehmenden‘ Programmen umgeht. Zudem erfahren wir, wie Bedingungen abgefragt werden und Aktionen mithilfe von Schleifen wiederholt mit verschiedenen Daten oder Variablenwerten ausgeführt werden können.



3.1 Zusammengesetzte Datentypen

Zusammengesetzte Datentypen sind Behälter für mehrere (einfache) Datentypen, wobei diese Elemente meist, aber nicht immer, veränderlich sind. Wir können hier zwischen folgenden Typen unterscheiden.

Datentyp	Beschreibung
Listen (<code>list</code>)	veränderliche Folgen von Elementen, die mehrfach auftreten können
Tupel (<code>tuple</code>)	unveränderliche Folgen von Elementen, die mehrfach auftreten können
Sets (<code>set</code>)	veränderliche Folgen von Elementen, die Unikate sein müssen
Dictionaries (<code>dict</code>)	veränderliche Listen von Schlüssel-Werte-Paaren, die keine feste Reihenfolge haben

Tabelle 8: Typen zusammengesetzter Datentypen

Bei einigen Datentypen sind die Elemente auch in einer festen Reihenfolge abgespeichert, so dass man diese als **Sequenzen** bezeichnet. Dazu zählen Listen, Tupel und Zeichenketten, auch wenn letztere eigentlich einen einfachen Datentyp darstellen. Auf die einzelne Elemente von Sequenzen kann normalerweise über deren Indexpositionen zugegriffen werden, um diese abzufragen oder verändern. Zu beachten ist hierbei, dass die Indexpositionen in Programmiersprachen wie Python immer mit 0 anstatt 1 beginnen, was insbesondere bei Programmieranfängern oft zu Fehlern führt! Da Dictionaries aus Effizienzgründen bei der Speicherung normalerweise keine feste

Reihenfolge haben, gibt es dort auch keine Indexpositionen, sondern der Zugriff erfolgt über den Schlüssel. Die Besonderheit bei Tupeln ist, dass sie, sobald sie einmal angelegt sind, nicht mehr verändert werden können. Mehr zu den jeweiligen Datentypen besprechen wir jeweils bei Bedarf.

3.1.1 Listen

Listen stellen im Prinzip die einfachste Form von Behältern dar. Sie können z.B. verwendet werden, um die Wörter eines Textes zu speichern. Sie werden in eckigen Klammern ([...]) angegeben, mit Kommas als **Trenner** (engl. *separator*) zwischen den Elementen, z.B.

```
woerter = ['dies', 'ist', 'ein', 'Satz']
```

oder

```
nummern = [2, 5, 8]
```

Zu beachten ist bei Listen von Zeichenketten auch, dass jedes Element wie üblich durch Anführungszeichen markiert sein muss.

Listen können ebenfalls mithilfe der `list()`-Funktion erstellt werden, mit oder ohne Initialisierung, z.B.

```
leereListe = list()
```

oder

```
woerter = list(['dies', 'ist', 'ein', 'Satz'])
```

Leere Listen werden meist dann erstellt, wenn man noch nicht weiß, welche Elemente genau darin gespeichert werden sollen, z.B. wenn Texte erst später mithilfe anderer Funktionen in Wörter aufgespalten werden, oder man eine Liste wieder löschen will, auch wenn es dazu eigentlich eine separate Methode gibt (siehe unten). Auf einzelne Elemente kann über `Listenname[Index]` zugegriffen werden, z.B. mit `woerter [1]` auf das zweite (!) Wort unserer oben definierten Wortliste.

Nützliche Listenmethoden sind in Tabelle 9 aufgelistet.

Methode	Funktionalität
<code>append()</code>	fügt ein Objekt am Ende der Liste an, erweitert also die Liste durch ein einzelnes Element, was jedoch wiederum eine Liste/Sequenz sein kann
<code>extend()</code>	fügt Elemente eines anderen listenartigen Objekts am Ende an, erweitert also die Liste um ein oder mehrere Elemente
<code>sort()</code>	sortiert – und verändert – die Liste nach bestimmten Kriterien (mehr dazu später)

<code>len()</code>	liefert die Länge der Liste zurück, also die Anzahl der Elemente
<code>clear()</code>	leert die Liste

Tabelle 9: Nützliche Listenmethoden

Der Unterschied zwischen der `append()`- und der `extend()`-Methode ist sehr wichtig: Bei `extend()` werden immer einzelne Elemente, die aus einfachen Datentypen bestehen, an die ursprüngliche Liste angehängt. Bei `append()` können auch komplexe Datentypen angehängt werden, deren (Unter-)Elemente aber nicht direkt in die Liste aufgenommen werden. Wenn aber Objekte mit mehreren Elementen an eine Liste angehängt werden, so müssen diese Listen jeweils später einzeln abgearbeitet werden, um tatsächlich auch auf alle einzelnen Elemente zuzugreifen, da es sonst leicht zu Fehlern kommen kann.

?

Übung 13 - Einfache Wortliste

Erstellen Sie ein neues Programm namens `02_woerterliste.py`.

Deklarieren und initialisieren Sie zunächst eine Liste von Wörtern, so wie oben im Text gezeigt.

Verwenden Sie dann eine `print()`-Anweisung, in der Sie als Argumente die einzelnen Elemente ausgeben, wobei Sie mit dem Element an der letzten Position einen Punkt als Zeichenkette konkatenieren, wie in der letzten Sitzung unter Zeichenkettenoperatoren besprochen.

Fügen Sie eine weitere `print()`-Anweisung hinzu, in der Sie die ersten beiden Argumente vertauschen und statt dem Punkt ein Fragezeichen mit dem letzten Element konkatenieren.

Was haben Sie jetzt im letzten Schritt jetzt fast vollständig simuliert?

3.2 Einfache Interaktion mit Programmen und Benutzern

Zusatzfunktionen in/aus externen Modulen können in Python über die `import`-Anweisungen eingebunden werden. Eine solche Funktionalität, um mit der Kommandozeile zu interagieren, bietet das `sys`-Modul, welches mit der Anweisung `import sys` in ein Programm eingebunden wird. Sobald das Modul zur Verfügung steht, kann man auf die `sys.argv`-Liste zugreifen, welche die Programmargumente beinhaltet, wobei `sys.argv[0]` immer den Programmnamen selbst enthält, also meist erst die Elemente ab Position 1 für uns interessant werden.

Die Nutzung von `sys.argv` ist dann sinnvoll, wenn Benutzer mit dem Programm relativ gut vertraut sind. Falls die Benutzer jedoch das Programm nicht (gut) kennen,

ist die Verwendung von `sys.argv` ohne geeignete Fehlerbehandlung, worüber wir in Abschnitt 5.1.3 mehr lernen werden, meist nicht sehr sinnvoll.

Eine Alternative zu Eingaben über die Kommandozeile ist es, Argumente vom Benutzer mithilfe der `input()`-Funktion einzuholen und in Variablen zu speichern, die dann verarbeitet werden können. Idealerweise sollte diese Funktion mit passenden Instruktionen über das optionale `prompt`-Argument, z.B.

```
studienfach = input('Was studieren Sie?\n')
```

verwendet werden. Die Zeichensequenz `\n` am Ende des Prompts dient dazu, dass vor der Zeile, auf der die Eingabe erfolgen soll, erst einmal ein Zeilenumbruch eingefügt wird, da sonst die Benutzereingabe direkt hinter dem Prompt erfolgen müsste, was nicht ideal aussieht. Die `input()`-Funktion eignet sich z.B. zum Erstellen einfacher Tests oder Abfragen anderer Informationen.

Bei beiden Methoden sollten die Programmargumente auf Gültigkeit geprüft werden, zumindest wenn falsche oder zu wenige Eingaben möglich sind. Wie, werden wir in Abschnitt 3.4 lernen, wenn wir Bedingungsabfragen behandeln.



Übung 14 - Argumente von der Kommandozeile einholen

Schreiben Sie zwei Programme, `03_get_args_argv.py` und `04_get_args_input.py`.

Im ersten sollen zwei Wörter als Programmargumente übergeben werden, und im zweiten mithilfe von zwei `input()`-Anweisungen ebenfalls zwei Wörter vom Programmbenutzer abgefragt und in entsprechenden Variablen gespeichert werden. In beiden Programmen sollen danach die zwei Wörter mit passenden Begleittexten zur Erklärung wieder ausgegeben werden, wobei beim ersten die beiden Wörter auf der gleichen Zeile ausgegeben werden sollen, und beim zweiten der Begleittext und die Wortausgabe auf der nächsten Zeile, und auch nach der zweiten Eingabe noch ein Zeilenumbruch erfolgen sollte.

Beim ersten Programm soll weiterhin als erste Ausgabe der Programmname mit passendem Begleittext ausgegeben werden.

Bei der letzten Ausgabe sollten Sie immer eine geeignete Kombination von Argumenten und Konkatenation innerhalb einer einzigen `print()`-Anweisung verwenden.

3.3 Lösungsstrategien und Schadensbegrenzung

Komplexere Programme erfordern gezielte und manchmal sehr komplexe Lösungsstrategien, die anfangs oft umständlich erscheinen. Dabei muss man häufig ein Programm in logische, einfacher zu handhabende, Teilschritte aufteilen, um die Ergebnisse später

wieder zu einem Ganzen zusammenzufügen. Auf Englisch nennt man diese Strategie **divide-and-conquer**.

Programmieren zu lernen, erfordert die gezielte Entwicklung und Schulung algorithmischen Denkens durch Ausprobieren und die Fähigkeit, aus eigenen Fehlern zu lernen. Deshalb sollten Sie keine Angst davor haben, manche Dinge einfach auszutesten, vielleicht, wenn sich dies anbietet, zunächst auch nur in der Python-Shell.

Programmfehler können jedoch zu Endlosschleifen führen, bei denen das Programm dann nicht mehr einfach zu stoppen ist. In solchen Fällen bleibt oft als letzte Lösung nur die Option, `Strg + c` zu drücken, wodurch das Programm auf Betriebssystemebene abgebrochen wird. Wir werden bald aber schrittweise elegantere Möglichkeiten besprechen, mit vorhersagbaren Fehlern umzugehen. Kritisch kann es allerdings beim Austesten werden, wenn Sie einmal anfangen, in Dateien zu schreiben, was wir aber in Abschnitt 5.1.3 noch im Detail behandeln werden.

Die etwas fortgeschritteneren Übungen ab diesem Punkt sind so angelegt, dass sie, neben Angabe des Programmnamens, immer mit einer kurzen Beschreibung dessen beginnen, was mit ihnen erzielt werden soll. Folgen Sie deshalb nicht einfach nur meinen schrittweisen Anweisungen. Überlegen Sie vielmehr, bevor Sie mit dem Schreiben des Programms beginnen, ob Sie sich schon vorstellen können, wie dies mit den Mitteln, die wir bisher kennengelernt haben, zu erreichen ist. Bei diesen Mitteln kann es sich zum Teil um etwas handeln, was gerade eben erst eingeführt wurde, später aber immer mehr um eine Kombination von Neuem und Methodiken, die wir schon vorher besprochen haben. Da die Übungen teilweise aufeinander aufbauen, bzw. schon bestehende Ansätze darin manchmal ausgebaut werden, kann es sehr gut sein, dass Sie unter Umständen ein schon bestehendes Programm als Vorlage verwenden können und nur in geeigneter Weise umschreiben müssen.

Zudem sollten Sie ebenfalls erlernen, auf bestimmte Schlüsselwörter oder Begriffe mehr oder weniger automatisch zu reagieren, da ich diese teilweise schon in den Anweisungen ‚versteckt‘ habe, um Sie zum Nachdenken anzuregen, oder sogar explizit angebe. Auf diese Weise werden Sie die Übungen hoffentlich nicht nur rein mechanisch, Schritt für Schritt, durchgehen, sondern Ihre Programmierfähigkeiten auch durch Eigeninitiative immer weiter ausbauen. Nur dadurch ist das Erlernen von Programmiertechniken wirklich möglich.

3.4 Bedingungsabfragen

Bedingungsabfragen sind ein wichtiger Bestandteil der Programmierung. Normalerweise muss an mehreren Stellen in jedem Programm überprüft werden, ob bestimmte Konditionen gegeben sind und, basierend auf dem Ergebnis, unterschiedliche Schritte, wie z.B. Verzweigungen oder Programmabbrüche, eingeleitet werden müssen, oder Entscheidungen darüber getroffen, was mit dem Inhalt einer Variablen geschehen soll.

Solche Abfragen erfolgen in Form von Anweisungen, wo das Schlüsselwort `if` gefolgt ist von ein oder mehreren Bedingungen und einem Doppelpunkt, z.B.

```
if wort1 < wort2:
```

Bedingt auszuführende Anweisungen müssen als **Block** eine Stufe, also eine feste Anzahl an Leerzeichen, eingerückt sein, was in IDEs meist durch das Drücken der Tabulatortaste (⇥) erreicht werden kann. Dabei ist, wie schon angesprochen, Python äußerst penibel, was die Einrückungstiefe angeht, da Blöcke hier nicht, wie in vielen anderen Programmiersprachen, durch Klammerung gekennzeichnet sind, sondern eben nur die Einrückung selbst. Gute IDEs, wie die WingIDE, machen jedoch die Einrückung nach Schlüsselwörtern wie `if` und sobald auch ein Doppelpunkt eingetippt wurde, automatisch. Wenn ein Editor oder eine IDE, die Sie verwenden, dies nicht erlauben sollte, verwenden Sie am besten vier Leerzeichen, da dies die offizielle Empfehlung darstellt.

Falls mehrere alternative Bedingungen abgefragt und angeführt werden müssen, werden diese jeweils mit

```
elif Bedingung(en):
```

eingeleitet, und um alle nicht speziell definierten Bedingungen abzudecken, kann zuletzt

```
else:
```

verwendet werden, wobei nach jedem dieser Schlüsselwörter wieder ein entsprechender Anweisungsblock eingefügt wird. Alternative Bedingungen müssen dabei jeweils auf derselben Einrückungsstufe wie das `if` stehen.

Die allgemeine Form der Syntax von Bedingungsabfragen sieht also folgendermaßen aus.

```
if Bedingung(en):  
    Anweisung(en)  
elif Bedingung(en):  
    Anweisung(en)  
else:  
    Anweisung(en)
```

Nachdem der letzte Bedingungsblock abgearbeitet ist, wird zum normalen Programmablauf übergeleitet, wobei der weitere Code wieder eine Stufe weniger eingerückt sein sollte. Falsche oder uneinheitliche Einrückung führt in Python meist zu Syntaxfehlern, die am Anfang bei den meisten unerfahrenen Programmierern und Programmierinnen sehr häufig auftreten werden. Auch könnte es vorkommen, dass Sie Code von

jemand anderem erhalten, der mit anderen Einrückungen formatiert ist, und den Sie in Ihre Programme übernehmen wollen. In diesem Fall müssten Sie dann die Einrückungen unter Umständen manuell korrigieren.

Manchmal kann es auch vorkommen, dass man bestimmte Bedingungsalternativen zwar schon voraussieht, diese aber noch nicht direkt implementieren kann oder will. In diesem Fall müssen diese Programmteile durch das Schlüsselwort `pass` gekennzeichnet werden, damit sie beim Testen des Programms übersprungen werden, da sonst ein Syntaxfehler auftritt.

Bei komplexeren Programmen ist es oft der Fall, dass verschachtelte Bedingungsblöcke, also Bedingungszeige innerhalb von Bedingungen, auftreten. Dabei muss man ganz besonders darauf achten, auch auf der richtigen Einrückungsstufe zu sein, da sonst zwar keine Syntaxfehler auftreten, aber logische, die die IDE auch bei der besten Syntaxprüfung nicht erkennen kann!

Übung 15 - Bedingungsabfragen zum Wortvergleich

Schreiben Sie ein neues Programm (`05_wortvergleich.py`), in dem Sie über die Kommandozeile zwei Wörter als Argumente angeben können.

In diesem Programm sollte mithilfe einer Bedingungsabfrage zunächst festgestellt werden, ob beide Wörter gleich sind, und falls ja, eine entsprechende Nachricht ausgegeben werden, wobei immer beide Wörter mit ausgegeben werden sollten.

Danach soll getestet werden, ob das erste Wort gemäß Computersortierung vor dem zweiten kommt, wieder mit entsprechender Meldung.

Zum Schluss sollte dann als letzte Alternative nur eine Meldung ausgegeben werden, dass das zweite Wort nach dem ersten kommt.

Achten Sie dabei insbesondere auf die konsistente Einrückung der Blöcke.



3.5 Schleifen

Schleifen werden benötigt, um bestimmte Programmteile wiederholt auszuführen, z.B. um jedes Wort eines in einer Liste gespeicherten Satzes auf bestimmte Eigenschaften zu überprüfen. Das sequenzielle Abarbeiten von Elementen nennt man **iterieren**, so dass man also über Listen iteriert, um Ihre Elemente abzuarbeiten. Diese Art von Iteration erreicht man in Python über `for`-Schleifen.

Zunächst wollen wir allerdings noch eine andere Art von Schleifen besprechen, nämlich solche, die so lange – oder während – bestimmte Bedingungen erfüllt sind ablaufen, weshalb sie `while`-Schleifen heißen. Weiterhin gibt es auch noch eine Möglichkeit, die Eigenschaften beider Schleifentypen zu kombinieren, wobei man dann von **list comprehension**, also Listenverständnis, spricht. Diese Form ist oft effizienter und eleganter als `for`-Schleifen zur Filterung oder Manipulation von Daten, aber auch etwas komplexer, weshalb wir sie erst in Abschnitt 10.5 besprechen.

Ähnlich wie bei Bedingungen, werden auch die Anweisungen innerhalb dieser Schleifen in einer Blockstruktur geschrieben. Die Schleifenkontrolle erfolgt dabei hauptsächlich im Schleifenkopf, also der Zeile, in der die Schleife definiert ist. Allerdings gibt es auch noch weitere Kontrollmöglichkeiten, und falls innerhalb einer Schleife eine Bedingung erfüllt ist, kann diese z.B. vorzeitig durch `break` komplett abgebrochen oder mithilfe von `continue` der nächste Schleifendurchgang erzwungen werden.

3.5.1 while-Schleifen

Wie schon oben angesprochen, laufen `while`-Schleifen, solange ein oder mehrere Bedingungen erfüllt sind. Die allgemeine Syntaxform sieht folgendermaßen aus.

```
while Bedingung(en):
    Anweisung(en)
```

Diese Schleifen können z.B. verwendet werden, um wiederholt Eingaben von Benutzern anzufordern oder bestimmte Textabschnitte abzuarbeiten/zu überspringen.



Übung 16 - While-Schleifen zur Eingabe nutzen

Schreiben Sie das Programm `06_satz_eingeben.py`, in dem Sie mithilfe der `input()`-Funktion vom Benutzer Wörter abfragen, bis entweder ein `.` oder `?` eingegeben wird und Sie dann den Satz noch einmal komplett ausgegeben.

Fragen Sie dazu zunächst ein anfängliches Wort vom Benutzer ab und speichern Sie es in einer geeigneten Variable.

Kopieren Sie dann den Inhalt der Variable in eine, die den Anfang des Satzes bildet. Starten Sie daraufhin eine `while`-Schleife, die überprüft, ob das eingegebene ‚Wort‘ nicht ein Punkt oder Fragezeichen ist.

Fordern Sie jetzt innerhalb des Schleifenblocks den Benutzer wiederholt dazu auf, Wörter oder Satzzeichen einzugeben, um diese wiederum in der Wort-Variable zu speichern.

Fügen Sie danach einen Bedingungsblock hinzu, durch den jeweils ein Leerzeichen und der Inhalt des letzten Wortes an die bestehende Satzvariable angehängt wird, aber nur, falls das eingegebene ‚Wort‘ nicht ein Punkt oder Fragezeichen ist.

Geben Sie zuletzt nach Ende des Schleifenblocks eine Nachricht aus, in der der Benutzer noch einmal den eingegebenen Satz und auch das zuletzt eingegebene Satzzeichen sehen kann.

3.5.2 for-Schleifen

Wie schon erwähnt, ermöglichen `for`-Schleifen es, mithilfe einer Variablen nacheinander auf alle Elemente von Sequenzen zuzugreifen, wobei außerdem der `in`-Operator verwendet wird, so dass die allgemeine Syntax wie folgt aussieht.

```
for Variable in Folge:
    Anweisung(en)
```

Übung 17 - Über einen Satz iterieren

Schreiben Sie das Programm `07_wort_verkleinerung.py`, in dem ein Satz in einzelne Wörter zerlegt und diese dann alle klein geschrieben ausgegeben werden sollen.

Lassen Sie darin zunächst den Benutzer einen Satz ohne Satzzeichen eingeben. Verwenden Sie dann die `split()`-Methode des `str`-Objekts (wie in 2.3 beschrieben) ohne Argument, um die Wörter des Satzes einer Wortliste zuzuweisen.

Schreiben Sie nun eine `for`-Schleife über alle Wörter in der Wortliste, in der jedes Wort unter Zuhilfenahme der `str`-Methode `lower()` ausgegeben wird.

Geben Sie zuletzt zur Information noch einmal den ursprünglichen Satz mit einer passenden Nachricht aus.

In diesem Kapitel haben wir zusammengesetzter Datentypen kennengelernt sowie Methodiken, wie man über eine bestimmte Art dieser Datentypen, nämlich Listen, iterieren kann. Zudem haben wir besprochen, wie man durch das Abfragen von Bedingungen den Programmfluss steuern und mit Benutzern über die Kommandozeile interagieren kann, um simple Eingaben in unsere Programme zu ermöglichen. Im nächsten Kapitel gehen wir etwas genauer auf den für die Verarbeitung von Sprache essenziellen Datentyp der Zeichenkette ein.

3.6 Lösungen zu den Aufgaben

Lösung 13 - Einfache Wortliste

Der erste Schritt sollte relativ einfach sein, da die zwei Formen, mit denen man die Wortliste deklarieren und instanziiieren kann, oben im Text stehen. Die erste eigentliche Programmzeile, nach dem Kopf mit Shebang-Zeile und dokumentierenden Kommentaren, sollte also entweder

```
woerter = ['dies', 'ist', 'ein', 'Satz']
```



oder

```
woerter = list(['dies', 'ist', 'ein', 'Satz'])
```

sein, wobei die erste Form kürzer ist. Wichtig ist bei beiden Formen, dass die Listenelemente bei der Initialisierung immer in eckigen Klammern angegeben werden müssen und dass alle Zeichenketten auch durch Anführungszeichen als solche markiert sind. Wenn Sie versuchen, das Programm auf der Kommandozeile auszuführen, und z.B. die schließende eckige Klammer vergessen haben, dann wird Ihnen ein Syntaxfehler für die Zeile nach der versuchten Deklaration der Liste angezeigt. Allerdings wird Ihnen, bevor Sie das Programm so speichern, auch schon die WingIDE Syntaxfehler für die Folgezeilen anzeigen, was Sie schon beim Schreiben des Programms auf den Fehler hinweist. Sollten Sie hingegen eines der Anführungszeichen vergessen, dann werden nicht nur die Folgezeilen als fehlerhaft markiert, sondern auch die schließende Klammer, so dass Sie dadurch darauf schließen können, dass innerhalb der Klammer etwas nicht stimmt. Auf diese Weise können – und sollten – Sie sich die Syntaxhilfe immer zunutze machen, wobei immer zu beachten ist, dass die Fehler unter Umständen erst für eine Stelle gemeldet werden, wo die Syntaxprüfung ‚bemerkt‘, dass etwas nicht stimmt.

Um auf die einzelnen Elemente der Liste hintereinander zugreifen und diese als Argumente für die `print()`-Funktion verwenden zu können, müssen Sie jeweils, wie oben besprochen, die Indexpositionen verwenden. Dabei sollten Sie nicht vergessen, dass das erste Element auf Position 0 steht, so dass die Ausgabe – zunächst ohne die Konkatenation des Satzzeichens – so aussehen sollte:

```
print(woerter[0], woerter[1], woerter[2], woerter[3])
```

Falls Sie fälschlicherweise bei Position 1 anfangen zu zählen und die Indexpositionen von 1–4 verwenden, wird Ihnen Python den folgenden Fehler melden:

```
builtins.IndexError: list index out of range
```

Dieser tritt jedes Mal auf, wenn Sie bei einer Liste versuchen, auf eine nicht existierende Indexposition zuzugreifen.

Die Leerzeichen bei der Ausgabe werden, wie Sie sich hoffentlich erinnern werden, automatisch durch die `print()`-Funktion zwischen den Argumenten eingefügt, was uns etwas Arbeit erspart. Allerdings können wir deshalb nicht auch einfach das Satzzeichen als letztes Argument mit einfügen, da sonst zwischen ihm und dem vorausgehenden Wort ein Leerzeichen ausgegeben würde, was aber nicht den Konventionen der Rechtschreibung entspricht. Stattdessen müssen wir das letzte Argument so verändern, dass das Satzzeichen dort mit angehängt, also konkateniert wird, weshalb die endgültige Form der ersten `print()`-Anweisung dann wie folgt aussieht:

```
print(woerter[0], woerter[1], woerter[2], woerter[3]+ '.')
```

Im letzten Schritt sehen Sie, wie einfach es ist, eine fast perfekte syntaktische Inversion eines einfachen Deklarativsatzes zu simulieren. Was wir in Übung 7 nur anhand der ersten zwei Wörter eines ähnlichen Satzes gezeigt hatten, und mit zwei einzelnen Variablen, können wir hier noch etwas deutlicher simulieren, indem wir einfach die ersten zwei Elemente einer einzigen Listenvariable bei der Ausgabe vertauschen, so dass die endgültige Form der zweiten Ausgabe so aussehen sollte:

```
print(woerter[1], woerter[0], woerter[2], woerter[3]+ '?')
```

Allerdings hat unser Programm dennoch einen Schönheitsfehler, nämlich, dass der Buchstabe am Satzanfang noch nicht groß geschrieben ist. Wie wir das erreichen, werden wir im nächsten Kapitel lernen. Außerdem ist die Eingabe einzelner Wörter, um einen Satz zu simulieren, den wir normalerweise als eine Einheit sehen, natürlich immer noch sehr umständlich und dient hier rein zur Illustration. Wir werden sehr bald Wege sehen, wie man solche Sätze einlesen, in geeignete Listen aufspalten, und dann weiterverarbeiten kann.

Lösung 14 - Argumente von der Kommandozeile einholen

Aus sprachwissenschaftlicher Sicht ergeben die zwei Programme in dieser Form nicht sehr viel Sinn, da das Ein- und Ausgeben zufällig gewählter Wörter ja nicht sehr bedeutungsvoll ist. Allerdings können solche Eingaben im Zusammenhang mit Speicherung der eingegebenen Wörter durchaus als Grundlage von Tests dienen oder, aus rein programmiertechnischer Sicht tatsächlich genutzt werden, um die Interaktion mit Benutzern zu erreichen.

Ein wichtiger Unterschied bei den zwei Programmen ist, dass beim Einholen der Argumente per `sys`-Modul tatsächlich erst das Modul importiert werden muss, was bei der `input()`-Funktion nicht nötig ist. Beim ersten Programm muss also zwangsweise die erste eigentliche Programmanweisung die Anweisung

```
import sys
```

sein, da ansonsten der Fehler

```
builtins.NameError: name 'sys' is not defined
```

auftritt. Die erste `print()`-Anweisung im ersten Programm sollte dann auch in etwa so aussehen:

```
print('Der Programmname ist', sys.argv[0])
```

Bei der Zuweisung an die zwei Variablen und deren Ausgabe ist zu beachten, dass das erste Argument jetzt tatsächlich an Position 1 in der Argumentenliste `sys.argv` steht und das zweite auf 2, da ja die Position 0 den Programmnamen enthält, weshalb das erste Wort z.B. mit

```
wort1 = sys.argv[1]
```

zugewiesen werden kann und das zweite analog dazu. Die Ausgabe für dieses Programm sollte, bei idealer Kombination von Argumenten und Konkatenation, dann wie folgt aussehen:

```
print('Wort1:', wort1 + ' ; Wort2:', wort2).
```

Beim zweiten Programm sollte die Verwendung der `input()`-Funktion für das erste Wort in etwa so aussehen:

```
wort1 = input('Bitte 1. Wort eingeben...\n')
```

Beim zweiten Wort sollte dies wieder analog geschehen. Zur Ausgabe könnten Sie diese Anweisung schreiben:

```
print('\nWort1:', wort1 + '\nWort2:', wort2)
```

Dabei bewirkt `\n` am Anfang des ersten Arguments wieder einen Zeilenumbruch vor dem eigentlichen Text der Ausgabe.

Lösung 15 - Bedingungsabfragen zum Wortvergleich

Bei dieser Übung kombinieren wir die Verwendung von Kommandozeilenargumenten mit Bedingungsabfragen, die Vergleiche verwenden. Auch wenn dieses Beispiel etwas naiv erscheinen mag, so sind diese Vergleiche ein wichtiger Bestandteil von bestimmten Algorithmen, wie z.B. denen zur Sortierung von Wörtern. Für diese gibt es glücklicherweise schon vorgefertigte und sehr effiziente Funktionen/Methoden, so dass wir sie meist nicht selbst implementieren müssen. Manchmal jedoch sind wir auch dazu gezwungen, spezielle Sortier Routinen selbst zu erstellen, so dass ein grundsätzliches Verständnis dafür, wie dies funktioniert, sehr wichtig ist.

Da wir wieder die Argumente von der Kommandozeile einholen wollen, müssen Sie als erstes das `sys`-Modul importieren und dann die Argumente an den Indexpositionen 1 und 2 von `sys.argv` in zwei geeigneten Variablen speichern, was wir schon im vorletzten Programm geübt haben. Nur wollen wir diesmal nicht nur die Variablen wieder ausgeben, sondern tatsächlich die darin gespeicherten Wörter miteinander vergleichen und, basierend darauf, geeignete Meldungen ausgeben, die uns zudem erlauben, besser zu sehen und verstehen, wie die Sortierung auf dem Computer funktioniert.

Beim ersten Vergleich testen wir mittels `==` zunächst auf Gleichheit, wobei das Ergebnis dann innerhalb der `if`-Anweisung evaluiert wird. Besonders wichtig ist, dass beim Gleichheitsoperator zwei gleiche Symbole direkt hintereinander stehen, da ja bei der Verwendung eines einzelnen Zeichens kein Vergleich, sondern eine Zuweisung ausgeführt würde. Wichtig ist auch, dass der folgende Block mit einem Doppelpunkt eingeleitet wird und von einer Einrückung gefolgt ist. Sind die beiden Wörter tatsächlich gleich, dann liefert die Bedingungsabfrage den booleschen Wert `True` zurück und wir geben eine passende Meldung aus, in der wieder die beiden

Variablen in geeigneter Form eingebettet sein sollten, so dass der erste Teil unserer Bedingungsabfragen in etwa so aussehen sollte:

```
if wort1 == wort2:
    print("Die Wörter sind gleich.\nWort 1\t" + wort1 +
          '\nWort 2\t' + wort2)
```

Bei meiner Ausgabe habe ich zwischen den beiden Wörtern auch einen Tabulator (`\t`) eingefügt, um die Wörter noch deutlicher voneinander zu trennen.

Sind die Wörter nicht gleich, so ist das Ergebnis `False` und es wird die Anweisung im `if`-Block zunächst nicht ausgeführt, das heißt dieser Block einfach ignoriert. Da wir aber alternative Bedingungsabfragen definiert haben, werden diese ebenfalls der Reihe nach evaluiert, bevor wieder zum normalen Programmfluss übergeleitet wird. In einem längeren und komplexeren Programm würde dann tatsächlich noch mehr geschehen, aber in unserem Programm wird der Programmfluss hier schon beendet.

Da wir noch zwei weitere Alternativen haben, könnten wir theoretisch beide als `elif`-Abfragen formulieren. Bei beiden müssen Sie natürlich darauf achten, dass Sie diese auf derselben Einrückungstiefe wie die `if`-Anweisung verankern, also nach dem ersten Block wieder die Einrückung zurücknehmen, da sonst ein Syntaxfehler auftritt. Das Ausformulieren mit einem konkreten Vergleich ist im letzten Fall gar nicht mehr nötig, da wir ja schon alle anderen Möglichkeiten abgedeckt haben. Deshalb können wir uns Schreiarbeit ersparen und einfach die letzte Alternative mit `else` einleiten, weil diese alles andere automatisch abdeckt.

Beim `elif`-Teil muss die Bedingung überprüfen, ob das erste Wort kleiner ist (`<`) als das zweite, da die Zahlencodes der einzelnen Buchstaben der beiden Wörter jeweils von links nach rechts miteinander verglichen werden, und Großbuchstaben, oder Buchstaben die allgemein auf einer niedrigeren Position liegen, zuerst im Zeichensatz auftreten. Dies ist sinnvoll, da auch die Buchstaben im Alphabet eine bestimmte Reihenfolge haben. Dabei ist die Besonderheit am Computer, dass zwischen Großbuchstaben und Kleinbuchstaben per Code unterschieden wird, so dass z.B. das groß geschriebene Wort *Das* vor dem klein geschriebenen *das* kommt, auch wenn diese für uns als Leser gleichwertig sind und sich nur durch ihre Positionen im Satz unterscheiden, nicht in ihrer Semantik. Die `elif`-Bedingung sollte also folgendermaßen formuliert sein:

```
elif wort1 < wort2:
```

Lösung 16 - While-Schleifen zur Eingabe nutzen

Bei dieser Übung müssen wir zweimal die `input()`-Funktion anwenden, einmal, um das erste Wort abzufragen und dann ein weiteres Mal innerhalb der Schleife. Beim ersten Mal bietet es sich an, die Benutzer auch darüber zu informieren, was im Programm überhaupt geschehen soll und dass die Eingabe des Satzes durch entweder einen Punkt oder ein Fragezeichen abgeschlossen werden soll. Dies kann in etwa so aussehen:

```
wort = input(
    'Bitte geben Sie Wörter ein, um einen Satz zu bilden.\n'
    'Zum Abschluss des Satzes geben Sie einfach . oder ? ein.\n')
```

Wenn das erste Wort eingegeben ist, dann bildet dies den Anfang des Satzes, und wir können eine Variable `satz` mit diesem Wort initialisieren, an die wir später innerhalb der `while`-Schleife alle Wörter anhängen, es sei denn, dass ein Satzzeichen vorliegt. Wir bauen also den Satz schrittweise immer weiter auf, indem wir Wörter anhängen, solange kein Satzzeichen auftritt. Zur Kontrolle der `while`-Schleife müssen wir also eine Bedingung verwenden, die besagt, dass das eingegebene Wort nicht ein Punkt oder ein Fragezeichen sein darf. Hierbei unterscheidet sich aber die Programmierlogik von der natürlichsprachlichen Logik, da das *oder*, weil beide Teilbedingen gleichzeitig erfüllt sein müssen, als *and* ausgedrückt werden muss, also wir im Prinzip sagen würden „Wenn das Wort nicht ein Punkt ist und (auch) nicht ein Fragezeichen, dann führe die Anweisung im Block aus“. Deshalb muss die Schleifenkontrolle so definiert werden:

```
while wort != '.' and wort != '?':
```

Falls Sie versehentlich an dieser Stelle ein `or` verwenden würden, dann hätten Sie damit eine Endlosschleife produziert und müssten Ihr Programm, sobald Sie feststellen, dass es nicht nach Eingabe eines Satzzeichens stoppt, durch Drücken von `Strg + C` abbrechen!

Da die `while`-Schleife aber immer nur das zuletzt eingegebene Wort überprüfen kann, müssen wir im ersten Schritt innerhalb des Blocks erst einmal ein neues Wort erbitten. Dies soll dann, falls dieselben Bedingungen wie oben erfüllt sind, an die bestehende Satzvariable angehängt werden. Zum Anhängen an den bisherigen Satz müssen wir das neue Wort mit einem vorangehenden Leerzeichen an die Variable anhängen, so dass unsere komplette `while`-Schleife jetzt so aussieht.

```
while wort != '.' and wort != '?':
    wort = input('Weiteres Wort oder Satzzeichen?\n')
    if wort != '.' and wort != '?':
        satz += ' ' + wort
```

Wurde innerhalb der Schleife, also bei der zweiten `input()`-Anweisung, eines der Satzzeichen eingegeben, dann ergibt sich bei der darauffolgenden Bedingungsabfrage `False`, weshalb das Satzzeichen in der Wortvariable nicht an den bestehenden Satz angehängt wird und der Programmfluss wieder bei der `while`-Schleife landet. Da diese aber ebenfalls `False` liefert, wird dadurch der komplette `while`-Block beendet und der Programmfluss kehrt wieder auf die äußerste Ebene zurück, wo wir jetzt unsere abschließende Nachricht ausgeben können, die folgendermaßen aussehen sollte:

```
print('Der Satz war:', satz + wort)
```

Der Grund, weshalb wir die Bedingungsabfrage innerhalb der Schleife hatten, wird jetzt hoffentlich klarer. Hätten wir einfach alle ‚Wörter‘ innerhalb der Schleife gleichermaßen an den bisherigen Satz angehängt, dann wäre auch unser Satzzeichen nicht direkt an das letzte Wort des Satzes angehängt, sondern davor ein Leerzeichen eingefügt worden, was nicht den Konventionen der Rechtschreibung entspricht. Deshalb wurde es zwar noch in der Wortvariable ‚gespeichert‘ und ist auch nach dem Schleifendurchlauf noch dort vorhanden. Daher kann es jetzt bei der Ausgabe direkt mit dem bestehenden Satz konkateniert werden.

Selbstverständlich ist das schrittweise Eingeben eines Satzes in dieser Form nicht unbedingt sehr sinnvoll als Übung, es sei denn, man will die Kreativität von Sprechern überprüfen. Sinnvoller könnte zum Beispiel sein, Sprachlernende mithilfe einer Liste aus vorgegebenen Elementen bezüglich ihres Kenntnisstands zu Kollokationen zu testen.

Lösung 17 - Über einen Satz iterieren

Mit dieser Übung greifen wir auf das nächste Kapitel vor, da wir hier neben der sequenziellen Abarbeitung einer Liste auch schon einige wichtige Methoden der Zeichenkettenverarbeitung verwenden. Was wir jedoch ebenfalls kennenlernen, ist eine bessere Option, die Bestandteile von Sätzen in Listen zu speichern, als diese entweder vom Benutzer abzufragen, wie in der letzten Übung, oder direkt vorzudefinieren, indem wir jedes einzelne Wort bei der Initialisierung der Liste angeben. Dies bildet dann später auch eine der Grundlagen für die Verarbeitung von ganzen Texten.

Das Programm selbst zu schreiben, dürfte eigentlich nicht sehr schwer sein, da wir außer den neuen `str`-Methoden nur schon verwendete Konzepte anwenden. Bei der Verwendung der anfänglichen `input()`-Funktion sollte wieder eine geeignete Instruktion mit ausgegeben werden und die Eingabe in einer Variablen wie `satz` gespeichert werden. Daraufhin kann dann unsere `woerter`-Liste einfach mittels `satz.split()` aus der Zeichenkette generiert, bzw. initialisiert, werden, wobei wir uns erst einmal an die Form des Methodenaufrufs mittels `Variablenname.Methodenname()` gewöhnen müssen. Auch ist es hier wichtig, zu verstehen, dass beim Aufruf der `split()`-Methode ohne Argument automatisch (als Voreinstellung) an Leerzeichen getrennt wird.

In der `for`-Schleife müssen wir nur eine temporäre Variable verwenden, um jedes einzelne Element in der Liste abzuarbeiten. Dabei muss der (ent)sprechende `in`-Operator verwendet werden, so dass der Schleifenkopf wie folgt aussieht.

```
for wort in woerter:
```

Bei jedem Durchlauf der Schleife wird jetzt automatisch das jeweils nächste Element der `woerter`-Liste in der temporären Variable `wort` abgelegt. Wir können dann im Schleifenblock ganz einfach auf deren Inhalt zugreifen und diesen mit der `print()`-Anweisung zeilenweise ausgeben, nachdem wir alle Buchstaben der darin befindlichen Zeichenkette mittels der `lower()`-Methode zu Kleinbuchstaben umgewandelt haben.

Die Ausgabe des ursprünglichen Satzes ist nicht unbedingt nötig, da er eigentlich noch auf der Kommandozeile oberhalb der einzeln ausgegebenen Wörter steht. Sie könnte aber nützlich sein, falls er so lange ist, dass er bei der zeilenweisen Anzeige der einzelnen Wörter mittlerweile schon vom Bildschirm verschwunden ist.

4 Grundlagen der Zeichenkettenverarbeitung



Dieses Kapitel bietet primär eine ausführliche Übersicht über den wichtigsten Datentyp in der Sprachanalyse und -verarbeitung, die Zeichenkette. Wir werden zuerst besprechen, wie man Zeichenketten bereinigen kann. Danach werden einige grundlegende Operationen zur Bearbeitung von Zeichen- und anderen Sequenzen vorgestellt und aufgezeigt, wie man Zeichenketten aus anderen extrahieren oder zu längeren zusammenfügen sowie Groß- und Kleinschreibung handhaben kann.

4.1 Zeichenketten

Zeichenketten stellen jeweils eine Liste aus 1-Element-Zeichenketten dar, die man z.B. mithilfe der `list()`-Funktion in ihre einzelnen Elemente aufspalten kann. Ebenso sind sie iterierbar, das heißt man kann eine `for`-Schleife über die einzelnen Elemente laufen lassen, wie wir dies mit den Wörtern in unserer letzten Übung getan haben. Um nur auf bestimmte Teile einer Zeichenkette zuzugreifen oder diese zu extrahieren, kann man mit sogenannten *Slices* arbeiten, die wir in Abschnitt 4.3.1 kennenlernen werden. Ansonsten kann man auch alle anderen allgemeinen Listenoperationen, wie z.B. Sortierung, Umkehrung etc., auf Zeichenketten anwenden.

Da Zeichenketten Objekte sind, kann man verschiedenen Aktionen mit ihnen durchführen, die wir zum Teil bereits kennengelernt haben.

Funktionalität	Methoden
bestimmte Eigenschaften abfragen	<code>islower()</code> , <code>isupper()</code> , <code>isdigit()</code> , <code>startswith()</code> , <code>endswith()</code> etc.
Zeichenketten durchsuchen	<code>find()</code> , <code>rfind()</code> etc.
Zeichenketten manipulieren	<code>upper()</code> , <code>lower()</code> , <code>split()</code> etc.

Tabelle 10: Nützliche Zeichenkettenmethoden

Mit Ihren Englischkenntnissen können Sie die Funktionalität vieler dieser Methoden schon in etwa identifizieren, aber wir werden sie natürlich im Folgenden, soweit benötigt, noch genauer besprechen.

Zeichenketten können auch Sonderzeichen enthalten, es sei denn, sie sind als roh markiert (`r' ... '`). So z.B. hatten wir gesehen, dass `\n` einen Zeilenumbruch und `\t` ein Tabulatorzeichen (also im Prinzip mehrere Leerzeichen, allerdings als ein einzelnes

Zeichen kodiert) bei der Ausgabe produzieren. Dabei ist der Zeilenumbruch jedoch für verschiedene Betriebssysteme intern unterschiedlich repräsentiert, unter Windows als `\r\n` (eine Kombination aus Zeilenvorschub und neuer Zeile) unter Linux oder MacOS als `\n` (also nur Zeilenvorschub) und auf älteren MacOS-Versionen, die allerdings nicht mehr sehr verbreitet sind, als `\n\r`. Praktisch gesehen stellt dies für uns in Python bei der Verarbeitung von reinen Textdateien normalerweise kein Problem dar, da beim Öffnen automatisch immer in `\n` umgewandelt wird und Python beim Schreiben auch ebenso automatisch wieder die für das Betriebssystem passende Konvention verwendet. Der einzige Fall, bei dem man vielleicht manuell Ersetzungen durchführen muss, ist, wenn man Programme unter Betriebssystemen laufen lassen will, die eine andere Konvention verwenden. Das sind z.B. unter Windows geschriebene Programme, die man unter Linux ausführen will, da bei letzterem die Shebang-Zeile u.U. nicht richtig gelesen wird, wenn Sie die Kombination `\r\n` enthält. Dieses Problem lässt sich allerdings relativ einfach lösen, indem man ein simples Konvertierungsprogramm schreibt, welches die Datei(en) in ein passendes Format umkopiert.



Übung 18 - Normale und rohe Zeichenketten unterscheiden

Aktivieren Sie den Reiter für die Python-Shell in der WingIDE.

Geben Sie zunächst die Anweisung `word='\nHallo'` ein und geben Sie dann das Ergebnis mithilfe einer `print()`-Anweisung aus.

Initialisieren Sie die Variable mit `r'\nHallo'` neu und geben Sie sie nochmals aus.

Für den Umgang mit einfachem Text benötigen wir meist nur normale Zeichenkettenvariablen. Allerdings werden wir in Kapitel 6 sehen, wozu genau die Verwendung des Rohformats nützlich sein kann.

4.2 Zeichenketten bereinigen

Manchmal enthalten Zeichenketten, die wir verarbeiten wollen, zusätzliche und/oder redundante Informationen, z.B. unnötige Leerzeichen oder Zeilenumbrüche. Eine einfache Bereinigung solcher Zeichenketten können wir mittels drei verschiedener Verfahren erreichen. Dabei ist zu beachten, dass durch die Anwendung der unten besprochenen Methoden auf die Zeichenkette diese eigentlich nicht verändert wird, sondern dass, wie bei der Übergabe als Argument an eine Funktion, nur der Inhalt der Variable verwendet wird. Um die Ketten tatsächlich zu verändern, muss das Ergebnis der Methode wieder einer Variable zugewiesen werden, wobei bei Verwendung desselben Variablenamens auf der linken Seite diese neu angelegt und initialisiert, also quasi überschrieben, wird. Allerdings können wir auch das Ergebnis einer vollständig neuen Variable zuweisen, was unter Umständen zu einer unnötigen Vervielfältigung von Variablenamen führt. Hierbei ist egal, ob Sie die Methoden auf eine Zeichenkette

selbst anwenden oder eine Variable, die eine Zeichenkette beinhaltet. Im Folgenden werde ich Ihnen meist kurze Beispiele geben, bei denen ich die Methoden direkt auf Zeichenketten anwende, in der darauffolgenden Übung werden Sie diese dann selbst mit Variablen verwenden.

Das erste Verfahren ist die eingebaute `strip()`-Methode, die jeweils am Anfang und Ende der Kette bestimmte Zeichen entfernt, die als Argument übergeben werden. Ohne Argument ist hier die Standardeinstellung, dass alles entfernt wird, was als Leerzeichen gilt – also eigentliche Leerzeichen, Tabulatoren, oder sogar Zeilenumbrüche. So liefert z.B. die Anweisung `word = 'Wort '.strip()` die Zeichenkette `word` ohne die zwei Leerzeichen am Ende zurück. Mit einem einzelnen Zeichen als Argument werden alle Vorkommen dieses Zeichens am Anfang und/oder Ende der Kette entfernt, sooft diese auftreten. Zum Beispiel liefert

```
'ein See'.strip('e')
```

als Ergebnis in `s` zurück, da am Anfang ein `e` und am Ende zwei gelöscht werden. Mit mehreren Zeichen, die als Argument in einer Zeichenkette stehen, wird diese Zeichenkette als Liste von Zeichen interpretiert und es werden alle diese Zeichen gelöscht, bis keines davon mehr am Anfang oder Ende auftritt. Zu beachten ist dabei, dass die Reihenfolge der Zeichen in dieser Kette irrelevant ist. Deshalb ist diese Methode nicht unbedingt zum Entfernen von Prä- und Suffixen geeignet, da dieselben Zeichen natürlich noch an einer anderen Stelle – Entweder am Anfang oder Ende – auftreten könnten. So liefert z.B. die Operation

```
'unmittelbaren'.strip('un')
```

als Ergebnis `'mittelbare'` zurück, wobei nicht nur das Präfix `{un}`, sondern auch das `<n>` der Flexionsendung `{en}` entfernt wird, was normalerweise nicht beabsichtigt ist. Es gibt zwar auch die Varianten `lstrip()` und `rstrip()`, die jeweils links (das heißt am Anfang) oder rechts (das heißt am Ende) die Zeichen entfernen. Aber da die Zeichen ungeachtet ihrer Reihenfolge entfernt werden, könnte dies auch zu Problemen führen, so dass diese Methoden eigentlich nur für die Bereinigung von Leerzeichen geeignet sind. Allerdings werden wir bald noch bessere Optionen kennenlernen, um solche Ersetzungen zielgenauer vornehmen zu können.

Zumindest wäre theoretisch ein Lösung für dieses Problem das ‚Abschneiden‘ von Ketten bestimmter Länge am Anfang oder Ende mittels der **Slicing**-Technik, die wir in 4.3 näher besprechen werden. Sie kann auf alle listenartigen Konstrukte angewandt werden, auf die sich über Indexpositionen zugreifen lässt. Dafür müsste man sich aber sicher sein, dass jegliche Prä- oder Suffixe, die man entfernen will, auch immer dieselbe Länge haben.

Um alle Vorkommen einzelner Zeichen oder feste Sequenzen von Zeichen innerhalb von Zeichenketten auszutauschen oder zu entfernen, kann man die `replace()`-Methode verwenden, wobei das erste Argument das darstellt, was ersetzt, und das zweite, wodurch es ersetzt werden soll. So z.B. ersetzt die Anweisung

```
'Zeile 1\nZeile 2\nZeile 3'.replace('\n', ' ')
```

alle Zeilenumbrüche durch Leerzeichen. Alle drei Verfahren haben jedoch mehr oder weniger starke Beschränkungen und können meist nur verwendet werden, wenn genaue, bekannte Bedingungen zutreffen oder etwas getestet werden soll.



Übung 19 - Zeichenketten bereinigen

Schreiben Sie das Programm `08_bereinigung.py`, in dem Sie verschiedene Bereinigungsmethoden austesten sollen, um Leerzeichen zu löschen.

Legen Sie zunächst eine Zeichenkette mit zwei Wörtern an, an deren Anfang, Mitte und Ende jeweils zwei Leerzeichen stehen.

Verwenden Sie dann eine `print()`-Anweisung, in der Sie die `strip()`-Methode, mit beschreibendem Begleittext, wodurch das Ergebnis erzielt wurde, ohne Argumente auf diese Kette anwenden. Um die Kette besser erkennen zu können, sollten Sie bei der Ausgabe mit Konkatenation arbeiten und den Beginn und das Ende des Ergebnisses jeweils durch `>>` und `<<` markieren.

Verwenden Sie danach eine `print()`-Anweisung, in der Sie gleichermaßen die `replace()`-Methode mit zwei Leerzeichen als erstem Argument und einem Leerzeichen als zweitem Argument auf diese Kette anwenden, wieder mit geeigneter Nachricht.

Verknüpfen Sie danach in einer dritten `print()`-Anweisung die `replace()`- und die `strip()`-Methode, indem Sie sie nacheinander auf die Kette anwenden.

Welche Schlüsse können Sie aus diesen Schritten ziehen?

Bisher haben wir Zeichenketten einfach als feste Kombinationen behandelt, wollen im nächsten Abschnitt aber lernen, wie wir sie als Sequenzen von einzelnen Zeichen behandeln können.

4.3 Mit Sequenzen arbeiten

4.3.1 Allgemeine Sequenzen und Listen

Wie wir schon gelernt haben, sind Zeichenketten eine spezielle Art von Listen und stellen somit eine der Formen von Sequenzen in Python dar. Alle Sequenzen haben bestimmte Funktionen/Operationen gemeinsam. So z.B. lässt sich mittels der `len()`-Funktion und der Sequenz als Argument die Länge dieser Sequenz ermitteln, oder zählt die `count()`-Methode, wie oft ein Argument in der Sequenz enthalten ist. Mithilfe des `in`-Operators, den wir schon im Zusammenhang mit `for`-Schleifen kennengelernt hatten, kann man herausfinden, ob ein Element in einer Sequenz

enthalten ist, oder auch testen, ob dies nicht der Fall ist, wenn man den `in`-Operator mit `not` verneint. Letzteres kann man anwenden, um bestimmte Datenteile zu überspringen oder zu filtern, z.B. indem man

```
if wort not in satz:
```

schreibt, wobei `satz` hier natürlich eine Liste von Wörtern darstellt.

Wie ebenfalls schon angesprochen, kann auf Elemente von Sequenzen meist durch ihre Indexpositionen zugegriffen werden. Bisher hatten wir aber immer nur einzelne Indexpositionen verwendet, so wie in unserer Simulation der syntaktischen Inversion, in der wir die ersten zwei Elemente einer Wortliste vertauscht ausgegeben hatten. Allerdings kann auch auf mehrere Elemente über die sogenannten **Slices** – das heißt buchstäblich Abschnitte – zugegriffen werden, die dann in Form von Bereichen innerhalb der eckigen Klammern angegeben werden. Diese Bereiche sind über Doppelpunkte voneinander getrennt, wobei jeweils die Start- und Endpositionen am Anfang stehen, aber als Besonderheit optional nach einem zweiten Doppelpunkt auch die Schrittweite angegeben werden kann, mit der die Elemente extrahiert werden sollen. Fehlt die Schrittweite, wird sinnvollerweise automatisch 1 angenommen. Die allgemeine Slice-Syntax sieht deshalb so aus.

```
Sequenzname[Startposition:Endposition(:Schrittweite)]
```

Wichtig ist wieder zu beachten, dass der Index bei 0 anfängt, und dass die Endposition *exklusiv* ist, so dass, wenn wir eine Variable `wort` mit der Zeichenkette zugeben initialisieren und dann ein Slice über `wort[0:2]` generieren, tatsächlich nur zwei Buchstaben bis zum zweiten ausgegeben werden und das Ergebnis somit `zu` ist, also nur das, was an den Positionen `wort[0]` und `wort[1]` steht. Was wir also damit erreicht haben, ist, dass wir das trennbare Präfix extrahiert haben. Wenn wir jedoch eher Interesse daran haben, dieses zu entfernen, um den Stamm zu extrahieren, dann können wir dies mit einer abgekürzten Schreibweise erreichen, nämlich durch `wort[2:]`, was in unserem Fall dasselbe erzielt wie `wort[2:7]`, nur dass wir die Länge des ursprünglichen Wortes dabei gar nicht wissen müssen, da beim Weglassen der Endposition immer alles bis zum Ende der Sequenz zurückgegeben wird. Dies erspart uns Arbeit und Code, weil wir ansonsten unter Umständen erst die Wortlänge selbst ermitteln hätten müssen. Dies wäre auch nicht sehr schwer, zumal wir dafür die `len()`-Funktion verwenden könnten, entweder in einer getrennten Anweisung, oder direkt in der Anweisung `wort[2:len(wort)]`, was aber um einiges umständlicher wäre, so dass Sie hoffentlich jetzt schon die Option zu schätzen wissen, einfach die Endposition freizulassen. Dieselbe Form der Abkürzung können Sie übrigens auch am Anfang verwenden, wobei das Weglassen der Anfangsposition immer bedeutet, dass ab Position 0 extrahiert wird, z.B. `wort[:5]`. Wir werden später noch sehen, wozu die beiden oben angeführten

Teiloperationen nützlich sind, aber vielleicht können Sie es sich ja mit Ihrem Wissen über die deutsche Verbmorphologie schon vorstellen.

Eine andere Form der Arbeitserleichterung bietet Python dadurch, dass man als Indexpositionen auch negative Werte angeben kann. Damit lässt sich beispielsweise mit

```
stamm = verb[:-2]
```

bei vielen Verben, aber natürlich nicht allen, aus dem Infinitiv der Stamm extrahieren. Da auch die Schrittweite negativ sein kann, kann man z.B. auch ein Wort mittels

```
wort[::-1]
```

umkehren, was z.B. zur rückläufigen Sortierung von Wortlisten nützlich ist, um dadurch Wörter mit gleichen Endungen/Suffixen erkennen zu können.

Wie Sie aus den obigen Ausführungen wahrscheinlich unschwer erkennen können, sind solche Slicing-Operationen insbesondere in der Morphologie von großem Nutzen.

4.3.2 Tupel

Tupel stellen eine andere, spezielle Form von Listen dar, bei denen die Elemente, sobald Sie einmal angelegt sind, nicht mehr verändert werden können, es sei denn, sie werden wieder komplett neu erzeugt. Sie können mittels der `tuple()`-Funktion angelegt werden und eignen sich insbesondere zur Speicherung zusammengehöriger Informationen, wie z.B. fester Eigenschaften, Übergabe von Argumenten, oder dem Vertauschen von Variablenwerten.

Wenn Sie nicht über die `tuple()`-Funktion erzeugt werden, werden sie in `()` initialisiert oder angegeben, z.B.

```
tags = ('Adjektiv', 'Adverb')
```

Zwar sind die einmal angelegten Elemente einer Tupel-Sequenz unveränderlich, das heißt die Zuweisung `tags[0] = 'Nomen'` würde zu einem Fehler führen, aber sonst sind ähnliche Operationen wie bei allen anderen index-basierten Sequenzen möglich.

Wie oben schon angesprochen, kann man Tupel zur Lösung unseres Tauschproblems aus Übung 7 anwenden. Um hier nicht eine zusätzliche Hilfsvariable anlegen zu müssen, kann man viel einfacher die Anweisung

```
(wort1, wort2) = (wort2, wort1)
```

verwenden.



Übung 20 - Den Stamm trennbarer Verben anzeigen

Schreiben Sie das Programm `09_stamm_bildung.py`, in welchem jeweils der Stamm trennbarer Verben mit ausgewählten Präfixen angezeigt werden soll.

Dabei soll auf der Kommandozeile als Argument eine Zeichenkette übergeben werden, in der durch Kommas getrennte Präfixe stehen, wobei auf die Kommas kein Leerzeichen folgen darf.

Diese Kette soll im ersten Schritt mittels `split()` in einer einzelnen Anweisung in eine Liste zerlegt, in ein Tupel gewandelt und einer geeigneten Variable zugewiesen werden.

Danach soll eine weitere Kette angelegt werden, in der Sie eine Liste trennbarer Verben, durch Leerzeichen voneinander getrennt, angeben, wobei pro Präfix jeweils mindestens zwei Verben auftreten sollten.

Diese Liste soll dann innerhalb einer `for`-Schleife wieder durch `split()` getrennt und abgearbeitet werden.

Innerhalb der Schleife sollen nur die Stämme (mittels `Slice`) für Verben ausgegeben werden, die mit einem der Präfixe anfangen. Um auf dies zu prüfen, können Sie zunächst die `startswith()`-Methode mit dem Präfix-Tupel als Argument verwenden, müssen dann aber noch einmal in einer `for`-Schleife alle Präfixe abarbeiten und testen, ob das Präfix tatsächlich mit dem jeweiligen Verb auftritt.

Wenn Sie das jeweilige Präfix gefunden haben, können Sie den Stamm extrahieren, indem Sie das Präfix und die Infinitivendung abschneiden. Allerdings müssen Sie dabei beachten, dass nicht alle Infinitive auch auf `{en}` enden, weshalb Sie die Sonderfälle, basierend auf Ihrem Wissen über die Verbmorphologie, extra behandeln müssen.

Testen Sie das Programm mit verschiedenen Präfixkombinationen.

In dieser Übung haben wir gesehen, wie man Zeichenketten fester Länge abschneiden kann, und wollen im nächsten Abschnitt besprechen, wie man auf flexiblere Art und Weise Zeichenketten, basierend auf anderen Analysen, extrahieren kann.

4.4 Zeichenketten extrahieren

Wie wir gesehen haben, erlaubt die `Slice`-Syntax eine einfache Extraktion von Ketten bekannter Länge und Position. Aber manchmal ist die genaue Position unbekannt, z.B. bei Infixen. In diesem Fall müssen wir zunächst die Anfangsposition, also den Anfangsindex, des gesuchten Teils ermitteln, was wir mithilfe der passend benannten `index()`-Methode der Zeichenkette erreichen. Diese Methode verlangt eine Zeichenkette als Argument und erlaubt optional auch die Angabe von Start- und Endpositionen zum Suchen.



Übung 21 - Infixe entfernen

Schreiben Sie das Programm `10_infix_loeschen.py`, in dem das Infix {zu} aus den speziellen Infinitivformen mit {zu} trennbarer Verben, wie z.B. *anzurufen*, gelöscht werden soll.

Legen Sie dazu zunächst eine Variable für das Infix an.

Ermitteln und speichern Sie dann die Länge des Infixes.

Legen Sie jetzt wieder eine Zeichenkette an, mit Komma als Trenner, in der Sie einige Infinitivformen trennbarer Verben auflisten.

Wie im letzten Programm, durchlaufen Sie dann diese Liste mit einer Schleife.

Ermitteln Sie innerhalb der Schleife zuerst die Indexposition des Anfangs des Infixes im Wort mit der `index()`-Methode und speichern Sie diese.

Geben Sie dann in einer `print()`-Anweisung das ursprüngliche Wort mit Infix und das Wort ohne Infix aus, wobei Sie Letzteres mithilfe von zwei Slices des Wortes aus dem Teil vor der Indexposition und dem Teil nach dem Infix zusammensetzen.

In den letzten zwei Übungen hatten wir geübt, Zeichenketten zu zerlegen. Jetzt besprechen wir, wie wir diese am besten zusammenfügen.

4.4.1 Zeichenketten effizient zusammenfügen

Bisher hatten wir schon zwei Optionen besprochen, um Zeichenketten ‚zusammenzufügen‘: die Konkatenation über den `+`-Operator und die Wiederholung mittels des `*`-Operators. Letztere ist für die sprachwissenschaftliche Arbeit eher weniger nützlich, es sei denn, man will Reduplikation simulieren. Allerdings hat die Konkatenation aus Effizienzsicht den Nachteil, dass immer mehr Speicherplatz zum Zwischenspeichern verwendet werden muss, da Zeichenketten in Python ja nicht veränderlich sind, so dass jedes Mal eine neue angelegt wird. In der Praxis wird sich dies erst bei größeren Programmen und der Generierung von langen Ketten auswirken, aber dennoch ist es meist besser, zum Zusammenfügen die `join()`-Methode des `str`-Objekts zu verwenden, welche sehr effizient zum Verknüpfen von Zeichenkettenlisten, wie z.B. Wortlisten oder ‚Sätzen‘, ist. Die Syntax dieser Methode ist jedoch etwas gewöhnungsbedürftig, da sie nicht, wie man vielleicht annehmen würde, auf eine schon bestehende Kette angewandt wird. Stattdessen nimmt sie eine beliebige Kette als Ausgangspunkt für die Verknüpfung und fügt diese dann zwischen allen Elementen einer Sequenz als ‚Trenner‘ ein, also

```
'Trennerkette'.join(Kettensequenz)
```

wobei diese Kette auch leer sein darf. In der Praxis lässt sich dies gut zum Zusammen-
setzen von Wörtern zu einem Satz mittels Leerzeichen oder von Zeilen zu Dateien mit
Zeilenumbrüchen verwenden. Diese Methode hat auch den Vorteil, dass tatsächlich
nur zwischen den Elementen etwas eingefügt wird und nicht auch am Ende, so wie das
unter Umständen bei mehrfacher Konkatenation in einer Schleife geschehen würde.

4.4.2 Groß- und Kleinschreibung handhaben

Neben den Möglichkeiten, Zeichenketten zusammenzufügen, zu zerschneiden, oder
Teile zu extrahieren, bietet Python auch verschiedene Methoden, um mit Groß- oder
Kleinschreibung umgehen zu können. So z.B. wandelt `upper()` eine komplette Kette
in Großbuchstaben um und `lower()` tut genau das Gegenteil. Vor allem `lower()` wird
häufig beim Vergleich von Wörtern aus Wortlisten herangezogen, um Schwierigkeiten
mit großgeschriebenen Wörtern am Satzanfang zu vermeiden. Allerdings ist das vor
allem im Deutschen nicht immer unproblematisch: Die Großschreibung ist schließlich
ein Kriterium zur Unterscheidung von Nomina und anderen Wortklassen, Nomina sind
sehr häufig auftretende Wörter, und einige von ihnen unterscheiden sich nur durch
die Großschreibung von bestimmten Verbformen, wie z.B. bei (*der*) *Bestand* und (*sie*)
bestand (*darauf*). In Sprachen wie dem Englischen ist dies weniger problematisch, da
dort nur die seltener auftretenden Eigennamen am Anfang großgeschrieben werden.

Speziell für das Deutsche ist auch eine andere Methode interessant, nämlich
`casefold()`, was wie `lower()` alle Buchstaben in Kleinbuchstaben, aber auch äquiva-
lente Sequenzen, wie z.B. *ß* zu *ss*, umwandelt. Dies kann beispielsweise problemlos
beim Wortvergleich für die Sortierung und Generierung von Wortlisten für Lexika
verwendet werden, da dort zum einen diese quasi-äquivalenten Zeichen richtig ein-
sortiert werden sollen und zum anderen im Lexikon zumindest bei der Reihenfolge der
Auflistung nicht zwischen Groß- und Kleinschreibung unterschieden wird.

Während die eben beschriebenen Methoden immer alle Elemente einer ganzen Kette
gleichzeitig verändern, macht `capitalize()` nur den Anfangsbuchstaben einer Kette
groß, was z.B. bei der Konversion von Verben in Nomina, wie in (*etwas*) *tun* zu (*das*)
Tun, verwendet werden kann, oder, um ein Wort am Satzanfang konventionsgerecht
auszugeben, so wie wir dies z.B. bei Ausgabe der syntaktischen Inversion gerne schon
vorher getan hätten. Die Methode `title()` wandelt alle kleingeschriebenen Wörter in
solche mit anfänglichem Großbuchstaben um, z.B. um im Englischen den sogenannten
title case für Überschriften zu produzieren. Allerdings liefert sie normalerweise keine
perfekten Ergebnisse, da ja kurze Funktionswörter auch in Titeln kleingeschrieben
werden. Die Methode `swapcase()` schließlich wandelt immer Groß-/Kleinschreibung
in das Gegenteil um, auch bei gemischt geschriebenen Wörtern.



Übung 22 - Konversion von Groß- und Kleinschreibung

Testen Sie mithilfe einiger selbstgewählten längeren Zeichenketten, die auch deutsche ‚Sonderzeichen‘ beinhalten, die oben beschriebenen Methoden in der Python-Shell, um ein Gefühl für ihre Verwendung und Nutzbarkeit zu entwickeln.

4.5 Zeichenketten formatieren

Bisher haben wir Zeichenketten nur zusammengefügt oder ihr Format bezüglich der Groß- oder Kleinschreibung kontrolliert, und sie dann im Rohformat ausgegeben. Um jedoch Ergebnisse zusammen mit beschreibendem Text auszugeben, ist es etwas ineffizient, immer mittels Konkatenation oder der `join`-Methode Ausgaben zu erzeugen. Zudem will man manchmal auch Ausgaben in Schleifen erzeugen, bei der bestimmte Ausgabeteile immer sauber untereinander stehen, so wie in einer Tabelle. Letzteres werden wir z.B. für die Ausgabe von Frequenzlisten in Kapitel 8 verwenden. Um solche Dinge zu erreichen, bietet Python, je nach Version, verschiedene Möglichkeiten zur Formatierung oder Interpolation von Werten, die wir im Folgenden besprechen wollen.

4.5.1 Verwendung des %-Operators

Die älteste, aber auch unübersichtlichste Art, Zeichenketten formatiert auszugeben, ist die Verwendung des %-Operators, die wir schon in Abschnitt 2.6.2 gesehen haben. Da die anderen, unten besprochenen, Formatierungsmethoden jedoch besser sind, besprechen wir die Verwendung von % hier nur, um es Ihnen zu erlauben, auch älteren Code zu verstehen.

Bei Anwendung dieser Technik werden innerhalb einer Zeichenkette zunächst Platzhalterpositionen durch Prozentzeichen und Angaben für den Datentyp sowie optional erweiterte Formatangaben markiert. Dann stellt man hinter der Zeichenkette ein weiteres Prozentzeichen und gibt dahinter einen Tupel mit Füllern für die Platzhalterpositionen an, die dann der Reihe nach in die Zeichenkette eingesetzt werden. Als Datentypmarkierungen stehen `d` für Integer, `f` für Fließkommazahlen und `s` für Zeichenketten zur Verfügung. Durch die Angabe des Datentyps spart man sich auch die explizite Konversion für Zahlen, die sonst bei der Konkatenation nötig wäre, und kann zudem auch noch Längen- und Ausrichtungsangaben für alle Datentypen festlegen sowie die Anzahl der Nachkommastellen bei Fließkommazahlen.

4.5.2 Die `format()`-Methode

Die `format()`-Methode der Zeichenkette erlaubt ebenfalls die Interpolation, also das Einfügen, von Variableninhalten oder Zeichenketten, diesmal mittels `{...}`-Platzhal-

tern. Zusätzlich bietet sie aber auch die Möglichkeit der Verwendung von Schlüsselwortargumenten für eine flexiblere Formatierung der Ausgabe. Die Platzhalter werden dabei einfach in geschweiften Klammern innerhalb der Zeichenkette angegeben, wobei die Inhalte und Schlüsselwortargumente als Argumente der Methode aufgelistet werden. Wir besprechen hier nur die für uns wichtigsten Optionen dieser Methode. Die allgemeine Syntax ist wie folgt.

```
'Zeichenkette'.format(Argument(e) [, Schlüsselwortargumente])
```

So zum Beispiel kann man eine Liste von Wörtern und deren Häufigkeiten in einem Text innerhalb einer `for`-Schleife wie in Abbildung 4 gezeigt ausgeben, gemäß der vorher ermittelten Länge des längsten Wortes formatiert.

```
'{0:{laenge}}\t{1}'.format(wort, haeufigkeit, laenge=laengstes_wort)
```

Abb. 4: Beispiel für die `format`-Methode

Hierbei wird im ersten Platzhalter, der explizit durch die `0` als solcher markiert ist, der Inhalt der ersten als Argument auftretenden Variable (also `wort`) eingefügt. Das zweite Argument (`haeufigkeit`) wird in den ebenfalls mithilfe von `1` explizit angegebenen Platzhalter eingefügt. Die Länge der Ausgabe vor dem Tabulator wird über das Schlüsselwort `laenge`, welches wiederum in geschweiften Klammern hinter dem Doppelpunkt innerhalb des ersten Platzhalters eingefügt ist und hinter den zwei ersten Argumenten in der Argumentenliste erscheint, auf die Länge des längsten Wortes gesetzt, welche in der vorher belegten Variable `laengstes_wort` gespeichert ist. Wörter, die kürzer als diese Länge sind, bekommen dann automatisch so viele Leerzeichen angefügt, bis die Anzahl von Zeichen erreicht wird, woraufhin der Tabulator und die Häufigkeit ausgegeben werden.

Wie aus der Erklärung von oben hoffentlich schon deutlich geworden ist, können Platzhalter auf verschiedene Art angegeben werden. Wenn sie leer sind (`{}`), dann ist die Position implizit; sind sie nummeriert (z.B. `{0}`), dann liegt eine explizite Position vor, die auf eine der Positionen in der Argumentenliste verweist. Zu guter Letzt gibt es die Möglichkeit, Platzhalter zu benennen (z.B. `{eins}`), wobei diese Bezeichnung als Schlüsselwort fungiert und als solche in der Argumentenliste auftreten muss (z.B. `'eins'=wort`). Bei den letzteren zwei Optionen ist es theoretisch auch möglich, die Platzhalter beliebig innerhalb der formatierten Kette zu vertauschen, da ja explizit auf sie verwiesen wird.

Die erweiterten Formatierungsoptionen, die auf den Doppelpunkt folgen, können Angaben zu Datentyp, Länge und/oder Ausrichtung des Platzhalters beinhalten. Den

Datentyp kann man wieder als Integer (`{1:d}`), Fließkommazahl (`{1:f}`) oder Zeichenkette (`{1:s}`) markieren, dem dann eventuell die entsprechenden Längenangabe und Ausrichtung vorangestellt werden. Um die Ausrichtung anzuzeigen, verwendet man eine spitze Klammer als ‚Pfeil‘. Hier steht also `>` für rechtsbündig, und `<` für linksbündig, wobei man auch eine Zentrierung durch einen nach oben gerichteten ‚Pfeil‘ also ein Caretzeichen (`^`) erreichen kann. Die Länge kann entweder durch eine feste Zahl (`{1:7d}`; maximal 7 Ziffern breit) festgelegt werden oder, wie wir in unserem Beispiel oben gesehen haben, durch ein Schlüsselwort mit späterer Variablenzuweisung in der Argumentliste (`{1:{breite}d}`). Für Fließkommazahlen kann auch die Präzision der Nachkommastellen angegeben werden (`{1:.5f}`), was nützlich für die Angabe relativer Frequenzen ist. Um die komplette Länge einer Fließkommazahl anzugeben, und nicht nur die Präzision, kann man dem Punkt eine ganze Zahl voranstellen (`{1:10.5f}`), wobei hier wieder mit Leerzeichen aufgefüllt wird, falls die Zahl kürzer ist.

4.5.3 Verwendung von f-strings

Seit Python 3.6 gibt es zusätzlich eine noch elegantere Methode, interpolierte Ausgaben zu erzeugen, die sogenannten **f-strings**. Sie erlauben die Interpolation von Variablen, Zeichenketten und sogar Funktionen in eine Zeichenkette, indem man vor dem öffnenden Anführungszeichen ein `f` setzt und, wie bei der `format`-Methode, die zu interpolierende Variable oder Funktion in geschweiften Klammern angibt. Diese Schreibweise ist viel übersichtlicher, da man sofort erkennen kann, was in die Zeichenkette interpoliert wird, was aus der f-string-Version des vorherigen Beispiels hoffentlich leicht deutlich wird:

```
f'{wort:{laengstes_wort}}\t{haeufigkeit}'
```

Wie schon bei der `format`-Methode werden die erweiterten Optionen dann hinter einem Doppelpunkt angeführt. Zudem erspart man sich hier die Verwendung von Schlüsselwörtern, da die Variablen alle direkt interpoliert werden.

Nachdem wir jetzt über ein ausreichendes Wissen zu Zeichenketten verfügen, können wir uns im nächsten Kapitel Optionen zuwenden, wie man längere solcher Ketten, die unsere Analysedaten repräsentieren, aus externen Quellen einlesen und schrittweise abarbeiten kann.

4.6 Lösungen zu den Aufgaben

Lösung 18 - Normale und rohe Zeichenketten unterscheiden

Sie sollten bei dieser Übung erkennen können, dass bei der ersten Ausgabe vor dem Wort ein Zeilenumbruch ausgegeben wird, wobei bei der zweiten genau das, was auch innerhalb der Anführungszeichen steht, widergegeben wird.



Lösung 19 - Zeichenketten bereinigen

Wir können dies der Einfachheit halber so schreiben:

```
kette = ' Wort1 Wort2 '.
```

Die erste `print()`-Anweisung sollte dann in etwa so aussehen:

```
print('Mit strip()-Methode ohne Argumente: >>' +  
      kette.strip() + '<<')
```

und die zweite

```
print('Mit replace()-Methode: >>' + kette.replace(' ', ' ') +  
      '<<')
```

Wie Sie hier deutlich sehen können, verändert sich die ursprüngliche Kette nicht durch die Anwendung der Operationen.

Die dritte `print()`-Anweisung dürfte etwas schwieriger zu schreiben sein, da es wahrscheinlich sehr ungewohnt für Sie aussehen wird, beide Methoden direkt hintereinander zu verknüpfen, anstatt sie nacheinander in zwei getrennten Anweisungen auszuführen. Das wäre natürlich ebenso möglich. Dabei müssten Sie aber das erste Ergebnis zunächst in einer zusätzlichen Variable zwischenspeichern, die Sie dann innerhalb der letzten `print()`-Anweisung ausgeben könnten. Wenn wir aber, so wie in der Übung vorgesehen,

```
print('Mit beiden Methoden: >>' +  
      kette.replace(' ', ' ').strip() + '<<')
```

schreiben, dann ist dies viel effizienter.

Lösung 20 - Den Stamm trennbarer Verben anzeigen

Bei dieser Übung müssen Sie zunächst einmal wieder das `sys`-Modul importieren, um über die Kommandozeile das Argument abzufragen, welches natürlich in `sys.argv[1]` steht. Die erste Anweisung im Programm ist diesmal etwas komplexer, da Sie zwei Funktionen/Methoden ineinander verschachteln müssen, um das Ergebnis der Variable direkt zuzuweisen, so dass sie ungefähr so aussehen sollte:

```
praefixe = tuple(sys.argv[1].split(','))
```

Bei der Eingabe ist sehr wichtig, dass Sie in der Zeichenkette keine Leerzeichen haben. Sonst liefert die `split()`-Methode der `str`-Klasse nicht das richtige Ergebnis, da wir bei jedem Auftreten eines Kommas, das von einem Leerzeichen gefolgt ist, ein Verb in die Liste schreiben, an dessen Anfang ein Leerzeichen steht. Wir werden in Kapitel 6 eine bessere Methode eines anderen Objekts, aber mit gleichem Namen, kennenlernen, die uns hier mehr Flexibilität bieten würde, so dass sowohl Kommata ohne als auch

mit folgenden Leerzeichen verarbeitet werden könnten, um die Fehlerträchtigkeit des Programms zu verringern.

Die Liste, die im zweiten Schritt als Zeichenkette angelegt werden soll, sollte ebenfalls nur einzelne Leerzeichen beinhalten. Die Zeichenkette verwenden wir zur Bequemlichkeit, da wir die Liste später in der Schleife leicht aufspalten können, aber uns bei der Eingabe Arbeit ersparen wollen, weil wir somit nicht alle Elemente einzeln als Zeichenketten anlegen und der Liste zuweisen müssen. Die Anweisung selbst könnte dann z.B. so aussehen:

```
verb_kette = 'abschütteln angeben anhören abholen abtreten '
            'beleidigen betreten entfernen entleeren übergeben '
            'überlegen wiederholen 'wiedergeben verlieren verteilen '
            'zerstören zuhören'
```

Da wir mittlerweile schon etwas an verschachtelte Konstruktionen gewöhnt sind, sollte die erste `for`-Schleife

```
for verb in verb_kette.split():
```

Ihnen jetzt nicht weiter schwerfallen. Natürlich muss hier in der `split()`-Methode, die die Liste generiert, kein Argument angegeben werden, da die Standardeinstellung automatisch an Leerzeichen aufspaltet.

Innerhalb der `for`-Schleife können wir jetzt mittels einer `if`-Anweisung testen, ob die Verben in der Liste tatsächlich mit einem der Präfixe anfangen, um die Liste entsprechend zu filtern. Dabei testen wir mithilfe des Tupels von Präfixen bei jedem Wort nur einmal die Bedingung, um nicht alle Präfixe in einer Schleife abarbeiten zu müssen. Was wir hier aber leider nicht erfahren, ist, welches Präfix tatsächlich vorhanden war, da die `startswith()`-Methode nur einen booleschen Wert zurückliefert. Deshalb müssen wir leider, nachdem wir durch

```
if verb.startswith(praefixe):
```

getestet haben, noch einmal eine `for`-Schleife über alle Präfixe laufen lassen und dort noch einmal einen Test laufen lassen, ob das Verb auch tatsächlich mit dem jeweiligen Präfix beginnt. Dabei verwenden wir diesmal nur das Präfix als Argument für die `startswith()`-Methode, so dass der Schleifenkopf

```
for praefix in praefixe:
```

wäre und die Bedingungsabfrage dann

```
if verb.startswith(praefix):
```

Die Fälle, die wir im letzten Schritt behandeln müssen, stellen zwei Alternativen dar, weshalb wir eine weitere `if`-Anweisung mit einem alternativen `else`-Block benötigen. Die Alternativen sind, dass falls das vorletzte Phonem/der vorletzte Buchstabe ein Liquid ist, also entweder ein *l* oder ein *r*, wir natürlich nur das *n* am Ende des Infinitivs

neben dem Präfix abschneiden dürfen, wohingegen sonst immer das Präfix und die letzten zwei Buchstaben, also *en*, abgeschnitten werden sollen. Also sollte unsere *if*-Anweisung wie folgt aussehen:

```
if verb[-2:-1] == 'l' or verb[-2:-1] == 'r':
```

Im darauffolgenden Block müssen wir nur eine entsprechende Meldung und ein Slice des Verbs ausgeben, welches als Anfangsposition die Länge des Präfixes hat, und als Endposition alles, außer dem letzten Buchstaben, also

```
print('Der Stamm von', verb, 'ist:', verb[len(praefix):-1])
```

Da wir im letzten Fall fast dasselbe Ausgeben müssen, nur dass diesmal die letzten zwei Buchstaben abgeschnitten werden sollen, lautet die Anweisung im *else*-Block

```
print('Der Stamm von', verb, 'ist:', verb[len(praefix):-2])
```

Lösung 21 - Infixe entfernen

Die Variable für das Infix anzulegen, erfordert nur eine einfache Zuweisung, weshalb wir das hier nicht besprechen müssen. Ebenso dürfte das Ermitteln der Länge des Infixes kein Problem darstellen, da wir die `len()`-Funktion ja erst im letzten Programm verwendet hatten. Auch das Anlegen von Zeichenketten als ‚Listenbehälter‘ und Iterieren über die mit `split()` generierten Elemente hatten wir ja gerade geübt, weshalb eigentlich erst das Ermitteln der Indexposition des Infixes etwas Neues, aber eigentlich nicht sehr schwieriges, darstellt. Theoretisch könnten wir bei der Anwendung der `index()`-Methode eigentlich nur den ersten Buchstaben des Infixes angeben, da immer die Anfangsposition der Zeichenkette gefunden wird. Allerdings ist es dennoch besser, das gesamte Infix anzugeben, da sonst auch manchmal derselbe Buchstabe in der Kette gefunden werden könnte, selbst wenn er nicht Teil unseres Infixes ist. Deshalb könnte unsere Anweisung

```
indexPos = wort.index('zu')
```

oder natürlich auch

```
indexPos = wort.index(infix)
```

sein, da wir das Infix ja am Anfang in einer Variable gespeichert hatten.

Im letzten Schritt bleibt uns also nur die Ausgabe, wie üblich mit informativer Meldung, die dann so aussehen sollte:

```
print('Wort mit Infix:', wort + '; ohne Infix:',  
      wort[0:indexPos] + wort[indexPos+infixLaenge:])
```

Hierbei erhalten wir das Präfix des ursprünglichen Worts mittels des ersten Slices, da wir ja ab dem ersten Buchstaben bis exklusive der Startposition des Infixes extrahieren. Dann addieren wir einfach die Anfangsposition des Infixes und dessen Länge, und

erhalten somit die Anfangsposition des Infinitivs ohne Präfix, die wir schließlich ab dieser Position bis zu Ende des ursprünglichen Wortes mittels des zweiten Slices extrahieren und die beiden Slices miteinander konkatenieren. Hierbei können wir die Endposition einfach weglassen, sofern wir den Doppelpunkt nicht vergessen haben.

Lösung 22 - Konversion von Groß- und Kleinschreibung

Bei dieser Übung sollten Sie erkannt haben, dass die `casefold()`-Methode leider mit Umlauten nicht funktioniert, also z.B. nicht *ä* in *ae* umwandelt, weshalb sie nicht ganz so nützlich ist wie sie per Definition erscheint.

5 Mit gespeicherten Daten arbeiten



In diesem Kapitel sollen Sie lernen, wie man in Python mit gespeicherten Daten arbeiten kann. Dazu gehört es, die verschiedenen Mechanismen zu verstehen, mit denen man Dateien zum Lesen und Schreiben öffnen und wieder schließen kann, aber auch, wie dabei eventuell auftretende Fehler behandelt werden können. Neben der Arbeit mit Dateien werden auch Möglichkeiten besprochen, wie man sicher und effizient plattformübergreifend mit Verzeichnissen und Pfaden arbeitet.

5.1 Gespeicherte Daten

Es gibt zwar mehrere Arten von Speicherformen für Daten, aber die in der sprachwissenschaftlichen Programmierung am häufigsten verwendete Form sind Textdateien, weshalb wir hier auch nur diese behandeln. Die dabei zu verarbeitenden Dateien sind meist lokal gespeichert. Manchmal muss man aber auch über ein Netzwerk oder das Internet darauf zugreifen. Dafür werden wir in Kapitel 10 eine Option besprechen.

Der Umgang mit gespeicherten Daten erfordert meist auch die Handhabung von Verzeichnisstrukturen, so wie wir das beim Erlernen der Verwendung der Kommandozeile gesehen hatten, z.B. zum Lesen, Erstellen und Navigieren, oder zum Abfragen von Dateieigenschaften wie Dateityp oder Zugriffsberechtigungen. Allerdings sind diese Prozesse zum Teil auch fehlerträchtig, da vielleicht der Zugriff nicht immer gewährleistet ist oder man einen nicht-existierenden Dateinamen angegeben hat, was ohne geeignete Fehlerbehandlung zu Programmfehlern oder sogar Programmabstürzen führen kann. Deshalb werden wir bald einige Techniken zur Fehlerbehandlung im Detail besprechen.

5.1.1 Dateien öffnen und schließen

Wie die anderen Datentypen, die wir bisher kennengelernt haben, sind Dateien in Python auch Objekte, und zwar vom Typ `file`. Um auf die Inhalte von Dateien zuzugreifen, gibt es verschiedene Modi zum Öffnen, die über das Schlüsselwortargument `mode` angegeben werden können. Fehlt dieses Argument, wird automatisch angenommen, dass die Datei nur zum Lesen geöffnet werden soll. Wir wollen hier wieder nur die für uns wichtigsten Modi besprechen. Wichtig ist allerdings auch, dass durch das Öffnen – zumindest zum Lesen – zunächst die Option zum Arbeiten mit einer Datei geboten wird. Ihr Inhalt wird erst später über andere Methoden tatsächlich abgearbeitet.

Am Anfang der meisten Verarbeitungsschritte steht normalerweise das Lesen, für welches der `mode`-Wert `r` (für engl. *read*) ist. Sobald die Datei zum Lesen geöffnet ist, kann man auf die komplette Datei oder nur Teile, wie einzelne Zeilen oder Zeichen, zugreifen.

Verwendet man den Modus `w` für das Schreiben (engl. *write*), so wird dadurch entweder automatisch eine neue, leere Datei erzeugt oder der Inhalt einer bereits existierenden Datei überschrieben. Letzteres geschieht, selbst wenn die Datei zunächst nur zum Zugriff geöffnet wird, was anfänglich leicht zu Fehlern führen kann! Deshalb sollte man diesen Modus nur mit Vorsicht verwenden, auch wenn er zur Ausgabe von Ergebnissen sehr wichtig ist und man in diesem Fall tatsächlich meist vorherige Ergebnisse löschen will. Wie beim Lesen kann man hier ebenfalls zeilenweise oder auch komplette Dateiinhalte auf einmal in eine Datei schreiben.

Um an eine schon bestehende Datei etwas anzufügen, falls man tatsächlich schon bestehende Ergebnisse beibehalten will, verwendet man den Modus `a` (engl. *append*). Damit wird die Datei geöffnet und die Ausgabeposition automatisch an das Ende der bestehenden Datei gesetzt. Existiert die Datei noch nicht, wird sie durch das Öffnen erst erzeugt. Dateianfang und -ende sind dann zunächst identisch.

Da Dateien auch im Binärmodus geöffnet werden können, kann man optional an den ersten Buchstaben des Modus noch ein `t` anfügen, um ein explizites Öffnen als Text zu erzwingen oder im Code als solches zu markieren. Fehlt dieses jedoch, so wird automatisch das Öffnen als Text angenommen, weshalb man es normalerweise weglässt.

Zum Öffnen des Dateiobjekts bietet Python zwei verschiedene Optionen. Bei der ersten wird die Datei erst mithilfe der Syntax

```
Dateivariablen = open(Dateiname[, mode='Modus',
                        encoding='Kodierung'])
```

geöffnet, dann abgearbeitet und schließlich wieder manuell durch

```
Dateivariablen.close()
```

geschlossen. Bei der zweiten Variante arbeitet man mit einem Block, ähnlich wie bei Bedingungsabfragen oder Schleifen, wobei ein Dateiojekt mittels des `with`-Operators und dem Schlüsselwort `as` kontextualisiert, und die Datei nach Ende des Blocks automatisch geschlossen wird. Die Syntax hierfür ist dann:

```
with open(Dateiname[, mode='Modus',
                        encoding='Kodierung']) as Dateivariablen:
    Block zum Abarbeiten
```

Diese zweite Variante sieht zwar etwas komplexer aus, hat aber neben dem automatischen Schließen auch den Vorteil, dass sie das Öffnen mehrerer Dateien gleichzeitig erlaubt, die dann auch parallel mittels der `zip()`-Funktion abgearbeitet werden können. Welche Form Sie verwenden, bleibt in den einfachen Fällen meist Ihnen überlassen. Jedoch sollten beide Optionen immer durch geeignete Fehlerbehandlung abgesichert werden!

Wie die Syntaxdarstellung oben zeigt, kann man auch mittels des Schlüsselwortarguments `encoding` eine Kodierung angeben. Da wir meist mit UTF-8-kodierten Daten arbeiten, sollten wir dies am besten immer mittels des Wertes `utf-8` angeben. Sonst wird die Voreinstellung Ihres Betriebssystems als Kodierung angenommen, was leicht zu Fehlern bei der Verarbeitung führen kann. Für die Ein- und Ausgabe werden wir in den meisten Fällen dieselbe Kodierung verwenden, wobei es allerdings auch sein kann, dass wir zur Konvertierung, insbesondere wenn wir Dateien von anderen Leuten bekommen, Dateien in einer Kodierung öffnen und dann als UTF-8 wieder ausgeben wollen.

Dateien, die im gleichen Verzeichnis wie das Programm liegen, können unter Windows normalerweise einfach über den Dateinamen ohne irgendeine Pfadangabe gefunden werden, was aber unter Linux/auf dem Mac nicht unbedingt der Fall ist. Hier empfiehlt es sich immer, einen relativen Pfad mit Angabe des aktuellen Verzeichnisses, also über `./`, zu verwenden, wobei die Pfadangabe einfach mit dem Dateinamen konkateniert werden kann, falls dieser in einer Variable abgespeichert ist. Selbst wenn Windows eigentlich einen anderen Verzeichnistrenner verwendet, funktioniert diese Pfadangabe dort auch, so dass Sie sich bei der Arbeit mit Dateien im gleichen Verzeichnis dies als Konvention angewöhnen sollten, um die Kompatibilität Ihrer Programme zu gewährleisten. Wir werden später noch robustere Möglichkeiten kennenlernen, um mit Pfadangaben auf kompatible Art und Weise umzugehen, damit Ihre Programme auch tatsächlich überall lauffähig sein sollten.

5.1.2 Dateiinhalte lesen

Beim Lesen von Dateiinhalten bietet uns das Dateiojekt verschiedene Methoden, die wir für unterschiedliche Zwecke einsetzen können. Die `read()`-Methode mit Integer-Argument liest so viele Bytes, wie als Argument angegeben sind, was allerdings für uns nur selten nützlich ist. Wird dieselbe Methode jedoch ohne Argument verwendet, so liest sie die ganze Datei auf einmal, und wir können sie dann in einer Zeichenkette speichern, um globale Ersetzungen etc. durchzuführen.

Die Verwendung von `readline()` ohne Argument liest bei jedem Aufruf immer eine Zeile, was nützlich sein kann, um n Zeilen zu überspringen, wie z.B. bei einem Header, der immer eine feste Länge hat. Beim Aufruf mit Argument werden wieder die angegebene Anzahl an Bytes gelesen, diesmal aber von der gerade aktuellen Zeile. Wurde die Methode also z.B. zunächst 5mal ohne Argument verwendet und dann mit dem Argument 3, so werden von der 6. Zeile 3 Bytes gelesen.

Die Methode `readlines()` liest alle Zeilen auf einmal und liefert sie als Liste zurück. Dies bietet oft bei kleineren Dateien eine einfache Option zum sequenziellen Abarbeiten oder Vergleich mit anderen Dateiinhalten, wie z.B. von zwei Wortlisten, die aus verschiedenen Dateien stammen. Insbesondere bei größeren Dateien jedoch bietet sich das zeilenweise Abarbeiten mittels einer Iteration über das Dateiojekt mit einer `for`-Schleife an, da dies Arbeitsspeicher spart. Die Syntax dafür ist dieselbe, wie für allgemeine Sequenzen, also

```
for Zeilenvariable in Dateiojekt:
```

5.1.3 Fehlerbehandlung

Wie schon oben angesprochen, können Fehler beim Arbeiten mit Dateien oder Verzeichnissen verschiedene Gründe haben, wie z.B. eine falsche Pfadangabe, einen falschen Dateinamen (das heißt die Datei existiert nicht, weil entweder im Programm oder auf der Kommandozeile ein falscher Name eingegeben wurde) oder das Fehlen von Berechtigungen zum Lesen und/oder Schreiben auf die Datei oder das Verzeichnis. In diesen Fällen gibt Python einen Betriebssystemfehler (`OSError`) mit Fehlerobjekt aus, welches mittels

```
OSError as Fehlerobjektname
```

gespeichert werden kann.

Die Absicherung durch Fehlerbehandlung, also das Abfangen möglicher voraussagbarer Fehler erfolgt durch einen Block, der mit dem Schlüsselwort `try` eingeleitet wird, gefolgt, wie üblich, von einem Doppelpunkt. Innerhalb dieses Blocks steht/stehen die Anweisung(en), die einen Fehler verursachen können, wie der Versuch (also engl. *try*), eine nicht-existente Datei zu öffnen.

Falls ein Fehler aufgetreten ist, erfolgt dessen Behandlung in einem `except`-Block, wo entweder der Fehler nur gemeldet, weitergeleitet oder das ganze Programm beendet werden kann. Letzteres wird z.B. oft dann geschehen, wenn das Programm zum Laufen unbedingt eine bestimmte Datei oder andere Argumente benötigt, ohne die es nicht lauffähig wäre. Manchmal kann es auch einen `finally`-Block geben, der für ‚Aufräumarbeiten‘, wie z.B. das Löschen schon angelegter Dateien etc., verwendet werden kann, wobei dieser aber eher bei größeren Programmen nötig sein wird.

Wir werden später noch mehr Fehlertypen an geeigneten Stellen besprechen, aber ich will hier zumindest eine erste Übersicht bieten. Einige dieser Fehlertypen hatte ich schon in Lösung 6 kurz angesprochen, um Sie darauf hinzuweisen, welche Fehler bei Ihren ersten Programmierschritten leicht auftreten können, aber dabei nicht im Detail erklärt, was genau sie bedeuten. Dies soll hier kurz nachgeholt werden.

Fehlertyp	Beschreibung
<code>NameError</code>	dieser Fehler tritt meist auf, wenn Sie eine Variable nicht definiert oder falsch geschrieben haben, oder wenn eine Variable oder ein Objekt nicht mit einem entsprechenden Wert instanziiert wurde
<code>SyntaxError</code>	zeigt an, dass Markierungen für Datentypen, wie Anführungszeichen oder Klammern, nicht richtig gesetzt wurden oder z.B. Doppelpunkte vor Beginn eines Bedingungsblocks oder einer Schleife fehlen
<code>IndentationError</code>	tritt immer dann auf, wenn ein Block nicht ausreichend eingerückt ist
<code>ZeroDivisionError</code>	wird angezeigt, wenn versucht wird, durch 0 zu teilen
<code>IndexError</code>	tritt auf, wenn man versucht, auf eine nicht-existierende Indexposition innerhalb einer Sequenz zuzugreifen, häufig z.B., falls erwartete Programmargumente in <code>sys.argv</code> nicht existieren, da sie nicht auf der Kommandozeile angegeben wurden
<code>OSError</code>	zeigt an, dass ein Pfad/Dateiname nicht gefunden werden konnte oder dass Zugriffsberechtigungen fehlen

Tabelle 11: Wichtigste Fehlertypen

In den folgenden Übungen wollen wir die verschiedenen Optionen austesten, um Dateien zu öffnen und abzuarbeiten, selbstverständlich immer mit Fehlerbehandlung.

Übung 23 - Öffnen und Abarbeiten einer Datei, Methode `a`

Erstellen Sie eine neue Datei namens `11_datei_lesen_a.py`.

Kopieren Sie entweder eine einfache Textdatei in das Verzeichnis, wo Ihr Programm liegt, oder erstellen Sie eine neue mit UTF-8-Kodierung. Diese Datei sollte mindestens 10 Zeilen beinhalten.

Im ersten Programmschritt soll zunächst der Dateiname über die Kommandozeile als Argument entgegengenommen werden.

Versuchen Sie danach, die Datei mittels der zuerst beschriebenen Option zum Lesen und mit geeigneter Kodierung zu öffnen. Denken Sie dabei immer erst daran, was versuchen auf Englisch heißt ...

Falls beim Öffnen ein Fehler auftreten sollte, speichern Sie das Fehlerobjekt in einer Variable und beenden Sie das Programm innerhalb des Ausnahme-Blocks (das heißt des `except`-Blocks) mit der Anweisung `sys.exit()` und der Fehlervariable als deren Argument, welches Sie allerdings erst mittels der `str()`-Funktion in eine Zeichenkette umwandeln müssen.

Falls kein Fehler aufgetreten ist, lesen Sie die Zeilen der Datei mithilfe der `readlines()`-Methode des Dateiobjekts und speichern Sie diese in einer Liste.



Schreiben Sie dann eine `for`-Schleife, bei der links vom `in`-Operator ein Tupel mit einer Variable für die Zeilennummer und einer für die jeweilige Zeile steht und rechts davon die `enumerate()`-Funktion mit der Liste als Argument. Diese Funktion generiert ein Tupel aus einer fortlaufenden Nummer und dem jeweiligen Listenelement.

Geben Sie als Nächstes innerhalb der Schleife eine passende Meldung, die Zeilennummer und die jeweilige Zeile aus, wobei Sie die Nummer minimal erst mittels `str()` zu einer Zeichenkette umwandeln müssen.

Testen Sie das Programm mit der Datei, die Sie zuvor in das Verzeichnis kopiert hatten, aber schreiben Sie auch einmal absichtlich den Dateinamen falsch, um zu sehen, wie der Fehler gemeldet wird.

Falls nötig, nehmen Sie noch Schönheitskorrekturen vor.



Übung 24 - Öffnen und Abarbeiten einer Datei, Methode b

Speichern Sie das zuletzt erstellte Programm als `12_datei_lesen_b.py` ab und schreiben Sie es zunächst so um, dass Sie die Datei über `with ... as` geöffnet wird. Dabei soll innerhalb des `with`-Blocks diesmal die komplette Datei in eine Zeichenkette eingelesen werden.

Beachten Sie dabei, dass die Fehlerbehandlung in diesem Fall erst nach Ende des `with`-Blocks erfolgen kann, wobei die Datei danach automatisch geschlossen wird.

Ändern Sie dann die `for`-Schleife so, dass über die `splitlines()`-Methode der Kette eine Liste erzeugt wird, die dann wiederum über `enumerate()` das Tupel erzeugt.

Verwenden Sie hier zur Ausgabe auf jeden Fall einen f-String, bei dem die Zeilennummer auch passend ausgerichtet ist.



Übung 25 - Öffnen und Abarbeiten einer Datei, Methode c

Schreiben Sie das vorherige Programm nochmals um (als `13_datei_lesen_c.py`), wobei Sie innerhalb des `with`-Blocks, wiederum mit `enumerate()`, über die Datei iterieren, und bei der Ausgabe der Zeilenvariable die `strip()`-Methode auf diese anwenden.

Bisher haben wir schon mehrere Arten kennengelernt, um zur Eingabe auf Dateien zugreifen zu können. Im nächsten Abschnitt wollen wir jetzt aber auch besprechen, wie wir Ausgaben erzeugen.

5.1.4 In Dateien schreiben

Wie oben schon beschrieben, können beim nachlässigen Öffnen von Dateien unter Umständen schon bestehende Dateien überschrieben werden, so dass Sie hier immer ganz besonders aufpassen müssen. Falls Sie Programme erstellen, die bestehende Dateien manipulieren sollen, empfiehlt es sich ebenfalls, sobald wir einmal erlernt haben, mit Verzeichnissen in Python umzugehen, dass Sie Ihre Ausgaben zur Sicherheit nicht wieder in die zu verändernde Datei schreiben, sondern stattdessen eine modifizierte Kopie mit gleichem Dateinamen in einem anderen Verzeichnis erstellen.

Zum Schreiben in Dateien haben wir wieder verschiedene Optionen. Wir können entweder die `write()`-Methode des Dateiobjekts verwenden, um eine Datei als komplette Zeichenkette in eine Datei auszugeben, dies zeilenweise in einer Schleife tun, oder Zeilen schrittweise aus Variablen oder Ergebnissen von Anweisungen ‚zusammenbauen‘. Um mehrere Zeilen auf einmal als Liste auszugeben, können wir die `writelines()`-Methode anwenden. Bei beiden Methoden ist zu beachten, dass nicht automatisch ein Zeilenumbruch am Ende jeder Zeile eingefügt wird wie bei `print()`. Zeilenumbrüche müssen, falls nicht ohnehin vorhanden, manuell eingefügt werden, um nicht versehentlich mehrere Zeilen miteinander zu konkatenieren.

In den meisten Fällen werden Sie Dateien einlesen, deren Inhalte analysieren, und dann die Ergebnisse in eine andere Datei schreiben, aber ab und zu gibt es auch Gründe, Dateiinhalte zu Backup-Zwecken einfach zu duplizieren. Um besser verstehen zu können, wie das geht, aber auch welche Probleme dabei auftreten können, werden wir dies jetzt üben.

Übung 26 - Eine Datei kopieren

Schreiben Sie ein Programm `14_datei_kopieren.py`, in dem Sie eine Datei öffnen und in eine andere ausgeben.

Dabei sollten Sie die Dateinamen als Argumente auf der Kommandozeile angeben und beide Dateien (mit Komma dazwischen) innerhalb der `with`-Anweisung öffnen.

Im `with`-Block sollten Sie dann über die Eingabedatei, diesmal natürlich ohne Verwendung von `enumerate()`, iterieren und diese in die andere Datei ausgeben.

Testen Sie das Programm mit der Beispieldatei, die Sie für die vorherigen drei Programme verwendet haben. Legen Sie aber vorher noch einmal manuell eine Kopie davon an.

Beim ersten Testversuch verwenden Sie für die Ein- und Ausgabedatei zwei verschiedene Dateinamen, beim zweiten jedoch denselben. Was passiert dadurch und warum?



5.2 Mit Verzeichnissen und Pfaden arbeiten

Wie wir schon festgestellt hatten, ist der Zugriff auf das Programmverzeichnis, wie in den bisherigen Beispielen, meist unproblematisch zu erreichen. Für andere Operationen jedoch ist die Interaktion mit dem Dateisystem über verschiedene Module erforderlich. So müssen wir Verzeichnisinhalte lesen können, um herauszufinden, welche Dateien überhaupt existieren, zu welchem Typ sie gehören, oder um diese abzuarbeiten, falls sie bestimmte Kriterien erfüllen. Ebenso müssen wir manchmal Verzeichnisse erstellen, um darin Daten in veränderter Form abzuspeichern, anstatt sie zu überschreiben, oder Dateien oder Verzeichnisse auch vom Programm aus löschen können. Diese Operationen ähneln denen, die wir im ersten Kapitel schon auf der Kommandozeile geübt haben, nur dass wir dort keine Fehlerbehandlung benötigt haben, da wir ja immer wussten, wie die Verzeichnisstruktur aussieht und was die Verzeichnisnamen waren. Beim Umgang mit Pfaden in unseren Programmen ist jedoch eine Fehlerbehandlung unvermeidbar, da wir z.B. nicht immer wissen, ob wir überhaupt auf ein Verzeichnis zugreifen oder dieses erstellen dürfen. Im Folgenden wollen wir deshalb Teile von zwei weiteren Modulen besprechen, die uns den Zugriff auf Dateien, Verzeichnisse, und Pfade erleichtern.

5.2.1 Das `os`-Modul

Das `os`-Modul können wir, wie üblich, einfach über `import os` importieren. Wie der Name schon sagt, ist dieses Modul für die Interaktion mit dem Betriebssystem (engl. *operating system*) da. Leider setzt das Benutzen dieses Moduls manchmal die Verwendung des jeweils betriebssystemspezifischen Verzeichnistrenners voraus, das heißt also `/` für Linux/macOS und `\` für Windows. Letzteres ist nicht so komfortabel, da der `\` als Sonderzeichen problematisch ist und normalerweise umständlich maskiert werden muss. Um Verzeichnisnamen sicher zusammensetzen zu können, kann man den Verzeichnistrenner allerdings auch mithilfe des Befehls `os.sep` abfragen und dann bei der Konkatenation von Pfaden verwenden.

Trotz dieses Umstandes bietet das Modul einige nützliche Funktionen zum Umgang mit Verzeichnissen und zur Abfrage von Dateieigenschaften. Die Funktion `listdir()` z.B. liefert den Inhalt eines Verzeichnisses als Liste von Namen der Elemente innerhalb dieses Verzeichnisses zurück, wobei sie leider nicht zwischen Dateien, Verzeichnissen, oder symbolischen Verknüpfungen (auf die wir hier nicht näher eingehen wollen) unterscheidet. Oft reicht der Elementname jedoch schon aus, wenn man nur auf eine bestimmte Art von Dateien zugreifen will und diese sowieso anhand ihrer Extension erkennbar ist. Das wird z.B. oft bei der Arbeit mit Textdateien funktionieren, da diese per Konvention meistens die Extension `.txt` haben. Dies muss allerdings nicht zwingend der Fall sein, beispielsweise oft bei sogenannten README-Dateien, die bei Programmen mitgeliefert werden und erklären, wozu das Programm dient und/oder ob bei der Installation oder Ausführung spezielle Dinge beachtet werden müssen.

Falls wir tatsächlich mehr Informationen bezüglich des Dateityps benötigen, dann bietet sich dafür die `scandir()`-Funktion an, die einen Iterator über alle Verzeichniseinträge als Objekte zurückliefert und unter anderem Zugriff auf bestimmte Eigenschaften des Objekts, wie dessen Namen über `name`, als auch die Methoden `is_dir()` und `is_file()` zur Erkennung des Typs bietet. Beide oben besprochenen Funktionen liefern ohne Argument den Inhalt des aktuellen Verzeichnisses zurück, erwarten aber ansonsten den korrekt formatierten Pfad zu einem Verzeichnis.

Manchmal müssen wir aber auch mit Dateien arbeiten, die nicht im aktuellen Verzeichnis liegen. Dann ist es oft nötig, Pfad und Dateinamen aufzuspalten, z.B. um eine (veränderte) Kopie einer Datei mit gleichem Dateinamen in ein anderes Verzeichnis zu schreiben. Etwas ähnliches werden wir auch bald – allerdings mithilfe eines anderen Moduls – üben. Um mithilfe des `os`-Moduls den gesamten Pfad vor einem Dateinamen zu extrahieren, können wir die Funktion

```
os.path.split(Pfad)
```

des Untermoduls `path` verwenden, welche ein Tupel aus Pfad und Dateinamen zurückliefert. Um das Gegenteil zu erzielen, nämlich um Pfade ‚zusammenzubasteln‘, verwenden wir dann einfach

```
os.path.join(Argumentliste)
```

wobei die Argumentliste auch aus mehreren Pfadteilen bestehen kann. Der große Vorteil dieser Methode ist übrigens, dass hier automatisch der richtige Verzeichnistrenner zum Zusammenfügen des Pfades verwendet wird.

Übung 27 - Auslesen des aktuellen Verzeichnisses mit `scandir()`

Schreiben Sie ein neues Programm `15_verzeichnislesen.py`, mit dessen Hilfe Sie den Inhalt des aktuellen Programmverzeichnisses getrennt nach Verzeichnissen und Dateien ausgeben.

Legen Sie dazu nach dem Modulimport zunächst zwei Listen für die beiden Datentypen an.

Verwenden Sie dann `scandir()`, um in einer Schleife alle Verzeichniselemente abzuarbeiten und dabei, je nach Typ, das jeweilige Element in der richtigen Liste abzuspeichern.

Um die Elemente in die Listen einzufügen, verwenden Sie jeweils die `append()`-Methode der Liste.



Geben Sie danach, nach geeigneten Meldungen über den Typ, die Inhalte der zwei Listen wieder aus, wobei Sie aber nur die Namen der Elemente ausgeben sollten. Für die `print()`-Anweisungen, die den Typ anzeigen, können Sie zur besseren Formatierung das `end`-Argument dieser Funktion so angeben, dass zwei Zeilenumbrüche auf die Meldung folgen. Das Schlüsselwort-Argument `end` wird dabei als Argument-Wert-Kombination angegeben, z.B. `end='\t'` für einen Tabulator.

5.2.2 Das `Path`-Objekt des `pathlib`-Moduls

Für komplexere Operationen mit Dateipfaden ist ein anderes Objekt aus einem weiteren Modul besser geeignet, da es uns unter anderem erlaubt, Pfade, inklusive von Dateien, auch unter Windows sicher mit `/` anzugeben. So müssen Sie nur noch einen Verzeichnistrenner verwenden und dabei auch nicht zwischen den verschiedenen Betriebssystemen unterscheiden. Dieses Objekt namens `Path` ist im `pathlib`-Modul definiert und kann über die Anweisung

```
from pathlib import Path
```

importiert werden. Diese Form des Imports ist für uns neu, da wir ja bisher immer komplette Module importiert haben, jetzt aber an diesem Beispiel sehen können, dass man bei Bedarf auch ganz einfach nur Teile von Modulen zu unserem Programm hinzufügen kann.

Einen neuen Pfad anzulegen, geht mit diesem Untermodul einfach über die Erzeugung eines neuen Objekts, z.B.

```
pfad = Path('./test')
```

welches dann auf das Unterverzeichnis namens `test` im aktuellen Verzeichnis verweist. Neue Pfade können aber ebenso einfach aus alten generiert werden, z.B.

```
neuesVerz = pfad / 'test'
```

wo dann der Schrägstrich quasi eine Konkatenation, aber gleich mit dem richtigen Verzeichnistrenner, durchführt. Auf diese Weise erzeugte Pfade können wiederum auch mit den Funktionen des `os`-Moduls, nach Umwandlung zu einer Zeichenkette, verwendet werden.

Das `Path`-Objekt bietet viele nützliche Methoden, von denen wir aber hier nur die für uns interessantesten besprechen. Die Methoden `cwd()` und `home()` liefern jeweils das aktuelle und das Benutzerverzeichnis zurück, während `is_file()`, `is_dir()` und `name` genauso wie im `os`-Modul funktionieren. Die Methode `exists()` überprüft, wie ihr Name schon besagt, ob eine Datei oder ein Verzeichnis besteht, während `makedirs()` ein neues Verzeichnis erzeugt, vorausgesetzt, dass wir die nötigen Berechtigungen dazu haben.

Die Anwendung der Methode `rglob(Muster)` ist besonders interessant, da diese eine Liste aller Dateien, auf die ein Muster zutrifft, rekursiv, das heißt inklusive aller möglichen Unterverzeichnisse, erstellt. So können Sie mit

```
pythonDateien = pfad.rglob('*.py*')
```

eine Liste aller Python-Dateien in Ihrem Übungsverzeichnis generieren, selbst wenn einige dieser Dateien in Unterverzeichnissen liegen sollten. Dabei werden jedoch nicht nur Dateinamen, sondern u.U. auch komplette Pfade zurückgeliefert.

Übung 28 - Ein neues Verzeichnis erstellen

Schreiben Sie das Programm `16_verzeichnis_erstellen.py`, in dem Sie im aktuellen Programmverzeichnis ein Unterverzeichnis namens `backup` erstellen und dort eine Datei hineinkopieren.

Importieren Sie dafür das `Path`-Objekt aus dem `pathlib`-Modul und lesen Sie dann wieder einen Dateinamen über die Kommandozeile ein.

Speichern Sie dann den aktuellen Programmpfad mithilfe der `cwd()`-Methode des `Path`-Objekts in einer passenden Variable.

Erstellen Sie danach einen neuen Pfad für das `backup`-Verzeichnis und speichern ihn in einer anderen Variable.

Erstellen Sie jetzt einen neuen Pfad für die Ausgabedatei aus dem Pfad für das `backup`-Verzeichnis und dem Namen der Eingabedatei.

Testen Sie dann mithilfe der `exists()`-Methode über die vorletzte Pfadvariable, ob das `backup`-Verzeichnis schon existiert oder nicht.

- Falls nicht, generieren Sie mit der `mkdir()`-Methode des `Path`-Objektes das neue Verzeichnis und geben Sie eine entsprechende Meldung aus.
- Falls ja, geben Sie nur eine geeignete Meldung aus.

Kopieren Sie zuletzt, wie im vorherigen Kopierprogramm, wieder die Datei, aber diesmal mithilfe des neuen Pfades für die Ausgabedatei, wobei Sie das `Path`-Objekt für die Datei erst zu einer Zeichenkette wandeln müssen.

Jetzt, wo wir wissen, wie wir mit Dateien und über Verzeichnisse auf unsere Analyse-
daten zugreifen bzw. deren Ergebnisse ausgeben können, können wir uns im nächsten
Kapitel etwas intensiver mit der Verarbeitung von Sprachmustern beschäftigen.





5.3 Lösungen zu den Aufgaben

Lösung 23 – Öffnen und Abarbeiten einer Datei, Methode a

Eine Datei als Argument auf der Kommandozeile zu übergeben, dürfte Ihnen mittlerweile nicht mehr schwerfallen, da wir die Übergabe von Kommandozeilenargumenten schon ausreichend geübt haben. Beim Erstellen des Pfades für das Öffnen der Datei haben Sie hoffentlich nicht vergessen, die relative Pfadangabe, so wie oben besprochen, zu generieren.

Der Versuch, die Datei zu öffnen, sollte innerhalb eines `try`-Blocks erfolgen, also:

```
try:
    datei = open('./' + dateiName, 'r', encoding='utf-8')
```

Beachten Sie hier wieder die Erzeugung der relativen Pfadangabe mittels Konkatenation. In meinem Beispiel sehen Sie auch, dass statt des Schlüsselwortarguments `mode` mit einem Wert auch nur der Wert angegeben werden kann, insofern er sich an der richtigen Position, das heißt direkt hinter dem Dateinamen befindet. Der `except`-Block sollte dann in etwa wie folgt aussehen, wobei wir den Fehler einfach in der sprechenden Variable namens `fehler` abspeichern, insofern er überhaupt aufgetreten ist.

```
except OSError as fehler:
    sys.exit(str(fehler))
```

Die `exit()`-Funktion des `sys`-Moduls, die wir im `except`-Block aufrufen, sorgt dafür, dass das Programm bei einem Fehlerauftritt ordnungsgemäß beendet wird. Selbst wenn wir nicht auf ein Kommandozeilenargument zugreifen würden, müssten wir also dieses Modul importieren, um diese Funktion zu verwenden. Wenn als Argument an `exit()` eine Zeichenkette übergeben wird, wird diese als Meldung auf der Kommandozeile ausgegeben. Allerdings ist das Fehlerobjekt keine Zeichenkette, sondern besteht, wie wir später noch lernen werden, aus mehreren Teilen, weshalb es erst in eine Zeichenkette gewandelt werden muss, um eine sinnvolle Ausgabe zu erreichen, auch, da sonst wieder ein Fehler auftritt, den wir aber nirgendwo in unserem Programm behandeln.

Das Einlesen der Zeilen in der Datei erfolgt durch

```
Zeilen = datei.readlines()
```

außerhalb der beiden Blocks, wodurch wir die Datei schon komplett abgearbeitet und in der `Zeilen`-Liste abgespeichert haben. Da wir jedoch die `open()`-Methode ohne Kontextualisierung durch `with ... as` verwendet haben, müssen wir die Datei danach auch – eigentlich – mit `datei.close()` schließen. Allerdings tritt kein Fehler auf, falls Sie das nicht tun, und die Datei wird bei unserem kurzen Programm sowieso am Ende des Programms automatisch geschlossen. In einem längeren Programm jedoch, wo unter Umständen mehrfach derselbe Variablenname zum Öffnen verschiedener

Dateien verwendet wird, würde dies tatsächlich einen (wieder nicht behandelten) Fehler hervorrufen. Dann würde Python ‚merken‘, dass Sie eine schon geöffnete Datei erneut öffnen wollen, was aber nicht zulässig ist.

Weil wir bisher nur `for`-Schleifen verwendet haben, bei denen links vom `in`-Operator eine einzelne Variable stand, ist diese Schleife wahrscheinlich erst einmal sehr ungewöhnlich zu schreiben: Jetzt tritt erst ein Tupel links auf und dann noch eine andere Funktion – genau genommen sogar eine Klasse (mehr dazu in Abschnitt 7.6) –, die die Schleife als Argument übernimmt. Der Schleifenkopf sollte dann in etwa so aussehen:

```
for (num, zeile) in enumerate(Zeilen):
```

Innerhalb der Schleife sollten Sie nicht vergessen, vor der `print()`-Anweisung erst einmal jeweils den Zeilenumbruch am Ende der auszugebenden Zeile mittels `strip()` zu löschen, da ja `print()` selbst standardmäßig einen Zeilenumbruch anfügt und sie sonst zwei Umbrüche ausgeben würden. Bei der Ausgabe beachten Sie bitte zweierlei: Erstens muss die Zeilennummer, die durch `enumerate()` generiert wird, auch zu einer Zeichenkette gewandelt werden; zweitens muss, wie bei allen Sequenzen in Python, der Anfangsindex bei 0 liegen, weshalb Sie für eine echte Nummerierung jeweils immer 1 addieren müssen. Die Anweisung sollte dann wie folgt aussehen.

```
print('Zeile Nummer', str(num+1) + ': ', zeile)
```

Noch eleganter könnten Sie allerdings auch einen f-String verwenden und

```
print(f'Zeile Nummer {num+1}: {zeile}')
```

schreiben.

Die Nummerierung von Zeilen in einer Ausgabedatei mag auf den ersten Blick eine etwas seltsame Übung darstellen. Aber neben dem Ziel, zu üben, wie man Dateien öffnen und abarbeiten kann, hat dies tatsächlich auch einen Nutzen, und zwar falls man wirklich mal ein Dokument produzieren muss, bei dem man sich auf die einzelnen Zeilen beziehen kann, um Kommentare darüber abzugeben oder diese zu zitieren. Zum Beispiel könnten Sie, falls sie einmal keinen vernünftigen Editor haben, der automatisch Zeilennummern für Sie ausdruckt, damit ganz einfach eine geeignete Fassung eines Programms mit Zeilennummern zum Ausdrucken produzieren.

Lösung 24 - Öffnen und Abarbeiten einer Datei, Methode b

Der Anfang dieses Programms bleibt gleich, der größte Unterschied zum vorherigen Programm ist, dass das Öffnen und Einlesen der Datei beides innerhalb des `try`-Blocks geschehen muss. Dieser bildet nämlich eine Einheit an deren Ende die Datei automatisch geschlossen wird, so dass die `try`- und `except`-Blöcke jetzt ungefähr so aussehen sollten:

```
try:
    with open('./' + dateiName, 'r', encoding='utf-8') as
        datei:
            dateiInhalt = datei.read()
except OSError as fehler:
    sys.exit(str(fehler))
```

Das Einlesen des Dateiinhalts ist im Prinzip nur leicht anders, da wir diesmal mit der `read()`-Methode nicht die Zeilen als Liste einlesen, sondern die komplette Datei. Auch die `for`-Schleife bleibt fast gleich, nur dass wir innerhalb des Aufrufs von `enumerate()` jetzt nicht direkt eine Liste übergeben, sondern diese erst mithilfe von `dateiInhalt.splitlines()` dynamisch generieren. Eine Besonderheit der `splitlines()`-Methode ist jedoch, dass sie standardmäßig alle Zeilenenden entfernt, so dass wir diese nicht extra vor der Ausgabe entfernen müssen, und somit unser Programm um zwei Zeilen kürzer geworden ist, da wir ja die Datei nicht manuell schließen mussten. Die `for`-Schleife könnte jetzt also folgendermaßen aussehen.

```
for (num,zeile) in enumerate(dateiInhalt.splitlines()):
    print(f'Zeile Nummer {num+1:>2d}: {zeile}')
```

Beachten Sie, dass innerhalb des f-Strings die Zeilennummer rechtsbündig ausgegeben wird und auf zwei Stellen festgelegt ist. Da unsere Beispieldatei nur sehr kurz ist, funktioniert das bei dieser Festlegung gut und die Ausgabe sieht ordentlicher aus, als wenn die Zahlen linksbündig angeordnet wären, da dann alle einstelligen Zahlen nach links verschoben erscheinen würden. Bei längeren Dateien müssten Sie allerdings idealerweise die Länge der Zeilennummer für die korrekte Ausrichtung erst einmal im Programm bestimmen. Dafür ist das Arbeiten mit `readlines()` besser geeignet, weil in unserem Fall nicht ermittelt werden kann, was die Länge der höchsten Zeilennummer ist. Alternativ könnten Sie auch eine Länge schätzen, da sie ja wahrscheinlich selten riesige Dateien abarbeiten werden.

Lösung 25 - Öffnen und Abarbeiten einer Datei, Methode c

Dieses Programm ist noch eine Zeile kürzer, da wir diesmal im `with`-Block auch gleichzeitig über das Dateiojekt (was ja mehr oder weniger nichts anderes als eine Liste von Zeilen darstellt) direkt innerhalb des Aufrufs von `enumerate()` und der `for`-Schleife iterieren können. Allerdings bekommen wir dabei normalerweise wieder die einzelnen Zeilen mit Zeilenumbruch zurückgeliefert, so dass wir diese bei der Ausgabe der Zeilenvariable mittels `strip()` entfernen müssen. Dadurch läuft dann fast das gesamte Programm, bis auf die `import`-Anweisung, die Zuweisung des Kommandozeilenarguments an die Variable, die den Dateinamen beinhaltet, und die Fehlerbehandlung, innerhalb des `try`-Blocks.

Die `strip()`-Methode hätten wir übrigens auch schon in `11_datei_lesen_a.py` anwenden können, um uns eine Zeile zu sparen, aber da dort der Fokus auf der Fehlerbehandlung lag, wollte ich diese Übung nicht komplizierter machen als nötig. Und selbstverständlich könnten wir auch das Programm, welches wir in dieser Übung erstellt haben, noch einmal um eine Zeile verkürzen, und zwar dadurch, dass wir das Kommandozeilenargument nicht erst in einer Variable speichern, sondern es direkt in der `open()`-Methode verwenden. Diese beiden Dinge können Sie aber gerne selbst austesten.

Lösung 26 - Eine Datei kopieren

Die meisten Konstrukte, die in dieser Übung angewendet werden, haben wir schon geübt. Beim Verwenden der Kommandozeilenargumente müssen Sie entscheiden, ob Sie die Dateinamen in Variablen speichern oder direkt über die `sys.argv`-Listenelemente beim Öffnen zugreifen wollen, um sich eine Programmzeile zu sparen. Falls Sie die Dateinamen lieber explizit anlegen, dann sollten Sie aus Effizienzgründen beide Argumente gleichzeitig aus `sys.argv` holen, was Sie mithilfe von

```
(eingabeDatei, ausgabeDatei) = sys.argv[1:]
```

leicht erreichen.

Beim Öffnen der Dateien mit der `with`-Anweisung ist es wahrscheinlich noch etwas ungewohnt für Sie, zweimal hintereinander eine `open()`-Anweisung, nur durch Komma getrennt, auszuführen und auch ebenfalls zweimal `as` zu verwenden, da wir ja auf zwei Dateiobjekte zugreifen wollen. Wichtig ist hier auch, dass die erste Datei nur im Eingabe- und die zweite im Ausgabemodus geöffnet werden sollte. Zum Iterieren über die Eingabedatei benötigen Sie nur eine einfache `for`-Schleife über das Dateiobjekt. Und da wir die Zeilen der Eingabedatei exakt so ausgeben wollen, wie sie ursprünglich sind, nämlich inklusive der Zeilenumbrüche, müssen wir bei der Ausgabe über

```
Ausgabedateiobjekt.write(Zeichenkette)
```

auch keine zusätzlichen Umbrüche mit einfügen. Deshalb muss als Argument nur die Zeile selbst angegeben werden, und Sie müssen anstelle des Ausgabedateiobjekts natürlich den von Ihnen gewählten Objektvariablennamen angeben.

Nach dem zweiten Testdurchlauf sollten Sie feststellen, dass die Eingabedatei jetzt leer ist, da Python unglücklicherweise zwei Dateiobjekte über den gleichen Dateinamen angelegt hat, wobei das erste ganz normal zum Lesen geöffnet wurde, das zweite aber, ohne Ihnen einen Fehler zu melden, für den gleichen Dateinamen zum Schreiben. Deshalb wurde die Eingabedatei schon vor dem eigentlichen Auslesen neu angelegt und somit überschrieben. Daher steht beim Iterieren über das Dateiobjekt auch kein Inhalt

mehr zur Verfügung und sowohl Ein- als auch Ausgabedatei sind jetzt dummerweise leer. Dies verdeutlicht Ihnen hoffentlich, wie wichtig es ist, sich beim Öffnen zum Schreiben immer darüber bewusst zu sein, ob man schon bestehende Ergebnisse überschreiben will und darf. Außerdem sehen wir dadurch, dass unser Programm in dieser Form eigentlich nicht als Kopierprogramm geeignet ist, da dieser potenzielle Fehler nicht behandelt wird. Allerdings ließe sich dieses Problem leicht beheben, indem Sie das Programm so umschreiben, dass direkt nach dem Programmstart darauf geprüft wird, ob die beiden Kommandozeilenargumente gleich sind, und falls ja, das Programm mit einer Fehlermeldung abgebrochen wird. Hierbei müssen Sie noch nicht einmal mit `try-except` arbeiten, da eine einfache `if`-Anweisung dafür ausreicht und Sie ja schon wissen, wie man ein Programm ordnungsgemäß abbricht.

Lösung 27 - Auslesen des aktuellen Verzeichnisses mit `scandir()`

Als erstes müssen Sie das Modul über `import os` laden, damit die `scandir()`-Funktion überhaupt zur Verfügung steht. Die beiden leeren Listen für die Dateitypen sollten natürlich sinnvollerweise `dateien` und `verzeichnisse` heißen, und Sie können Sie entweder mittels `[]` oder über die `list()`-Funktion anlegen.

Da `scandir()` einen Iterator liefert, können Sie über `os.scandir()` ganz einfach eine `for`-Schleife laufen lassen, die die einzelnen Elemente in dem Verzeichnis zurückliefert. Sinnvollerweise verwenden Sie deshalb als Schleifenvariable `element` und zwei Bedingungen, die dieses Objekt auf `is_file()` bzw. `is_dir()` prüfen, und hängen Sie den Inhalt dieser Variable entsprechend des Ergebnisses an die jeweilig passende Liste an. Das sollte Ihnen mittlerweile nicht schwer fallen.

Bei den `print()`-Anweisungen, die den jeweiligen Ausgaben der zwei Listen vor- ausgehen sollten, ist das einzig Neue, dass Sie anstelle eines einzelnen Zeilenumbruchs jetzt zwei erzeugen sollten, indem Sie das Ende der Ausgabe mittels des entsprechenden `end`-Schlüsselwortarguments mit `'\n\n'` initialisieren und somit die Standardeinstellung verändern.

Zur Iteration über die einzelnen Elemente der zwei Listen können Sie wieder dieselbe Schleifenvariable verwenden, wobei Sie auf die Eigenschaft `name` des jeweiligen Elements über `element.name` zugreifen. Dies ist vielleicht zuerst etwas ungewohnt, da wir ja bisher normalerweise Methoden von Objekten aufgerufen haben, bei denen natürlich dann die runden Klammern für die optionalen Argumente folgen müssen. Über Eigenschaften werden wir in Kapitel 7 noch mehr erfahren, und auch lernen, wie Sie sie selbst erzeugen können.

Lösung 28 - Ein neues Verzeichnis erstellen

Dieses Programm ist jetzt schon etwas komplexer, da wir nicht nur mehrere Importe haben, sondern auch mehrere Pfade generieren, dann testen müssen, ob einer dieser Pfade schon existiert, ihn erzeugen falls nicht, und zuletzt noch eine Datei in das neu angelegte Verzeichnis kopieren müssen, wobei auch noch etwaige Fehler abgefangen

werden sollten. Was Sie allerdings in diesem Fall nicht überprüfen müssen, ist, ob sie eventuell eine Datei überschreiben könnten, da Sie ja ganz bewusst denselben Dateinamen zum Schreiben in ein anderes Verzeichnis verwenden wollen.

Allerdings vereinfacht die Anwendung des `Path`-Objekts unsere Arbeit schon erheblich, da wir zum Anlegen der ersten drei Pfade, nachdem Sie den Namen der Eingabedatei in einer Variable abgespeichert haben, nur die folgenden Anweisungen benötigen.

```
pfad = Path.cwd()
neuesVerz = pfad / 'backup'
ausgabeDatei = neuesVerz / eingabeDatei
```

Beim Überprüfen, ob der Pfad für das Backup-Verzeichnis existiert, müssen Sie mit dem `not`-Operator arbeiten, da nur die Methode `exists()` tatsächlich vom `Path`-Objekt zur Verfügung gestellt wird, aber nicht eine Methode, die auf das Gegenteil prüft, was wir ja in diesem Fall tun wollen. Bei der Ausgabe der jeweiligen Meldungen muss natürlich auch die Pfad-Variable, die schließlich ein Objekt beinhaltet, wieder mit `str()` in eine Zeichenkette gewandelt werden, da sonst ein Fehler auftritt.

Bei der eigentlichen Kopieroperation müssen Sie allerdings die Umwandlung mittels `str()` nur für den Ausgabepfad verwenden, da Sie ja den Eingabepfad einfach mittels der altbewährten Konkatenation einfach wieder aus dem Dateinamen erzeugen können, auch wenn hier die sauberere Methode theoretisch die Anwendung des `Path`-Objekts wäre.

6 Sprachmuster erkennen und bearbeiten



Sprache besteht aus einer Vielzahl von Mustern, von Flexionsformen hin bis zu längeren phraseologischen Einheiten. In diesem Kapitel erlernen Sie, wie man solche Muster mithilfe sogenannter regulärer Ausdrücke auffinden und bearbeiten kann, und dabei wiederum potenziell auftretende Fehler behandeln.

6.1 Reguläre Ausdrücke

Wie wir vorher schon gesehen hatten, haben die Standardmethoden, die für Zeichenketten existieren, den wesentlichen Nachteil, dass man nur mit festen Mustern arbeiten kann, um diese Ketten zu verändern oder bestimmte Teile zu extrahieren. **Reguläre Ausdrücke** jedoch bieten uns Möglichkeiten, komplexere Muster zu definieren und zu suchen, um damit Daten zu bereinigen oder bearbeiten. Da der Begriff regulärer Ausdruck etwas schwer von der Zunge geht, wird er auch häufig zu **Regex** (von engl. **regular expression**) abgekürzt, oder Regexes für den Plural. Weil Regexes nicht einfach zu erlernen sind, werden wir am Anfang zunächst mit sehr einfachen Mustern arbeiten, dann aber schrittweise die Komplexität erhöhen, je mehr Grundbegriffe wir kennengelernt haben und diese miteinander verknüpfen können.

Zur Verwendung von Regexes in Python – ohne Installation eines Zusatzmoduls – kann man das `re`-Modul importieren. Dort sind diese Ausdrücke als Objekt definiert, weshalb wieder über verschiedene Methoden auf sie zugegriffen wird. Wir listen hier zunächst die wichtigsten und nützlichsten Methoden auf und besprechen ihre Funktionen jeweils kurz. Danach beschäftigen wir uns konkret damit, wie die Objekte, die unsere Suchergebnisse beinhalten, aufgebaut sind und man auf bestimmte Teile davon zugreifen kann, und erlernen nach und nach die einzelnen Konstrukte, die uns erlauben, immer komplexere Suchmuster zu entwerfen.

Methode	Funktionalität
<code>search(Muster, Kette)</code>	sucht nach dem (ersten) Auftreten des Musters in einer Zeichenkette
<code>match(Muster, Kette)</code>	sucht ebenfalls nach dem Muster, aber nur am Anfang einer Zeichenkette, ist also eingeschränkter in seiner Funktion
<code>findall(Muster, Kette)</code>	liefert alle gefundenen Vorkommen als Liste von Zeichenketten zurück

<code>finditer(Muster, Kette)</code>	liefert einen Iterator über alle gefundenen Match-Objekte zurück, so dass einzeln in einer Schleife auf die Suchergebnisse zugegriffen werden kann
<code>sub(Muster, Ersetzung, Kette)</code>	ersetzt alle vorkommen eines Musters in einer Zeichenkette, wobei zur Ersetzung auch Funktionen verwendet werden können
<code>split(Muster, Kette)</code>	liefert eine Liste, ähnlich der gleichnamigen Zeichenkettenmethode, zurück, ist aber durch das Muster wesentlich flexibler

Tabelle 12: Wichtigste Methoden der `re`-Objekts

Die in Tabelle 12 genannten Methoden haben oft zusätzliche Argumente, z.B. zu ihrer (positionellen) Einschränkung, was aber hier nicht besprochen wird. Um die Ersetzung bei der Verwendung von `re.sub()` zu beschränken, kann das optionale Schlüsselwortargument `count` verwendet werden. Dies kann manchmal sehr wichtig sein, wenn z.B. immer nur das aktuell gefundene Muster gezielt durch eine bestimmte Zeichenkette verändert werden soll, bei der ein Teil variabel sein soll, z.B. bei der automatischen Nummerierung der Attribute von XML-Elementen, über die wir in Abschnitt 10.7 mehr erfahren werden.

6.2 Allgemeine Suchsyntax

Um mit dem `re`-Modul zu arbeiten, haben wir zwei Optionen, einmal die direkte Verwendung des Moduls, was bei einmaliger Anwendung kürzer und kompakter ist, oder die Verwendung eines vorkompilierten `re`-Objekts und dessen Methodenaufruf über die Objektvariable. Letzteres ist insbesondere für die wiederholte Anwendung nützlich. Ein Beispiel für die allgemeine Syntax der ersten Verwendungsoption ist:

```
ergebnis = re.search(Suchmuster, Suchziel, [Kompilierungsflags])
```

In diesem Fall wird die `search()`-Methode des Objekts, die wir gleich besprechen werden, direkt mit einem Muster auf eine Zeichenkette angewendet, die das Suchziel darstellt, wobei wir zwei Argumente angeben müssen. Bei der zweiten Option wird im ersten Schritt zunächst das Muster kompiliert und dann die `search()`-Methode auf dieses vorkompilierte Muster mit dem Suchziel als einzigem Argument angewandt.


```
muster = re.compile(Suchmuster, [Kompilierungsflags])
ergebnis = muster.search(Suchziel)
```

Mehr zu Kompilierungsflags werden wir in Abschnitt 6.10 besprechen, wenn wir Regexes selbst besser verstehen. Trifft das Suchmuster auf den jeweiligen Ausdruck zu, wird ein Treffer-Objekt (engl. *match object*) als Ergebnis zurückgeliefert, ansonsten der Wert `None`. Letzteres ermöglicht einen einfachen Test auf einen Treffer durch

```
if ergebnis:
```

da der Wert `None` immer `False` ist und dementsprechend bei einem `True`-Ergebnis immer ein Treffer vorliegen muss.

6.3 Mit dem Match-Objekt arbeiten

Das Treffer-Objekt beinhaltet – zumindest theoretisch – eine Liste von Treffergruppen. Mehr dazu können wir erst später besprechen, wenn wir etwas über Gruppierungen gelernt haben. Dabei steht der komplette Treffer bei Index 0, was die (optionale) Voreinstellung ist, falls nicht ein Index bei der Trefferabfrage angegeben wird. Mögliche Untergruppen eines Treffers erscheinen dann ab Indexposition 1. Um auf die einzelnen Informationen zu Treffern zuzugreifen, gibt es mehrere Methoden.

Methode	Funktionalität
<code>group (Index/Name)</code>	liefert die gefundene Ergebniskette oder eine Untergruppe des Musters zurück
<code>start (Index/Name)</code>	liefert den Anfangsindex des Treffers/der Gruppe innerhalb der Kette zurück
<code>end (Index/Name)</code>	liefert den Endindex des Treffers/der Gruppe zurück
<code>span (Index/Name)</code>	liefert ein Tupel aus Start und Ende zurück, vereint also <code>start ()</code> und <code>end ()</code>

Tabelle 13: Methoden des Trefferobjekts

Mehr zu Namen besprechen wir ebenfalls später, wenn wir mehr über Gruppierungen wissen. Zunächst wollen wir üben, ganz einfache Muster zu finden.



Übung 29 - Einfache sequenzielle Muster erkennen

Schreiben Sie ein neues Programm (`17_einfache_muster.py`), welches in der Datei `beispiel_saetze.txt` nach einer als Programmargument übergebenen Zeichenkette sucht und dessen Vorkommen in der Ausgabe innerhalb der jeweiligen Trefferzeile durch eckige Klammern markiert anzeigt.

Verwenden Sie dazu die erste oben besprochene Option der Verwendung des `re`-Objekts mit der `search()`-Methode, um ein mögliches Ergebnis zu finden.

Falls auf der Zeile ein Ergebnis vorliegt, verwenden Sie die `span()`-Methode auf dieses Ergebnis, um zunächst die relevanten Indexpositionen zu ermitteln und in zwei geeigneten Variablen abzuspeichern.

Schreiben Sie dann eine `print()`-Anweisung, in der Sie den Inhalt der Zeile vor dem Treffer, `' ['`, den Treffer, `'] '`, und den Inhalt der Zeile nach dem Treffer ausgeben.

Tipp: mit Slicing arbeiten. Zur Unterdrückung des Zeilenumbruchs bei der Ausgabe sollten Sie dabei das Schlüsselwort-Argument `end` mit einer leeren Zeichenkette verwenden.

Testen Sie dann das Programm mit den Ketten `in`, `aus` und `man`.

Was können Sie hier bei genauer Beobachtung erkennen?

Falls Sie schon etwas Ahnung von Korpuslinguistik haben, versuchen Sie auch die Frage zu beantworten, was für eine Art von Programm Sie hiermit in einfachster Form entwickelt haben.

In den folgenden Unterabschnitten werden wir jetzt Schritt für Schritt die wichtigsten Teilkonzepte der Regexes kennenlernen und dabei herausfinden, wie wir unsere Suchen analog dazu optimieren können. Dabei werden auch die Beispiele, die wir behandeln und die anfangs noch relative abstrakt sein werden, nach und nach immer mehr Sinn ergeben, je mehr dieser Konzepte wir miteinander verknüpfen können.

6.4 Zeichenklassen

Zeichenklassen stellen quasi die einfachste Ebene von Regexes dar. Sie sind flexible Gruppierungen von Zeichen, die es ermöglichen, einfache Alternativen für das Auftreten einzelner Zeichen in einem Muster anzugeben. Um nützlicher zu sein, müssen sie allerdings normalerweise quantifiziert werden, worüber wir im nächsten Abschnitt mehr erfahren werden.

Die Alternativen in einer Zeichenklasse werden in eckigen Klammern aufgelistet (`[...]`). So zum Beispiel findet die Suche nach der Zeichenklasse `[ae]` alle Wörter oder Zeichenketten, die entweder die Kleinbuchstaben `<a>` oder `<e>` beinhalten, oder auch beide, z.B. *ja*, *an*, *etwas*, *gearbeitet*, oder *Knabe*, was hier durch die Unterstreichung hervorgehoben ist. Solche Klassen können auch als Bereiche, markiert durch Bindestrich(e), oder

Abkürzungen (engl. *shorthands*) ausgedrückt werden. So findet die Klasse mit dem Bereich `[0-9]` Zeichenketten, die Zahlen beinhalten, die Abkürzung `\w` Wortzeichen (**allerdings leider ohne Bindestriche!**), `\W` nicht-Wortzeichen, `\s` alle Formen von Leerzeichen (inklusive Tabulatoren und Zeilenumbrüchen) und `.` ein beliebiges Zeichen.

Zeichenklassen können auch verneint werden, indem man ein Zirkumflex hinter der öffnenden Klammer einfügt, oder, wie wir oben bei `\W` gesehen haben, deren Abkürzungen groß schreibt. Beispielsweise findet die Klasse `^[0-9]` alles, was nicht eine Ziffer ist. Allerdings bedeutet eine derartige Negation nicht immer auch tatsächlich das Gegenteil von dem, was man vielleicht erwarten würde. So könnte man meinen, dass die negierte Zeichenklasse `^[A-ZÄÖÜ]`, die im Prinzip das ‚Gegenteil‘ zu den Großbuchstaben darzustellen scheint, nur alle Kleinbuchstaben findet, was aber leider ein Trugschluss wäre. Warum, und was damit genau gefunden wird, können Sie selbst testen, sobald wir die nächste Übung beendet haben.

Im Folgenden wollen wir jetzt ein Programm schreiben, welches uns erlauben wird, nicht nur flexible Zeichenklassen auszutesten, sondern dabei auch einige Probleme zu beheben, die wir im letzten Programm erkannt haben. Allerdings wird es dadurch, und weil wir hier einige der schon vorher erlernten Konzepte zusammenfügen, etwas komplexer, so dass Sie sich etwas mehr Zeit dafür lassen und Ihr Programm auch etwas ausführlicher kommentieren sollten, als Sie es vielleicht sonst täten. Bitte beachten Sie auch, dass wir in diesem Programm noch keine Fehlerbehandlung für fehlerhaft eingegebene reguläre Ausdrücke durchführen werden.

Übung 30 - Zeichenklassen testen

Schreiben Sie das Programm `18_zeichenklassen_testen.py`, mit dem Sie das tun können, was der Dateiname besagt.

Nehmen Sie zunächst eine Zeichenklasse als Argument entgegen.

Um sicherzustellen, dass auch wirklich ein Argument übergeben wurde und eine Klasse im Suchbegriff vorkommt, schreiben Sie einen `try` und einen `except`-Block.

Im `try`-Block

- speichern Sie zunächst den Suchbegriff in einer Variable ab,
- und überprüfen dann in einer `if`-Anweisung mittels des `not`-Operators und einer `re`-Suche mit der Zeichenklasse `[\.\[\]\]`, ob im Suchbegriff keine möglichen Zeichenklassen vorkommen. Den Suchbegriff für den Test werden wir später noch im Detail besprechen.
 - Falls die Überprüfung `True` zurückliefern sollte, beenden Sie das Programm mit einer passenden Nachricht.

Falls im `except`-Block ein `IndexError` abgefangen wird, beenden Sie das Programm innerhalb dieses Blocks ebenfalls mit einer geeigneten Nachricht, da dies bedeutet, dass kein Argument an das Programm übergeben wurde.



Arbeiten Sie jetzt wieder (mit Fehlerbehandlung) die schon vorher verwendete Beispieldatei zeilenweise ab und prüfen Sie zunächst mittels einer `re`-Suche auf der Zeile, ob der Suchbegriff überhaupt gefunden wurde.

Falls ja, legen Sie eine leere Zeichenkettenvariable und eine Variable für die Startposition des Treffers an, die später zum Zusammensetzen der Ergebnisse dienen werden.

Verwenden Sie dann die `finditer()`-Methode, um eine Schleife über alle Treffer-Objekte auf der jeweiligen Zeile laufen zu lassen.

Bestimmen Sie jeweils die Anfangs- und Endpositionen des Treffers und speichern Sie diese in geeigneten Variablen ab.

Fügen Sie danach alles ab dem aktuellen Startpunkt bis zum Anfang des Treffers, eine öffnende eckige Klammer, den Treffer, mittels der `group()`-Methode und eine schließende eckige Klammer an die neue Zeichenkette an.

Weisen Sie nun der Startvariable die Endposition des Treffers zu, da sie den potenziellen Anfangspunkt für das nächste Ergebnis darstellt.

Geben Sie zuletzt die neu zusammengefügte Zeile mit den Markierungen und den Rest der ursprünglichen Zeile ab der letzten Endposition, aber ohne den Zeilenumbruch, aus.

Atmen Sie tief durch und testen Sie das Programm ohne Argument, mit den Ketten `[Ss]e`, `[Mm]an`, `seh[er]`, `[^\w]` sowie anderen beliebigen Beispielen.

Wie Sie aus dieser Übung gesehen haben sollten, können wir jetzt schon wesentlich komplexere Muster erfassen, wobei diese aber immer noch relativ fest vorgegeben sein müssen. So können wir z.B. angeben, dass vielleicht bestimmte Teile des Musters, wie etwa Wortzeichen, auch mehrmals vorkommen können. Wie wir dies erreichen, aber auch begrenzen, können, lernen wir zum Teil schon im nächsten Abschnitt.

6.5 Quantifizierung und Begrenzung

Die Quantifizierung erlaubt es uns, anzugeben, dass ein Zeichen, eine Zeichenklasse oder Gruppe von Zeichen – mehr dazu in Abschnitt 6.8 – entweder optional ist, wie z.B. das `<h>` in *Joghurt* vs. *Jogurt*, oder mindestens in einer gewissen Anzahl auftreten muss, z.B. das zweimalige `<e>` in *See*. Idealerweise verwendet man die Quantifizierung zusammen mit Grenzmarkierungen oder Ankern, worüber wir ebenfalls in Abschnitt 6.8 mehr erfahren werden.

Zur Angabe der Quantifizierung verwendet man verschiedene **Quantifikatorsymbole** und Klammern, die wir hier kurz zusammenfassen wollen. Ein Asterisk (*) zeigt an, dass, was vorangeht, fehlen oder unbegrenzt oft auftreten kann. Dies bedeutet also in natürliche Sprache übersetzt ungefähr ‚kein bis unendlich viele Mal‘. So findet `e\w*`

das Zeichen `e` alleine, oder gefolgt von einer beliebigen Anzahl von Zeichen innerhalb eines Wortes (z.B. *gealtert*, *Element*) oder keinem (z.B. *alle*).

Das Fragezeichen (`?`) zeigt an, dass, was voransteht, optional ist oder (maximal) einmal auftreten darf, übersetzt also ‚vielleicht (nicht) bis (maximal) einmal‘ heißt. Ein Beispiel dafür wäre `wohns?t`, welches *wohnt* oder *wohntst* findet.

Das Plus-Zeichen (`+`) bedeutet mindestens einmal, aber bis zu einer beliebigen Anzahl, also ‚(mindestens) einmal bis unendlich viele Male‘. Beispielsweise findet `geh\w+` *gehst* oder *gehend*, aber nicht *geh*.

Um präziserer Längenbereiche anzugeben, kann man geschweifte Klammern (`{...}`) verwenden. So findet `\w{5}` – zumindest theoretisch, da wir ja Wörter bisher noch nicht abgrenzen können – genau 5 Wortzeichen, `\w{5, }` mindestens 5 Wortzeichen, `\w{5,10}` zwischen 5 und 10 Wortzeichen, und `\w{0,5}` kein bis 5 Wortzeichen.

Die allgemeinen Quantifikatoren `+` und `*` versuchen immer, maximale Treffer zu erzielen, das heißt, sie sind ‚gierig‘ (engl. *greedy*). Dies kann aber zu Begrenzungsproblemen führen, z.B. beim ‚Löschen‘ von HTML Tags mittels der `sub()`-Methode, wobei `sub` eine Abkürzung des englischen *substitute*, also dt. *ersetzen*, ist. Wendet man

```
re.sub('<.+>', '', '<b>fetter Text</b>')
```

auf den als zweites Argument angegebenen HTML-Tag (siehe Kapitel 10) an, der den darin befindlichen Text als fettgedruckt (engl. *boldface*) markiert, um diesen Tag zu entfernen, ist das unerwünschte Resultat, dass gar kein Text mehr übrig bleibt. Die gierige Quantifikation `+` erfasst nämlich so viele beliebigen Zeichen wie nur möglich, bevor ein `>` gefunden wird. Dabei ist leider in diesen beliebigen Zeichen auch das `>` selbst enthalten, weshalb alles bis inklusive der letzten spitzen Klammer gelöscht wird. Die Lösung hierfür ist, dass wir den Quantifikator selbst noch einmal durch `?` einschränken, so dass er maximal einmal bis zur Endmarkierung des Suchbegriffs auftreten darf und dann der Ausdruck

```
re.sub('<.+?>', '', '<b>fetter Text</b>')
```

tatsächlich nur den Text ohne Tagbestandteile zurückliefert.

6.6 Maskieren und Verwendung von Sonderzeichen

Wie wir schon gesehen hatten, haben ‚Interpunktions‘-Zeichen manchmal spezielle Bedeutung in Regexes, z.B. das Fragezeichen als Quantifikator, der Punkt als beliebiges Zeichen, oder der Bindestrich als Bereichsanzeiger, wenn auch nur in Zeichenklassen. Um diese buchstäblich verwenden zu können, kann man sie entweder durch einen rückwärtsgerichteten Schrägstrich maskieren, so dass `\.` jetzt tatsächlich ‚Punkt‘ bedeutet und `\\` einen rückwärtsgerichteten Schrägstrich selbst findet, oder sie in Zeichenklassen setzen, wie etwa in `[.?]`. Um Bindestriche ohne Maskierung zu

verwenden, muss man sie an Stellen in Zeichenklassen setzen, wo sie keinen Bereich markieren können, das heißt an den Anfang oder das Ende, wie in `[\w-]+`.

Im Prinzip sollten wir immer, wenn wir ein Muster erstellen, eine rohe Zeichenkette dafür verwenden. Wie wir schon in der Lösung zu Übung 30 festgestellt hatten, erspart uns dies, zu viele rückwärtige Schrägstriche zur Maskierung verwenden zu müssen, insbesondere auch, da wir in Regexes häufig mit Zeilenumbrüchen (`\n`) oder verschiedenen Arten von Leerzeichen (`\s`) arbeiten werden und diese sonst jedes Mal maskieren müssten.

6.7 Regex-Fehlerbehandlung

Da selbst einfachere Regexes manchmal relativ kompliziert und dadurch fehleranfällig sein können, benötigen sie idealerweise immer eine Fehlerbehandlung. Diese wird durch das `re.error`-Objekt zur Verfügung gestellt. Dieses Fehler-Objekt hat eine Reihe von Attributen, von denen für uns die Folgenden am nützlichsten sind.

Attribut	Information
<code>msg</code>	beinhaltet eine Nachricht, welcher Fehler aufgetreten ist (auf Englisch)
<code>pattern</code>	liefert das Muster zurück, bei dem der Fehler aufgetreten ist
<code>pos</code>	liefert die Position (als Integer) zurück, an der Python den Fehler gefunden hat

Tabelle 14: Nützliche Attribute des `re.error`-Objekts

Wichtig ist zu beachten, dass die von `pos` angezeigte Position den Fehler nicht immer eindeutig erkennen lässt, da z.B. bei nicht ordnungsgemäß geschlossenen Zeichenklassen die Position der öffnenden Klammer als Fehlerort angegeben wird! Deshalb erfordert eine Interpretation des Fehlers oft Rückschlüsse, die auf Position und Fehlermeldung gemeinsam basieren.



Übung 31 - Regexes testen

Das vorherige Programm war dadurch, dass es unbedingt eine Zeichenklasse bei der Eingabe erwartet hat, etwas zu restriktiv. Schreiben Sie es deshalb als `19_regexes_testen.py` so um, dass Sie alle möglichen wohlgeformten regulären Ausdrücke damit testen können. Dazu können Sie zunächst das bestehende Programm, da ja viele Teile gleich sind, erst unter dem neuen Namen abspeichern, und dann zunächst die Bedingungsabfrage herauslöschen, die auf eine Zeichenklasse prüft.

Allerdings wollen wir diesmal auch üben, mit einem vorkompilierten Regex-Objekt zu arbeiten. Deshalb kompilieren wir das Kommandozeilenargument erst, bevor wir es der Variable für den Suchbegriff zuweisen. Falls Sie später auf der Kommandozeile eine fehlerhafte Regex eingeben, wird dadurch automatisch ein Fehlerobjekt erzeugt und wir können diesen Fehler abfangen.

Fügen Sie deshalb zur Fehlerbehandlung von Regex-Fehlern hinter der `exception` für das (fehlende) Programmargument eine weitere für einen `re`-Fehler mit ein, und beenden Sie darin das Programm, falls ein Fehler aufgetreten ist, mit einer Kombination aus Nachricht, Muster und Position des Fehlerobjektes in geeigneter Form zur Information des Nutzers.

Passen Sie die Regex-Suche und Iteration im unteren Programmteil jeweils so an, dass Sie hier das vorkompilierte Objekt verwenden, anstatt die Methoden über das Modul aufzurufen.

Testen Sie Ihr Wissen zur Quantifizierung, indem Sie einige der Beispiele von oben sowie auch selbst gewählte anhand des zuletzt geschriebenen Programms ausprobieren.

Versuchen Sie insbesondere, Wörter mit einer bestimmten Anzahl von Buchstaben zu finden.

Beachten Sie dabei jedoch immer die Einschränkungen innerhalb unseres Programms sowie die Beschränkungen, die die reine Quantifizierung uns auferlegt. Was können wir aufgrund letzterer hier noch nicht erreichen?

6.8 Verankerung, Gruppen und Alternation

Wie wir gerade erst wieder beobachten konnten, können Treffer auch manchmal an Stellen auftreten, wo wir sie eigentlich nicht gesucht haben, beispielsweise innerhalb von Wörtern, selbst wenn wir eigentlich ganze Wörter finden wollten. Regexes zu verankern erlaubt es uns, präziser zu bestimmen, wo ein Muster treffen sollte. Die Regex-Implementierung der `re`-Moduls stellt uns dafür mehrere Anker zur Verfügung, von denen der nützlichster die Wortgrenze `\b` (engl. *word boundary*) ist, so dass wir mithilfe des Ausdrucks `\b\b{5}\b` mit einer Wortgrenze jeweils am Anfang und Ende tatsächlich nur Wörter finden könnten, die exakt fünf Buchstaben lang sind und somit jetzt eines der Probleme, die wir aufgrund unseres beschränkten Wissens noch in der letzten Übung hatten, lösen können. Dies sollten Sie gleich austesten, bevor Sie weiterlesen.

Andere Optionen zur Verankerung in größeren Texteinheiten stellen die Anker `^` für Zeilenanfang, `$` für Zeilenende, `\A` für den Beginn einer Kette und `\Z` für deren Ende dar. Hier ist die Verwendung des Carets (`^`) nicht zu verwechseln mit der innerhalb einer Zeichenklasse.

Gruppen erlauben uns, ganze Zeichenfolgen auf einmal zu treffen und quantifizieren, anstatt nur einzelne Zeichen. Erzielt wird dies durch Klammerung mit runden Klammern `(...)` um eine oder mehrere Gruppen. So z.B. findet `(nd)?` unter Umständen eine optionale Partizipialendung und `(en)` eine Infinitivendung, wobei beides natürlich nicht bei allen Wortformen zutrifft. Der Klammerinhalt kann normalerweise zur Extraktion oder Verwendung als sogenannte Rückwärtsreferenz – mehr dazu in Abschnitt 6.9 – gespeichert werden. Will man nur gruppieren, aber nichts speichern, kann die Speicherung durch `?:` direkt hinter der öffnenden Klammer unterdrückt werden. Weiterhin ist es auch möglich durch

```
?P<Gruppenname>
```

eine Gruppe explizit zu benennen, wie z.B. in

```
(?P<inf_suffix>en)
```

Zugriff auf die gespeicherten Gruppen erhalten wir entweder über

```
Treffer.group (Nummer)
```

bei einfacher Gruppierung oder

```
Treffer.group (Name)
```

bei benannter Gruppierung. Rückwärtsreferenzen mittels `\1`, `\2` etc. oder `(?P=Name)` erlauben es uns, etwas noch einmal wiederzuverwenden, was durch eine Gruppierung schon gefunden wurde, unter Umständen, um verdoppelte Wörter zu finden. Wir können sie aber auch in einfachen Ersetzungen mit `re.sub()` verwenden, z.B. um Wörter miteinander zu vertauschen, so wie bei unseren Beispielen zur Inversion.

Außer den schon besprochenen gibt es auch noch zusätzliche Methoden zum Umgang mit Gruppierungen. So liefert `groups()` ein Tupel aller Treffergruppierungen als Zeichenketten zurück und `groupdict()` ein Dictionary benannter Gruppierungen, was z.B. zur Extraktion von Attributen aus HTML oder XML-Code, die wir in Kapitel 10 besprechen werden, sehr nützlich sein kann, insofern uns diese Attribute im Voraus bekannt sind.

Um mehrere Muster innerhalb einer Gruppe gleichzeitig angeben zu können, kann man zwischen den verschiedenen Elementen einen Separator `(|)` eingeben. So könnten wir über `(nd|en)?` die zwei vorher schon angesprochenen Suffixe gleichzeitig als optionale Alternativen angeben, weshalb wir hier auch von **Alternation** sprechen.



Übung 32 - Wörter mit Regexes suchen

Testen Sie Ihr Wissen über die gerade beschriebenen Regex-Optionen, indem Sie in der Beispieldatei und mittels des zuletzt geschriebenen Programms nach Wörtern suchen,

- mit bestimmter Länge (diesmal präzise),
- mit verschiedenen Prä- oder Suffixgruppen,
- oder an bestimmten Positionen.

Testen Sie auch mit einer Rückwärtsreferenz, sowohl einer einfachen als auch einer benannten, ob Sie ein Wort finden können, was ‚verdoppelt‘ vorkommt.

6.9 Weitere Treffereingrenzungen

Wir haben gerade schon gesehen, wie wir durch Verankerung unsere Möglichkeiten, präzise Treffer zu erzielen, um einiges erhöhen können. Aber Wort- oder Zeilengrenzen sind natürlich nicht immer die einzigen Kontexte, in denen wir suchen wollen. Manchmal wollen wir auch ausdrücken, dass bestimmte Wörter oder Zeichenketten innerhalb des Kontextes auftreten dürfen oder auch nicht. Da der Kontext aber beiderseits des Suchbegriffes relevant sein kann, brauchen wir dafür auch Möglichkeiten, ihn entweder auf der linken, rechten oder auf beiden Seiten einzugrenzen. Dies erzielen wir bei Regexes durch sogenannte **Umherschau** (engl. *lookaround*), wo wir durch weitere Muster bestimmen, was vor oder hinter einem Ausdruck (nicht) auftreten darf, ohne jedoch dabei in den Treffer mit aufgenommen zu werden. Um zu erreichen, dass diese Muster nicht Teil(e) des Treffers werden, verwenden wir wieder runde Klammern, bei denen gleich nach der öffnenden Klammer ein Fragezeichen steht, so ähnlich wie wir das schon für die Unterdrückung der Speicherung mittels (`?:...`) gesehen haben.

Die **Vorausschau** (engl. *lookahead*) schränkt den möglichen Folgekontext ein, also das, was rechts vom eigentlichen Suchmuster steht. Die positive Form ist dabei markiert durch

```
(?=Ausdruck)
```

das heißt, etwas muss folgen, wohingegen was nicht folgen darf, also die negative Form, durch

```
(?!Ausdruck)
```

angegeben wird. Zum Beispiel findet der positive Ausdruck `Haus(?=aufgabe)` *Hausaufgabe*, *Hausaufgaben*, und *Hausaufgabenheft*, und der negative `Haus(?!aufgabe)` *Hausmann*, *Hausfrau*, *Haushalt* etc., aber nicht die Ergebnisse der vorher angegebenen positiven Suche. Bei der Vorausschau sind auch variable Muster erlaubt.

Im Gegensatz dazu schränkt die **Rückwärtsschau** (engl. *lookbehind*) den möglichen vorhergehenden Kontext ein, wobei aber leider nur eine feste Länge erlaubt ist. Hier ist die positive Form markiert durch

```
(?<=Ausdruck)
```

das heißt, etwas muss vorausgehen, und die negative durch

```
(?<!Ausdruck)
```

das heißt, etwas darf nicht vorausgehen. Die positiven oder negativen Markierungen für die Vorwärts- und Rückwärtsschau unterscheiden sich also nur dadurch, dass bei der Rückwärtsschau eine stilisierte Pfeilspitze (<), die zurückzeigt, direkt hinter dem Fragezeichen angegeben wird, wohingegen bei der Vorausschau diese Richtungs- oder Positionsangabe fehlt. Beispiele für die Rückwärtsschau sind für die positive Form `(?<=Haus)aufgabe`, welches nur Komposita mit {aufgabe} findet, die mit {Haus} anfangen und für die negative `(?<!Haus)aufgabe`, was nur solche findet, die nicht mit {Haus} anfangen. Auch `(?<!(?:Haus|Teil)aufgabe)` funktioniert, da es sich ja um zwei Alternativen gleicher Länge handelt, und findet beispielsweise *Arbeitsaufgabe*, aber nicht *Hausaufgabe* oder *Teilaufgabe*. Um bei der Rückwärtsschau die Beschränkung in der Variabilität zu umgehen, kann man allerdings mehrere Klammern hintereinander verwenden, was leider in den meisten Beschreibungen zu Regexes nicht erwähnt wird. So funktioniert also auch `(?<!Haus)(?<!Arbeits)aufgabe`, um weder *Haus-*, noch *Arbeitsaufgabe* zu finden, obwohl beide nicht die gleiche Länge haben.

6.10 Kompilierungsflags

Kompilierungsflags verändern die normalen Eigenschaften des Ausdrucks auf verschiedene Art. Die Namen in Klammern hinter den jeweiligen Syntaxoptionen sind die Bezeichner für die Optionen in Python 2. Ich werde sie nur hier anfügen, da sie teilweise sprechender sind, und für den Fall, dass Sie einmal älteren Python-Code lesen und verstehen müssen. Wie üblich, besprechen wir wieder nur die wichtigsten Optionen.

Die Verwendung von `re.I` (`IGNORECASE`) bewirkt, dass das Muster nicht nach Groß- und Kleinschreibung unterscheidet, so dass `re.search(r'Haus', re.I)` sowohl *Haus*

als auch `haus`, aber auch `HAUS` oder `hAUSt`, also alle Permutation mit Klein- oder Großbuchstaben, findet. Denselben Effekt könnte man, natürlich viel umständlicher, erreichen, wenn man jeden Buchstaben als Zeichenklasse mit dem jeweiligen Groß- und Kleinbuchstaben schreiben würde. Wie hier gezeigt, ist dies gerade häufig nützlich, wenn der Anfangsbuchstabe entweder groß- oder kleingeschrieben werden kann, etwa wenn ein Morphem Bestandteil eines Kompositums sein kann, oder bei einer Präfixsuche, die sowohl Wörter am Satzanfang als auch innerhalb des Satzes finden soll.

Das Flag `re.X` (VERBOSE) dient zur Kommentierung oder Verbesserung der Übersichtlichkeit bei komplexeren Mustern, oder um die Funktion einfacher erkennen zu können, wie z.B. `(der|die|das) \b([A-ZÄÖÜ]\w+)\b`, was folgendermaßen umgeschrieben werden kann.

```
(der|die|das) # findet (unflektierten) Artikel
\s
\b([A-ZÄÖÜ]\w+)\b # findet beliebiges Nomen, re.X
```

Hierbei wird alles rechts von einem Rautensymbol (#) als Kommentar interpretiert und alle Leerzeichen, inklusive Zeilenumbrüchen, ignoriert, so dass alle normalen Leerzeichen als `\s` geschrieben werden müssen. Falls Rautenzeichen im Muster selbst auftreten sollen, müssen diese maskiert werden.

Normalerweise treffen `^` und `$` bei einer Zeichenkette, die mehrere Zeilen beinhaltet, nur jeweils am Anfang oder Ende der kompletten Kette. Oft ist es aber der Fall, dass wir beim Abarbeiten von Dateien die ganze Datei als eine lange Zeichenkette mit vielen Zeilenumbrüchen einlesen und Ersetzungen z.B. am Ende jeder Zeile durchführen wollen, wie etwa ein Satzzeichen zu entfernen. Dazu können wir `re.M` (MULTILINE) verwenden, was uns tatsächlich den einfachen Zugriff auf alle Zeilenenden mit `$` ermöglicht.

Ebenso trifft ohne die Anwendung von `re.S` (DOTALL) normalerweise der Punkt (`.`) keine Zeilenumbrüche selbst, auch wenn er sonst alle möglichen Zeichen abdeckt. Mehre Flags können auch durch `|` miteinander kombiniert werden.

In diesem Kapitel haben wir gelernt, wie wir mithilfe von Regexes unsere Suchen nach bestimmten Zeichenketten optimieren können. Beim Programmieren für Sprachanalysen kommt es jedoch nicht nur darauf an, effizient Muster zu erkennen, sondern auch den Aufbau unserer Programme möglichst effizient zu gestalten, teils um die Programme zu verkürzen, teils um komplexere Aufgaben damit einfacher zu gestalten. Um zu erfahren, wie dies geht, wenden wir uns im nächsten Kapitel der Modularisierung und Objektorientierung zu.



6.11 Lösungen zu den Aufgaben

Lösung 29 - Einfache sequenzielle Muster erkennen

Wie man das Kommandozeilenargument abfragt und über die Eingabedatei – natürlich mit Fehlerbehandlung – iteriert, haben wir schon zur Genüge geübt, so dass die ersten Schritte im Programm eigentlich selbstverständlich sind und wir nur besprechen müssen, was genau in der `for`-Schleife, die die Zeilen abarbeitet, geschehen sollte.

Hier sollte zunächst das Ergebnis der Suchoperation mittels

```
ergebnis = re.search(suchbegriff, zeile)
```

der entsprechenden Variable zugewiesen werden. Im nächsten Schritt sollte dann eine `if`-Anweisung, so wie oben im Text als Beispiel gegeben, überprüfen, ob überhaupt ein Ergebnis vorliegt. Einen `else`-Block benötigen wir hier nicht, da für den Fall, das nichts gefunden wird, auch keine Aktion im Übungstext spezifiziert ist. Liegt jedoch ein Treffer vor, so sollten die Start- und Endpositionen innerhalb eines Tupels mit

```
(anfang, ende) = ergebnis.span()
```

also der Anwendung der `span()`-Methode, den entsprechenden Variablen zugewiesen werden.

Zuletzt müssen wir nur noch die entsprechenden Teile der Zeile mittels Slicing extrahieren und dann wieder zusammen mit den Klammern, die zur Markierung des Ergebnisses dienen, mittels

```
print(f'{zeile[:anfang]}[{ergebnis.group()}]{zeile[ende:]}',  
      end='')
```

ausgeben. Dabei verwenden wir wieder das `end`-Schlüsselwortargument, diesmal aber mit leerer Zeichenkette zur Unterdrückung der zusätzlichen Zeilenumbrüche.

Zu beobachten wäre hier gewesen, dass zwar die Markierung durch die Klammern ein Ergebnis hervorhebt, aber, falls dieselbe Kette mehrmals auf der gleichen Zeile auftritt, nur jeweils das erste Ergebnis markiert wird, was nicht optimal ist. Unter Umständen hätten Sie erwartet, dass wir durch unsere Suchen entweder Präfixe oder Präpositionen finden, wobei ein großer Teil der Vorkommen der gesuchten Zeichenketten innerhalb anderer Wörter auftritt. Wir werden in Abschnitt 6.8 noch lernen, wie man reguläre Ausdrücke so einschränkt, dass tatsächlich nur solche Ergebnisse gefunden werden.

Außerdem haben wir in diesem Programm gar nicht sichergestellt, dass überhaupt ein Muster als Argument übergeben wurde, was Sie allerdings nur dann feststellen werden, falls Sie vergessen, das Muster auf der Kommandozeile als Argument zu übergeben. In einem sicheren Programm sollte aber auch die Übergabe von Argumenten getestet werden, weshalb wir das im nächsten Programm tun wollen.

Für diejenigen unter Ihnen, die schon über Grundwissen in der Korpuslinguistik verfügen, sollte hoffentlich klar geworden sein, dass wir hiermit ein ganz rudimentäres Konkordanzprogramm entwickelt haben.

Lösung 30 - Zeichenklassen testen

Zunächst sollten Sie nicht vergessen, wieder das `sys`- und das `re`-Modul zu importieren. Um dann auf die Eingabe eines Arguments zu überprüfen, sollte die Zuweisung an die entsprechende Variable für den Suchbegriff diesmal schon innerhalb des `try`-Blocks geschehen. Falls hier kein Argument übergeben wurde, so ist die Indexposition 1 von `sys.argv` nicht belegt, weshalb der `IndexError` auftreten würde, den Sie im `except`-Block abfangen sollten.

Ebenfalls innerhalb des `try`-Blocks sollte dann nach dem Versuch, den Suchbegriff in der Variable abzuspeichern, die Überprüfung des Suchbegriffs auf das Vorkommen unseres Musters erfolgen, was mit der einfachen negierten `if`-Abfrage

```
if not re.search('[.\\[\\]\\]', suchbegriff):
```

geschehen kann. Hier wollen wir ja nur wissen, ob der Suchbegriff nicht vorkommt, so dass wir in diesem Fall ebenfalls das Programm, aber mit einer passenden Meldung, abbrechen. Die Definition des Suchmusters ist etwas komplex geraten, da wir noch nicht besprochen hatten, dass wir sinnvollerweise rohe Zeichenketten für unsere Suchmuster in regulären Ausdrücken verwenden sollten, damit wir nicht alle rückwärtsgerichteten Schrägstriche – engl. *backslash* – maskieren müssen. Im Prinzip müssen wir deshalb hier für die bisher besprochenen Regex-Konstrukte eine Zeichenklasse angeben, die

1. mit einer öffnenden eckigen Klammer beginnt,
2. einen Punkt (als Abkürzung für jedes beliebige Zeichen) beinhaltet,
3. prüft, ob entweder eine öffnende eckige Klammer (diesmal für unsere gesuchte Klasse selbst), oder ein rückwärtsgerichteter Schrägstrich (als möglicher Anfang einer Abkürzung) existiert, wobei wir
4. die schließende eckige Klammer unserer Klasse ebenfalls maskieren müssen, damit sie nicht als Teil unserer zu überprüfenden Zeichenklasse interpretiert wird.

Als rohe Zeichenkette könnten wir den Suchbegriff etwas kürzer und lesbarer als `r'[.\\[\\]']` gestalten, da wir hier nur noch die öffnende Klammer und den Schrägstrich selbst maskieren müssen. Sollten Sie versehentlich einen Teil der komplexeren Definition vergessen haben, wird Ihnen das `re`-Modul als Fehler "unterminated character set" melden.

Die Beispieldatei mit Fehlerbehandlung zum Lesen zu öffnen und zeilenweise abzuarbeiten, dürfte wieder kein Problem darstellen, und Sie könnten sogar fast denselben `try`-Block einfach aus der vorherigen Datei übernehmen, wobei der `except`-Block sowieso gleich bleibt. Allerdings wollen wir diesmal nicht gleich ein Suchergebnis einer Variable zuweisen, um damit zu arbeiten, sondern nur mittels der `if`-Anweisung

zunächst überprüfen, ob überhaupt ein Treffer vorliegt, da wir ja wissen, dass unter Umständen auch mehr als ein Treffer vorliegen kann.

Die leere Zeichenkette soll dafür verwendet werden, um schrittweise Teile der Zeile bis zum jeweiligen Treffer zu extrahieren, und diese dann, zusammen mit den jeweiligen Markierungen und dem Treffer, abzuspeichern, so dass die komplette Zeile, aber eben diesmal mit allen Treffern markiert, zusammengesetzt werden kann. Falls wir dabei mehrere Treffer – mit mehreren Start- und Endpositionen – haben, muss unter Umständen die vorherige Endposition des alten Treffers zur Anfangsposition für die Extraktion aller Zeichen bis zum Beginn des nächsten Treffers werden. Deshalb benötigen wir eine Hilfsvariable, die wir sinnvollerweise `start` nennen können.

Danach iterieren wir mittels einer `for`-Schleife und der `finditer()`-Methode des `re`-Objekts über alle Treffer-Objekte und arbeiten diese ab. Zur Bestimmung der Anfangs- und Endpositionen können wir hier wieder ein Tupel über die `span()`-Methode erzeugen, wie schon in der letzten Übung. Danach hängen wir an die bestehende, anfangs leere, Zeichenkettenvariable für die neue Zeile mit den Markierungen über Slices alle Zeichen ab der Startposition bis zum Anfangsindex des Treffers, wieder eine Startmarkierung, dann den Treffer, und eine Endmarkierung, an. Um auf den Treffer zuzugreifen, verwenden wir, wie oben besprochen, die `group()`-Methode des Treffer-Objekts, die ohne Argument immer den gesamten Treffer zurückliefert, so dass die Anweisung

```
zeileNeu += f'{zeile[start:anfang]}[{treffer.group()}]'
```

sein sollte. Damit wir aber beim etwaigen nächsten Treffer nur die Teile der Zeile zum Anhängen ab dem Ende des vorherigen Treffers extrahieren, müssen wir der Variable `start` die Endposition des letzten Treffers zuweisen.

Nach dem Abarbeiten aller Treffer können wir wieder unser Ergebnis für die jeweilige Zeile ausgeben, was entweder

```
print(zeileNeu + zeile[ende:-1])
```

oder

```
print(zeileNeu + zeile[ende:], end='')
```

sein sollte, da wir ja das Zeilenende abschneiden oder unterdrücken wollen.

Lösung 31 - Regexes testen

Zum Abspeichern des Programms unter einem neuen Namen müssen Sie nur in der IDE die Option `Datei → Speichern unter...` auswählen und den neuen Namen vergeben. Hierdurch wird automatisch eine Kopie mit dem neuen Namen erstellt, die im Editorfenster die alte Datei ersetzt, so dass Sie gleich damit weiterarbeiten können.

Nach dem Löschen der Bedingungsabfrage müssten Sie die ursprüngliche Zuweisung

```
suchbegriff = sys.argv[1]
```

zu

```
suchbegriff = re.compile(sys.argv[1])
```

ändern. Dadurch enthält dann die Variable `suchbegriff` natürlich nicht mehr eine Zeichenkette, sondern jetzt ein Regex-Objekt. Dieses ist direkt verwendbar, falls bei der Kompilierung nicht ein Fehler aufgetreten ist, welchen wir aber später in der passenden `exception` abfangen würden.

Die zusätzliche Fehlerbehandlung für den potenziellen Regex-Fehler können Sie einfach hinter der ursprünglichen Fehlerbehandlung für den Indexfehler mit anfügen, da Python mehrere `except`-Blocks erlaubt. Dieser sollte in etwa so aussehen:

```
except re.error as e:
    sys.exit(f'Regexfehler="{e.msg}" in Muster: "{e.pattern}"'
            f' an Position {e.pos}')
```

Die Anpassung der Suche und Iteration sind sehr einfach, da jetzt die Methoden direkt über das Regex-Objekt `suchbegriff` aufgerufen werden können und somit nur noch ein einzelnes Argument im Methodenaufruf, nämlich das Suchziel, angegeben werden muss. Deshalb können wir jetzt

```
suchbegriff.search(zeile)
```

und

```
suchbegriff.finditer(zeile)
```

schreiben.

Das Programm sollte Ihnen jetzt erlauben, mithilfe einer geeigneten und fehlerfreien Regex alle möglichen Regex-Muster auszutesten und mögliche Fehler dabei gleich zu melden. Wozu wir jedoch bisher keine Meldung bekommen ist, falls auf keiner der Zeilen ein Ergebnis gefunden wurde. Dies sehen wir momentan immer nur daran, dass keinerlei Ausgabe in unsere Datei erfolgt. Wie Sie dieses Problem relativ leicht beheben könnten, überlasse ich ihnen als Denkaufgabe. Einen kleinen Tipp gebe ich aber, und zwar, dass Sie eine boolesche Variable dafür verwenden könnten, die Sie am Programmende abfragen und gegebenenfalls dann eine geeignete Meldung ausgeben. In einem größeren Programm mit geeigneter Interaktion mit Benutzern sollte man solch eine Lösung auf jeden Fall mit implementieren.

Beim Austesten der verschiedenen Wortlängenbegrenzungen sollte Ihnen aufgefallen sein, dass auch Wörter gefunden werden, die mehr Buchstaben haben, als wir angegeben haben, da wir bisher noch nicht wissen, wie wir in Regexes Wortgrenzen angeben können. Sie könnten sich zunächst damit behelfen, dass Sie Leerzeichen um Ihre Suchbegriffe setzen. Dies würde ihnen vielleicht die meisten Wörter korrekt anzeigen, aber leider keine, die am Zeilenanfang oder vor einem Satzzeichen stehen.

Lösung 32 - Wörter mit Regexes suchen

Zur Suche nach Prä- oder Suffixgruppen sollten Sie diesmal mit Alternation arbeiten und die Prä- oder Suffixe an den jeweiligen Wortpositionen mittels `\b` verankern. Die Positionen, an denen Sie suchen können, sind natürlich Zeilenanfänge oder Enden, wobei Sie beachten sollten, dass am Zeilenende in unserer Beispieldatei immer ein Punkt steht, so dass Sie mit einer Suche `\b\w+\$` oder etwas genauer Quantifiziertem nichts finden würden.

Da man auf der Kommandozeile auch Argumente ohne Anführungszeichen eingeben kann, könnten Sie beim Versuch, doppelte Wörter zu finden, ein Problem haben, falls Sie vergessen, Ihr Argument in betriebssystemkonformen Anführungszeichen anzugeben, es sei denn, Sie verwenden `\s`, um das Leerzeichen zwischen dem ersten und zweiten Wort zu markieren. Falls die Anführungszeichen oder `\s` fehlen, erkennt Python anstatt einem Argument aber zwei, die durch ein Leerzeichen getrennt sind, und verwendet nur den Teil Ihres Musters vor dem Leerzeichen. Dadurch werden bei korrekter Formulierung alle Wörter markiert ausgegeben, da das Muster entweder

```
(\b\w+\b) \1
```

oder etwas wie

```
(?P<wort1>\b\w+\b) (?P=wort1)
```

bei Verwendung einer benannten Gruppe, sein sollte.

7 Modularisierung und Objektorientierung



Bei der Programmierung kommt es häufig vor, dass man gleiche oder ähnliche Funktionen mehrmals, aber mit verschiedenen Daten, oder in unterschiedlichen Programmen, ausführen muss. Um Programme besser strukturieren zu können und zu vermeiden, ständig ‚das Rad neu zu erfinden‘, teilt man deshalb komplexere Programme in Unterprogramme auf, die entweder innerhalb desselben Programms oder in externen Modulen ‚ausgelagert‘ werden. In diesem Kapitel wollen wir besprechen, wie eine solche Modularisierung erreicht werden kann und wie man bestehende oder selbstverfasste Module wieder in eigene Programme importieren kann. Dabei wollen wir auch die verschiedenen Stufen der Modularisierung, von der Erstellung benutzerdefinierter Funktionen über separate Module bis hin zur Objektorientierung, besprechen.

Um später effizientere Funktionen und Module entwickeln zu können, bietet es sich jedoch an, erst noch einen weiteren Datentyp, den Dictionary (`dict`), genauer zu besprechen, da dieser oft zur Speicherung komplexerer Informationen in Modulen verwendet wird.

7.1 Dictionaries

Dictionaries erlauben es uns, Schlüssel-Werte-Paarungen zu speichern, wobei jeder Schlüssel nur einmal zugelassen ist, so wie bei den Elementen eines Sets. Dieser Datentyp eignet sich hervorragend zum Speichern von einfachen Wörterbüchern oder Frequenzlisten, also immer da, wo entweder Äquivalenzen auftreten oder einem Element ein bestimmter Wert zugeordnet werden soll, wobei diese Werte auch aus komplexeren Datentypen bestehen können. Zum Beispiel kann man einen Index aller Wörter in einem Text erstellen, bei dem der Schlüssel die jeweilige Wortform ist und der Wert aus einer Liste von Textpositionen besteht, an dem diese Wortform auftritt.

Einen leeren Dictionary kann man entweder über die `dict()`-Funktion, wie etwa

```
woerterbuch = dict()
```

oder direkt durch die Verwendung von leeren geschweiften Klammern anlegen, z.B.

```
woerterbuch = {}
```

Ein Dictionary kann aber auch schon direkt beim Anlegen mit Werten initialisiert werden, wie in

```
woerterbuch = {'ein': 'a', 'das': 'the', 'die': 'the'}
```

Dabei werden Schlüssel und Werte einander mittels eines Doppelpunkts zugeordnet und die einzelnen Paare durch Kommas getrennt, wie auch bei anderen Sequenzen. Um Daten in einen bestehenden Dictionary direkt einzufügen bzw. diese zu verändern, verwendet man den Namen des Dictionaries, gefolgt von einem Schlüsselnamen in eckigen Klammern, wie in

```
woerterbuch['ein'] = 'a'
```

oder eine der Zugriffsmethoden zur Iteration, die wir als Nächstes besprechen werden.

Methode	Funktionalität
<code>keys()</code>	liefert alle Schlüssel zurück, in derselben Reihenfolge, in der sie ursprünglich eingefügt wurden
<code>values()</code>	ist das Gegenstück zu <code>keys()</code> , liefert also alle Werte zurück, wird aber seltener verwendet
<code>items()</code>	liefert ein Tupel von Schlüssel-Werte-Paaren zurück

Tabelle 15: Wichtigste Dictionary-Methoden

Wichtig ist dabei, dass Dictionaries normalerweise nicht sortiert sind, es sei denn, dass sowieso schon eine sortierte Eingabe von Schlüsseln erfolgt ist. Deshalb muss oft entweder eine geeignete Sortierung bei der Ausgabe oder Verarbeitung erfolgen, oder eine sortierte Kopie des Dictionary erzeugt werden, über die man später iterieren kann.

Zwei weitere nützliche Methoden sind `setdefault(Schlüssel, Wert)` und `clear()`. Ersteres gibt den aktuellen Wert eines Schlüssels zurück, falls dieser existiert, legt aber eine Vorgabe für den Wert an, falls er noch nicht besteht. Dies ist vor allem wichtig, wenn der Schlüsselname im Programm generiert wird und ein Zähler als Wert angelegt werden muss. Letzteres löscht, wie auch bei Sequenzen, den Dictionary-Inhalt.

Der `in`-Operator kann verwendet werden, um zu prüfen, ob ein Schlüssel existiert, oder um in einer `for`-Schleife mittels der oben beschriebenen Iterationsmethoden über alle Schlüssel, Werte oder deren Kombinationen zu iterieren.

7.2 Modularisierung

Modularisierung erlaubt es, Programmabläufe durch Zerlegung übersichtlicher zu gestalten und mehrfach benötigte Programmteile auszulagern und wieder zu verwenden, um Redundanzen zu vermeiden. Wir können drei verschiedene Ebenen unterscheiden, die der (benutzerdefinierten) Funktionen, Module und Klassen.

Funktionen erledigen meist kleinere Arbeitsschritte, die öfter im Programm gebraucht werden, z.B. bei der Konvertierung bestimmter Daten, die wiederholt mit verschiedenen Objekten wie Sätzen, oder auch ganzen Dateien ausgeführt werden

sollen. Module dienen normalerweise dazu, mehrere solcher Funktionen, aber auch für die Aktionen benötigte Variablen, zusammenzufassen. Sie werden oft in einer einzelnen Datei abgespeichert, die man später einfach zur Wiederverwendung in verschiedene Programme importieren kann, so, wie wir das schon mit einigen Modulen getan haben. Letztlich erlauben es Klassen, neue Datentypen als Objekte mit eigenen Methoden und zugeordneten Variablen/Attributen zu erstellen, wobei auch mehrere Klassen in einer Moduldatei definiert werden können, um z.B. ähnliche Objekte zusammengruppieren und effizient zu importieren.

7.3 Benutzerdefinierte Funktionen

Benutzerdefinierte Funktionen übernehmen, ebenso wie eingebaute Funktionen, meist Argumente. Ihr Aufruf erfolgt über `Funktionsname([Argument(e)])` und sie liefern dann entweder berechnete Werte zurück oder verändern programminterne Daten. Ihre Definition erfolgt mithilfe des vorangestellten Schlüsselwortes `def`, und meist wird das Ergebnis mittels einer `return`-Anweisung zurückgeliefert, so dass die allgemeine Syntax – mit optionalen Elementen in eckigen Klammern – wie folgt aussieht.

```
def Funktionsname([Argument(e)]) :  
    [Anweisung(en)]  
    [return Variable/Ausdruck]
```

Variablen, die innerhalb einer Funktion definiert sind, sind rein **lokal**, das heißt, selbst wenn sie denselben Namen tragen wie global verwendete Programmvariablen, unterscheiden sie sich von diesen. Dies erleichtert unsere Arbeit, da wir uns bei Variablen, die den gleichen Zweck erfüllen, nicht jedes Mal einen anderen Namen ausdenken müssen, so dass wir beispielsweise die Variable `wort` oder `satz` sowohl für ein Wort oder einen Satz innerhalb als auch außerhalb, einer Funktion verwenden können. In selteneren Fällen müssen wir aber auch aus einer Funktion heraus auf **global** definierte Variablen zugreifen, was wir in Abschnitt 10.7 besprechen werden.

Äußerst wichtig ist, dass benutzerdefinierte Funktionen immer vor ihrem Aufruf im Programm definiert sein müssen! Vor allem, falls man vorher schon eine andere Programmiersprache erlernt hat, wo das nicht der Fall ist, kann das sehr verwirrend sein und anfänglich oft zu Fehlern führen.

7.4 Module verstehen

Module sind Container für Sammlungen von häufig benutzten Funktionen oder Klassen und beinhalten meist auch eigene Variablen. Dabei sind einfache Module auch nur

einfache Python-Dateien, die wie interne Module importiert werden können, allerdings ohne die Dateiendung `.py` anzugeben. Komplexere Module können aus mehreren Dateien und Verzeichnissen bestehen, die oft als Pakete installiert werden, z.B. mithilfe des **Python package installer** (`pip` oder `pip3`). Das Erstellen komplexer Module werden wir hier nicht besprechen.

Modulnamen werden normalerweise per Konvention klein geschrieben, so wie wir dies von den Modulen `sys`, `os` und `re` schon kennen. Häufig werden auch nur bestimmte Funktionen mittels

```
from Modulname import Funktionsliste
```

importiert, wobei aber auch alles über

```
from Modulname import *
```

oder, bei ausführbaren Modulen, einfach den Modulnamen importiert werden kann. Wie bei den uns schon bekannten Modulen kann man mit

```
Modulname.Funktionsname
```

auf Funktionen oder Methoden zuzugreifen. Beim Import ganzer Module gibt es auch die Möglichkeit, ein Alias für den Modulnamen mit

```
import Modulname as Alias
```

zu vergeben, z.B. falls der ursprüngliche Modulname sehr lang und umständlich einzutippen ist.

Im Folgenden wollen wir jetzt einmal ausprobieren, wie man ein eigenes, einfaches Modul entwickeln kann.

Übung 33 - Übersetzen, Teil 1

Schreiben Sie das Modul `uebersetzer.py`, durch das mithilfe mehrerer Funktionen, einem Wörterbuch und einer Liste von Sätzen, einfache Sätze Wort für Wort übersetzt werden können. Entscheiden Sie dabei selbst, basierend auf den folgenden Instruktionen, welche Module Sie importieren müssen.



Legen Sie dazu zunächst ein Wörterbuch `woerterbuch_de_en.txt` mit den Wörtern *das, ist, ein, eine, Satz, Buch, Blume, Zeitung, Frau, Mann* und deren englischen Übersetzungen, getrennt durch einen Doppelpunkt, an.

Erstellen Sie dann in der Datei `saetze_de.txt` eine Liste mit einfachen Deklarativsätzen, die diese Wörter auf Deutsch beinhalten, aber zudem auch ein paar Nomina, die nicht im Wörterbuch stehen.

Definieren Sie daraufhin im Modul die Funktion `woerterbuch_lesen`, die als Argument eine beliebige Wörterbuch-Datei übernimmt, die Datei zeilenweise abarbeitet, jeweils in deutsches und englisches Wort aufspaltet, diese in einem Dictionary als Schlüssel-Wert-Paar ablegt, und schließlich den Dictionary über eine `return`-Anweisung zurückliefert.

Achten Sie dabei auch auf eine geeignete Fehlerbehandlung, da bei falscher Angabe der Datei das Programm, welches später das Modul benutzt, abbrechen sollte.

Schreiben Sie danach die Funktion `saetze_lesen`, die eine Satzdatei einlesen und eine Liste von Sätzen zurückliefern soll.

Beim Einlesen beider Dateien sollten Sie auch zur Vorsicht dafür sorgen, dass keine Leerzeichen am Anfang oder Ende der jeweilige Zeilen stehen oder dass die Zeilen, die Sie abarbeiten, nicht leer sind.

Mit den Schritten in der obigen Übung haben wir nur dafür gesorgt, dass wir die entsprechende Funktionalität haben, um die vom Programm benötigten Ressourcen einzulesen. Allerdings fehlt dabei noch die Funktionalität zum Übersetzen und Ausgeben der Übersetzungen, die wir in der nächsten Übung entwickeln wollen. Das Einlesen von Ressourcen ist aber in der Praxis ein wichtiger Schritt bei vielen echten Analyseprogrammen, und die Deklaration von Daten in Programmen, die wir bisher benutzt haben, diente eigentlich nur der Verdeutlichung bestimmter Konstrukte.

Übung 34 - Übersetzen, Teil 2

Fügen Sie jetzt dem Programm eine Funktion `uebersetzen` hinzu, die ein Wörterbuch und einen Satz als Argumente übernimmt und eine Übersetzung zurückliefert. In der Funktion

- Legen Sie zuerst eine leere Liste für Wortübersetzungen an und verändern Sie den als Argument übergebenen Satz so, dass der erste Buchstabe verkleinert und das Satzzeichen abgeschnitten wird.
- Spalten Sie dann den Satz mittels `re.split()` wieder an Leerzeichen auf und iterieren Sie über die Wörter.
- Innerhalb der Schleife prüfen Sie zunächst, ob das jeweilige Wort im Wörterbuch existiert.



- Falls ja, hängen Sie mithilfe der `append()`-Methode dessen Übersetzung an die entsprechende Liste an.
- Falls nicht, hängen Sie `???` an, damit nicht im Lexikon stehende Wörter in der Übersetzung als solche gekennzeichnet sind.

Geben Sie daraufhin die mit Leerzeichen wieder verbundenen Wörter zurück, wobei Sie den ersten Buchstaben der Ergebniskette wieder vergrößern und das Satzzeichen ebenso anhängen.

Schreiben Sie zuletzt noch die Funktion `uebersetzung_ausgeben`, die als Argumente den Originalsatz und die Übersetzung mit geeigneten Anmerkungen ausgibt.

Erstellen Sie danach die Datei `20_uebersetzung.py`, in der Sie zunächst alle Funktionen aus dem Modul explizit importieren, dann das Wörterbuch und die Satzliste mittels der geeigneten Funktionen generieren und in einer Schleife alle Sätze mit Übersetzungen ausgeben.

7.5 Mit Modulen arbeiten

7.5.1 Module testen

Wie Sie wahrscheinlich schon bemerkt haben, ist es etwas umständlich, immer ein neues Programm zu schreiben, nur um die Funktionalität Ihrer eigenen Module zu testen. Dies ist insbesondere der Fall, wenn das Modul noch gar nicht fertig ist, sie aber schon vorher austesten wollen, ob bestimmte Teile auch tatsächlich so funktionieren, wie Sie es sich vorgestellt haben. Glücklicherweise bietet uns Python eine elegantere Möglichkeit, dies zu tun, indem wir einen zusätzlichen Block in unser Modul einbauen, der es direkt ausführbar macht. Da Module im Prinzip nichts anderes sind als Programme, ergibt das auch Sinn, da es uns erlaubt, sie je nach Bedarf entweder als eigenständige Programme oder als Programmteile zu verwenden, die spezielle Funktionalität in anderen Programmen bieten.

Um Module als Programme ausführbar zu machen, benötigen wir nur einen zusätzlichen Bedingungsblock am Modulende, der testet, ob das Modul als eigenständiges Programm ausgeführt wird. Dieser Block hat die folgende Form.

```
if __name__ == '__main__':
    Anweisung(en) zum Testen
```

Hier können wir nach Belieben alle schon implementierten Programmteile ausprobieren und uns die Ergebnisse durch `print`-Anweisungen anzeigen lassen, wobei wir auch auf `import`-Anweisungen verzichten können, da wir uns ja in dem Programm befinden, in dem die benutzerdefinierten Funktionen oder Methoden implementiert wurden. Wir werden diese Option ab jetzt öfter verwenden, insbesondere wenn wir später mit grafischen Benutzerschnittstellen arbeiten.

7.5.2 Externe Module installieren

Jede Python-Distribution enthält schon einige nützliche Module wie die, die wir bisher importiert und verwendet haben. Jedoch gibt es noch eine sehr große Menge an speziellen Modulen für unterschiedliche Zwecke, die andere Entwickler über den **Python Package Index** (PyPI) (<https://pypi.org/>) oder andere Code-Repositorien zur Verfügung stellen. Diese Pakete können meist mithilfe des Python package installers (`pip`) über die Kommandozeile installiert werden. Bei alleiniger Installation von Version 3 kann dieser einfach über das Kommando `pip` aufgerufen werden, bei paralleler Installation von Version 2 und 3 muss aber explizit der Aufruf `pip3` verwendet werden, so dass die vollständige Syntax für Benutzer mit vollen (Administrator-)Berechtigungen

```
pip(3) install Modulpaket
```

ist. Sollten Ihnen Ihre Berechtigungen normalerweise nicht erlauben, Module zu installieren, so können Sie diese dennoch, rein für sich selbst, als Nutzer ohne spezielle Berechtigungen über

```
pip(3) install --user Modulpaket
```

installieren. Für mehr Details, wie z.B. Installation heruntergeladener Module, können Sie die Seite <https://packaging.python.org/tutorials/installing-packages/#installing-from-pypi> konsultieren.

Übung 35 - PyQt installieren

Gehen Sie auf die PyPI-Seite (<https://pypi.org/>) und suchen Sie nach `PyQt5`. Wir werden dieses Modul später zur Erstellung von grafischen Benutzeroberflächen verwenden, wollen es aber jetzt schon installieren.

Folgen Sie dem Link zu dem Paket unter dem *Python bindings for the Qt cross platform UI and application toolkit* steht.



Lesen Sie sich die Beschreibung durch und suchen Sie nach dem passenden `pip`-Befehl zur Installation.

Kopieren Sie diesen und verwenden Sie die Kommandozeile, um das Paket zu installieren, falls nötig nur als Benutzer.

7.6 Klassen und Objekte

Objekte sind Behälter für Daten, die eigene Variablen beinhalten und auch Methoden zur Verarbeitung ihrer Argumente/Daten zur Verfügung stellen. Methoden stellen dabei die Schnittstelle zum Anwender dar, der idealerweise nichts über die interne Datenstruktur oder Verarbeitung selbst wissen muss, um das Objekt sinnvoll anwenden zu können. Sie sind in Python als Klassen definiert, die sozusagen ihren Bauplan darstellen, und erst als konkrete Objekte instanziiert werden, wenn man tatsächlich Objektvariablen anlegt.

Neue Objekte werden, wie bei einigen Funktionen zur Erzeugung existierender Datentypen, wie z.B. Listen oder Tupel, über

```
Objektvariablenname = Objektname([Argumente])
```

erzeugt. Wiederum per Konvention beginnen üblicherweise benutzerdefinierte Klassennamen mit Großbuchstaben, um sie von Modulnamen zu unterscheiden.

Die Form von Klassendefinitionen ist ähnlich der von benutzerdefinierten Funktionen. Nur beginnen sie mit dem Schlüsselwort `class` anstelle von `def`, gefolgt von einem Klassennamen. Optional können in runden Klammern mögliche Elternklassen angegeben werden, von den die Klasse abgeleitet ist. Der abschließende Doppelpunkt leitet, wie üblich, einen Block ein, in dem alle Variablen und Methoden der Klasse definiert werden.

Es gibt bei Objekten zwei Typen von Variablen: Klassen- und Instanzvariablen. Klassenvariablen gelten für alle Objekte der Klasse gemeinsam und werden z.B. für Zähler aller kreierte Objekte einer Klasse verwendet. Da wir sie aber für unsere Zwecke nicht benötigen, werden sie hier nicht genauer besprochen. Wichtiger sind für uns die Instanzvariablen, die zwar denselben Variablennamen für jedes Objekt tragen, aber immer nur für das jeweilige Objekt gelten, so dass sie auch immer verschiedene Werte haben können. Diese Werte stellen normalerweise die Eigenschaften – oder **Attribute** – des Objekts dar, und werden mit

```
self.Variablenname
```


deklariert. Auf diese Eigenschaften kann man zwar auch von außen direkt zugreifen, aber idealerweise sollten eigentlich nur Methoden des Objekts selbst sie verändern oder zurückliefern, um eine konsistente Schnittstelle für Anwender zu bieten.

Wichtig ist, wie oben schon angesprochen, dass eine (Modul-)Datei mehrere Klassendefinitionen enthalten kann, vor allem dann, wenn diese zusammengehörige Funktionalitäten liefern oder ähnliche Kategorien darstellen. So z.B. bietet es sich an, alle Wortarten innerhalb eines Wort-Objektes zu implementieren.

7.6.1 Methoden

Die Methoden einer Klasse werden, ebenso wie normale Funktionen, mit

```
def Methodenname(self):
```

und einem darauffolgenden Block definiert. Allerdings muss bei Methodendefinition als erstes Argument immer die Referenz `self` auf die Klasse übergeben werden, damit die Klasse auch ‚weiß‘, dass es sich um eine zu ihr gehörige Methode handelt. Beim Aufruf der Methode allerdings ist dieses Argument implizit, also nicht mehr anzugeben!

Klassen beinhalten meist als erste Methode eine **Initialisierungsmethode**

```
__init__(self, [Argument(e)])
```

die als sogenannter **Konstruktor** zum Aufbau der Klasse dient, also wichtige Eigenschaften der Klasse gleich beim Anlegen initialisiert. Bei Objektinstanziierung übergebene Schlüsselwortargumente werden normalerweise zum Initialisieren von Instanzvariablen mit gleichem Namen verwendet, also

```
self.Argumentvariable = Argumentvariable
```

Letzteres eignet sich auch zum Setzen von Standardwerten über

```
Argumentvariable = Wert
```

so dass man beim Initialisieren der Klasse diese Argumente nicht mehr angeben muss, sondern nur dann einen Wert für das Argument übergeben, falls dieser vom Standard abweicht. Wir kennen solche Standardwerte schon von Funktionen wie `print()`,

wo implizit immer das Schlüsselwortargument `end` mit dem Wert `'\n'` verwendet wird, falls nichts Gegenteiliges angegeben wird. Die `__init__()`-Methode, insofern sie definiert ist, wird automatisch beim Anlegen eines neuen Objekts aufgerufen.

7.6.2 Klassenschema

Die meisten Klassendefinitionen folgen einem ähnlichen Schema, so dass es nützlich ist, sich diesen Aufbau nochmals schematisch zu verdeutlichen.

```
class Klassenname[ ]:
    def __init__(self, [Argument(e)]):
        self.variableX = argumentX
        self.variableY = argumentY ...

    def Methode1(self, [Argument(e)]):
        Methodendefinition

    def Methode2(self, [Argument(e)]):
        ...
```

Die Anzahl der benötigten Instanzvariablen und Methoden variiert dabei je nach Zweck des Objekts, also welche Eigenschaften und Methoden es zur Verfügung stellen muss. Innerhalb der `__init__()`-Methode können natürlich nicht nur Argumente an Instanzvariablen übergeben werden, sondern auch Anweisungen stehen, die daraus wiederum andere Variablenwerte erzeugen.

In der folgenden Übung werden wir versuchen, ein größeres, linguistisch relevantes, Objekt selbst ansatzweise zu erstellen, wobei Sie stark auf Ihr eigenes Wissen zur Flexion des Deutschen zurückgreifen müssen. Dies wird schon wesentlich mehr Zeit und Aufwand erfordern als die meisten anderen Programme, die wir bisher geschrieben haben, aber Ihnen dadurch auch einen besseren Eindruck davon vermitteln, wie komplex es sein kann, natürliche Sprache zu analysieren oder zu modellieren.



Übung 36 - Ein Wortobjekt schreiben

Legen Sie eine neue Datei `wort.py` an, in der Sie nur eine sehr unvollständig implementierte Verbklasse anlegen werden.

Importieren Sie zunächst das `re`-Modul, da wir dies später benötigen werden.

Legen Sie dann die Klasse `Verb` mit einem `__init__()`-Konstruktor an.

Innerhalb des Konstruktors sollen Schlüsselwortargumente für Infinitiv, Person, Formalitätsgrad (informell/formell), Numerus, Tempus, Modus und Typ (regu-

lär/irregulär) übergeben werden können, wobei Sie auch gleichzeitig geeignete Standardwerte für einige der Argumente anlegen sollten.

Da die Klasse ohne ein Infinitivargument nie richtig initialisiert werden kann, soll, falls dieses Argument fehlt, ein `NameError` mit entsprechender Fehlermeldung nach oben an das sie verwendende Programm weitergeleitet werden.

Initialisieren Sie innerhalb des Konstruktors entsprechende Instanzvariablen mittels der Argumente und legen Sie auch eine zusätzliche Instanzvariable an, der der Verbstamm, basierend auf dem Infinitiv, zugewiesen wird.

Implementieren Sie noch zwei Methoden, mit denen die Partizipien 1 und 2 gebildet werden können.

Um die Klasse testen zu können, fügen Sie auch einen Block zum Testen hinzu, so wie dies in 7.5 beschrieben ist, und testen Sie die Funktionalität mithilfe selbstgewählter Verben.

Versuchen Sie zusätzlich die Präsens-Indikativbildung für reguläre Verben zu modellieren.

Nachdem wir in diesem Kapitel gelernt haben, wie wir unsere Programme effizient in wiederverwendbare, aber durchaus auch eigenständige, Module zerlegen können, wollen wir uns im nächsten Kapitel Methodiken der quantitativen Analyse zuwenden.

7.7 Lösungen zu den Aufgaben

Lösung 33 - Übersetzen, Teil 1

Da wir beim Einlesen der Dateien etwaig auftretende Fehler behandeln müssen, sollte zunächst unbedingt das `sys`-Modul importiert werden, damit wir im Notfall wieder mit `sys.exit()` das Programm abbrechen können.

Die beiden Dateien können Sie entweder in einem externen Editor oder direkt in der WingIDE erstellen, insbesondere da Letztere ja schon so eingestellt sein sollte, dass Dateien automatisch als UTF-8 kodiert abgespeichert werden. Wie genau Ihre Sätze in der Datei aussehen, die die Deklarativsätze beinhalten soll, können Sie diesmal selbst entscheiden, solange die Wörter aus dem Wörterbuch und die zusätzlichen, unbekannten Nomina darin vorkommen.

Die Definition der Funktion `woerterbuch_lesen` muss mit dem Schlüsselwort `def` eingeleitet werden und in runden Klammern hinter dem Funktionsnamen einen sprechenden Variablennamen für das Argument selbst beinhalten. Hierfür verwenden wir am besten etwas wie `woerterbuch_datei`, so dass man beim Lesen der Funktion sofort erkennen kann, was als Argument übergeben werden soll. Außerdem wird Ihnen später beim Aufruf der Funktion in einem Programm auch jede gute IDE diesen Argumentnamen anzeigen. Innerhalb der Funktion selbst müssen Sie zunächst einen



leeren Dictionary anlegen, um ihn später mit Wort-Übersetzungspaaren zu befüllen. Zur Fehlerbehandlung sollte dahinter selbstverständlich ein `try`-Block stehen, in dem Sie versuchen, die Datei zu öffnen, und das Programm bei einem Fehler abbrechen, da es ja überhaupt nicht funktionieren kann, wenn wir keine Übersetzungspaare zur Verfügung stehen haben.

Zum Abarbeiten der Zeilen in der Datei sollten Sie dann eine `for`-Schleife verwenden, in der Sie zunächst mit der `strip()`-Methode alle potenziellen Leerzeichen an Anfang oder Ende entfernen und dann mittels einer `if`-Anweisung überprüfen, ob überhaupt noch etwas in der Zeile steht. Falls das Ergebnis `True` ist, können Sie die Zeile mittels `split(':')` in ein Tupel aus deutschem Wort und englischer Übersetzung aufspalten und dann dem Dictionary das Wort als Schlüssel mit der jeweiligen Übersetzung als dessen Wert zuweisen. Letztendlich muss nur noch der Dictionary mit `return` zurückgeliefert werden, so dass die gesamte Funktion in etwa so aussehen sollte.

```
def woerterbuch_lesen(woerterbuch_datei):
    woerterbuch = {}
    try:
        with open(woerterbuch_datei, 'r', encoding='utf-8') as
            wb:
                for zeile in wb:
                    zeile = zeile.strip()
                    if zeile:
                        wort, uebersetzung = zeile.split(':')
                        woerterbuch[word] = uebersetzung
    except OSError as fehler:
        sys.exit(str(fehler))
    return woerterbuch
```

Die Funktion `saetze_lesen` ist sehr ähnlich aufgebaut, nur dass wir hier zuerst eine leere Liste für die Sätze anlegen, die Zeilen dann nicht aufspalten, sondern mittels `append()` direkt in die Liste einfügen und dann diese zurückliefern. Die Funktion selbst sollte dann so aussehen.

```
def saetze_lesen(satz_datei):
    saetze = []
    try:
        with open(satz_datei, 'r', encoding='utf-8') as sd:
            for zeile in sd:
                zeile = zeile.strip()
                if zeile:
                    saetze.append(zeile)
    except OSError as fehler:
```

```

    sys.exit(str(fehler))
return saetze

```

Lösung 34 - Übersetzen, Teil 2

Das Anlegen der neuen Funktionsdefinition und der leeren Liste dürfte jetzt nicht mehr weiter schwierig für Sie sein. Als Variablennamen für die Argumente bieten sich hier `woerterbuch` und `satz` an. Um den Satz in die passende Form zu bringen, sollten Sie am einfachsten mit zwei Slices arbeiten, einem für den ersten Buchstaben des Satzes, den Sie dann gleich mit `lower()` verkleinern können und einem mit dem Rest ohne das Satzzeichen.

Das Aufspalten des jeweiligen Satzes in Wörter dürfte nicht sonderlich schwer sein, da die `re.split()`-Methode der gleichnamigen Methode von Zeichenketten sehr ähnlich ist, nur dass sie viel flexibler ist, weil Sie ja dort Regex-Muster verwendet können. Da wir jedoch in unserem Fall davon ausgehen, dass alle Sätze wohlgeformt sind, indem nur ein Leerzeichen zwischen allen Wörtern steht, was bei Daten, die wir nicht kennen, durchaus nicht immer der Fall sein muss, reicht uns hier ein einfaches `\s` innerhalb einer rohen Zeichenkette. Theoretisch könnten Sie hier zunächst eine temporäre Liste über `re.split()` erzeugen und dann im nächsten Schritt über diese iterieren, aber effizienter ist es natürlich, die Methode direkt in einer `for`-Schleife anzuwenden, so wie wir dies schon vorher in der Übung zum Löschen des Infixes getan haben.

Zur Überprüfung des Vorkommens des Wortes im Wörterbuch kommt der `in`-Operator zum Einsatz. Liefert die Bedingungsabfrage `True` zurück, benutzen wir wieder `append()`, um die passende Übersetzung an die Liste mit Übersetzungen anzuhängen und sonst die Zeichenkette `'???'`, um anzuzeigen, dass es sich hier um ein unbekanntes Wort handelt. Hier nichts anzuhängen wäre später bei der Ausgabe irreführend, da ja sonst einfach ohne Markierung etwas fehlen würde.

Zur Rückgabe der Übersetzung müssen Sie dann nur die Elemente der Übersetzungsliste miteinander über Leerzeichen verknüpfen, wozu Sie mit `join()` arbeiten sollten. Beim Ergebnis der Verknüpfung sollten Sie aber gleich mittels `capitalize()` den ersten Buchstaben, also den Satzanfang, wieder vergrößern und den abschließenden Punkt wieder durch Konkatenation anfügen. All dies kann selbstverständlich direkt in der `return`-Anweisung geschehen, da wir ja gelernt haben, dass in dieser Art von Anweisung nicht nur Werte sondern auch ganze Ausdrücke verwendet werden können, so dass wir keine Hilfsvariable benötigen. Die komplette Funktion sollte dann in etwa so aussehen:

```

def uebersetzen(woerterbuch, satz):
    uebersetzungen = list()
    satz = satz[0:1].lower() + satz[1:-1]
    for wort in re.split(r'\s', satz):
        if wort in woerterbuch:

```

```

        uebersetzungen.append(woerterbuch[word])
    else:
        uebersetzungen.append('???')
    return ' '.join(uebersetzungen).capitalize() + '.'

```

Die letzte Funktion, `uebersetzung_ausgeben`, ist nicht nötig, da wir die Übersetzung auch direkt mittels der vorherigen Routine generieren und ausgeben können. Allerdings kann sie dazu dienen, eine sinnvoll kommentierte Ausgabe zu generieren, die immer die von uns erwartete Form hat, selbst wenn jemand anderes unser Modul verwendet. Außerdem lässt sie sich leicht anpassen, falls einmal später ein anderes Ausgabeformat gefordert wäre. Sie sollte sehr einfach zu schreiben sein, da nur zwei Argumente übergeben und eine simple Ausgabe erzeugt werden müssen. Diese Funktion könnte ungefähr so aussehen.

```

def uebersetzung_ausgeben(satz, uebersetzung):
    print(f'Original: {satz}\tÜbersetzung: {uebersetzung}')

```

Wie Sie sehen können, werden hier alle Erklärungen, der Originalsatz sowie die Übersetzung auf einer Zeile ausgegeben, wobei der Übersetzungsteil zu besseren Übersichtlichkeit mittels eines Tabulators etwas nach rechts verschoben wird. Allerdings ist dies nicht in allen Fällen deutlich zu sehen, wenn der Originalsatz schon in der Nähe der nächsten Tabulatorposition endet. Falls wir später ein noch übersichtlicheres Ausgabeformat erzeugen wollen, so können wir die `print`-Anweisung der Ausgabefunktion einfach zu

```

print('\n'.join(['Original:', satz, 'Übersetzung:',
    uebersetzung]))

```

umschreiben, wodurch alle Teile untereinander ausgegeben werden.

Unser Programm, welches das Modul verwendet, sollte dann wie folgt aussehen.

```

from uebersetzer import woerterbuch_lesen, saetze_lesen, uebersetzen,
uebersetzung_ausgeben

```

```

wb = woerterbuch_lesen('./woerterbuch_de_en.txt')
saetze = saetze_lesen('./saetze_de.txt')

```

```

for satz in saetze:
    uebersetzung_ausgeben(satz, uebersetzen(wb, satz))

```

Hier sind die einzelnen Schritte getrennt verwendet. Dabei benötigen wir die jeweiligen Hilfsvariablen, die in den Zwischenschritten erzeugt werden, nicht und könnten das Programm stattdessen kompakter so schreiben:

```

from uebersetzer import (woerterbuch_lesen, saetze_lesen,
    uebersetzen, uebersetzung_ausgeben)

```

```
wb = woerterbuch_lesen('./woerterbuch_de_en.txt')

for satz in saetze_lesen('./saetze_de.txt'):
    uebersetzung_ausgeben(satz, uebersetzen(wb, satz))
```

Natürlich könnten wir das Programm auch noch kompakter schreiben, indem wir die Wörterbuchgenerierung ebenfalls innerhalb der Schleife in der Funktion zum Ausgeben durchführen. Allerdings wäre das weniger Effizient, da dadurch das Wörterbuch jedes Mal erneut eingelesen wird, wenn ein Satz übersetzt werden soll. Dies verdeutlicht hoffentlich, dass man sich immer genau bewusst sein sollte, was die einzelnen Funktionen tatsächlich tun, und ob es sich wirklich lohnt, kürzeren Code zu produzieren.

Lösung 35 - PyQt installieren

Insofern Python im Pfad eingetragen ist und Sie eine halbwegs stabile Internetverbindung haben, dürfte die Installation über

```
pip(3) install PyQt5
```

oder

```
pip(3) install -user PyQt5
```

keine Probleme für Sie bereiten. Falls `pip` nicht gefunden wird, so müssen Sie unter Umständen den in Kapitel 1 beschriebenen Schritten folgen, um Python dem Pfad hinzuzufügen, und es dann noch einmal probieren.

Lösung 36 - Ein Wortobjekt schreiben

Wie sie sich hoffentlich schon beim Lesen des Namens gedacht haben, ist dieses Modul eigentlich für Größeres gedacht, nämlich dafür, dass Sie darin Python-Klassen für alle möglichen Wortarten entwickeln können. Allerdings wäre das komplette Projekt viel zu groß für unseren sehr beschränkten Rahmen. Deshalb werden wir nur einen relativ kleinen Teil dieses Projekts angehen.

Diese Übung ist im Vergleich zu den bisherigen etwas anders, da ich hier weniger Instruktionen für die Implementierung einzelner Programmschritte gebe, sondern mich zum großen Teil darauf verlasse, dass Sie ein ausreichendes Wissen über die deutsche Verbmorphologie haben oder sich dies aneignen können. Zudem sollten Sie auch genügend Zeit damit verbringen, Ihre Klasse mit verschiedenen Verben zu testen.

Der Import des `re`-Moduls und das Anlegen der Klasse selbst dürfte Ihnen keine Schwierigkeiten bereiten. Für die Variablennamen der Schlüsselwortargumente in der `__init__`-Methode bieten sich die kleingeschriebenen Namen der Verbeigenschaften an, also `infinitiv`, `person`, `form_grad` (für den Formalitätsgrad), `numerus`, `tempus`, `modus`, und `typ`, wobei letzterer zwischen regulären und irregulären Verben unterscheiden soll. Allerdings dürfen Sie dabei nicht vergessen, dass das erste Argument

einer Methode in deren Definition immer `self` sein muss. Sonst tritt ein `TypeError` auf, weil an dieser Stelle die Referenz auf das Objekt selbst erwartet wird und kein Schlüsselwortargument. Als Standardwert für `infinitiv` sollten wir `None` verwenden, da ja die Infinitivform erst übergeben wird und somit nicht vorher feststehen kann, für person '1', numerus 'singular', tempus 'praesens', modus 'indikativ', und typ 'r', da wir davon ausgehen, dass die meisten Verben reguläre sind.

Innerhalb des `__init__`-Blocks sollte zuerst überprüft werden, ob beim Anlegen des Objekts kein Infinitiv übergeben wurde, also der Wert des Schlüsselwortarguments immer noch `None` ist. Falls ja, dann sollte der `NameError` mittels `raise` an das Programm zurückgeliefert werden, welches die Klasse verwendet, um dort eine geeignete Fehlerbehandlung durchführen zu können. Ansonsten können nach der Bedingungsabfrage einfach die Werte der Schlüsselwortargumente an Instanzvariablen, z.B. über

```
self.infinitiv = infinitiv
```

übergeben werden, und zuletzt noch der Stamm aus dem Infinitiv extrahiert und der Instanzvariable `self.stamm` zugewiesen werden. Bei diesem letzten Schritt müssen wir allerdings testen, ob nicht der vorletzte Buchstabe des Infinitivs ein Liquid ist, da dann nur das finale *n* abgeschnitten werden muss, um den Stamm zu erhalten. Ansonsten müssen immer die letzten zwei Buchstaben bei der Extraktion mittels eines Slice abgeschnitten werden. Dieser Test lässt sich am einfachsten mittels der Regex

```
re.search(r'[lr]', self.infinitiv[-2:-1])
```

durchführen. Der vollständige Bedingungsblock sollte also folgendermaßen aussehen.

```
if not re.search(r'[lr]', self.infinitiv[-2:-1]):
    self.stamm = self.infinitiv[:-2]
else:
    self.stamm = self.infinitiv[:-1]
```

Der komplette Konstruktor sollte dann in etwa so aussehen, wobei Sie auch auf die Einrückung innerhalb der Klassendefinition achten müssen, da die Methodendefinitionen der Klassendefinition untergeordnet sind. Der Übersichtlichkeit halber, und um Platz zu sparen, verzichten wir hier bei der Illustration der Methoden auf die zusätzliche Einrückung.

```
def __init__(self, infinitiv=None, person='1', form_grad=
    'informell', numerus='singular', tempus='praesens',
    modus='indikativ', typ='r'):
    if not infinitiv:
        raise NameError('Keine Infinitivform angegeben!')
    self.infinitiv = infinitiv
    self.person = person
    self.form = form
```



```

self.numerus = numerus
self.tempus = tempus
self.typ = typ
if not re.search(r'[lr]',self.infinitiv[-2:-1]):
    self.stamm = self.infinitiv[:-2]
else:
    # Liquid vor Infinitiv-Endung
    self.stamm = self.infinitiv[:-1]

```

Vergessen Sie dabei wieder nicht, dass in der Methodendefinition als erstes Argument `self` übergeben werden muss, da das Objekt beim Aufruf der Methode sonst nicht ‚verstehen‘ kann, dass es sich um eine Methode handelt.

Die Implementierung der Methode, um das Partizip 1 zu bilden, ist im Prinzip sehr einfach, da immer nur ein <d> an den Infinitiv angehängt werden und der Infinitiv immer beim Anlegen des Objektes angegeben werden muss. Diese Methode könnte also so geschrieben werden:

```

def partizip1(self):
    return self.infinitiv + 'd'

```

Ebenso wie im Konstruktor muss hier wieder als Argument `self` übergeben werden. Die Bildung des Partizip 2 ist etwas schwieriger, da wir, je nach Auslaut des Stammes, neben dem Präfix {ge} auch eine passende Flexionsendung anfügen müssen, entweder {t}, {et}, oder {en}. Da wir hier nur reguläre Verben behandeln, müssen wir zunächst erst einmal eine Bedingung einfügen, um dies zu testen, wobei wir aber zunächst auf die Angabe von Alternativen verzichten, da wir noch keine irregulären Fälle behandeln wollen. Um die graphemischen Realisierungen der Auslaute des Stammes abdecken zu können, müssen wir teilweise etwas komplexere reguläre Ausdrücke zur Modellierung der Partizipialformen anwenden. Grob gesagt, müssen wir zunächst beachten, ob der Stamm entweder aus phonologischer Sicht auf einen Frikativ, einen Langvokal oder Schwa vor <n>, velare Nichtnasale, <l> oder <m>, oder aber einen verdoppelten Konsonanten <nn>, <mm>, oder <ss> endet. In diesem Fall müssen wir als Flexionssuffix ein {t} anhängen. Im zweiten Fall müssen wir testen, ob am Stammende ein alveolarer Plosiv, also <d>, <t> oder <n> (ohne die vorher schon erfassten Formen), auftritt und gegebenenfalls {et} anhängen, wobei in allen anderen Fällen immer {en} angefügt werden kann. Die vollständige Definition der Methode sieht dann ungefähr folgendermaßen aus.

```

def partizip2(self):
    if self.typ=='r':
        # finaler Frikativ, Langvokal, Liquid,
        # nicht-alveolarer Plosiv, oder Doppelkonsonant
        # am Stammende

```

```

if re.search(r'(?c?h|(?ie|(?:[aoäöü]h|r)n)'
            r'|[pbmfgklnsz])$', self.stamm):
    return 'ge' + self.stamm + 't'
elif re.search(r'(?[dtn])$', self.stamm):
    return 'ge' + self.stamm + 'et'
else:
    return 'ge' + self.stamm + 'en'

```

Bei der Auflistung der möglichen Vokale dürfen wir die Umlaute nicht vergessen. Die Kombination `c?h` in der Regex für den ersten Fall deckt entweder ein für sich allein stehendes `<h>` oder auch `<ch>` und `<sch>` ab. Ansonsten dürfte es beim Verstehen der Ausdrücke eigentlich keine größeren Probleme geben. Ein Problem existiert jedoch noch bei unserer momentanen Implementierung der regelmäßigen Formen, da bei präfigierten Verben, wie z.B. *ermahnen*, übergeneriert wird, da dort kein Präfix `{ge}` eingefügt werden darf. Dies müsste unter Umständen durch ein weiteres Attribut in Verbindung mit zusätzlichen Bedingungen gelöst werden, durch die das Präfix unterdrückt werden könnte.

Wie Sie den Testblock gestalten, bleibt Ihnen überlassen, aber auf jeden Fall müssen Sie ihn erst einmal mit `if __name__ == '__main__':` auf der richtigen ‚Einrückungsstufe‘, also nicht innerhalb der Klassendefinition, beginnen, da es sich um einen Teil des Moduls und nicht der Klasse handelt. Danach bietet es sich an, erst einmal ein Verbojekt anzulegen, mit dessen Hilfe dann die gewünschten Formen zur Ausgabe generiert werden können. Da wir auf jeden Fall zum Anlegen des Objekts einen Infinitiv benötigen, könnten wir diesen z.B. über die Kommandozeile einlesen, eventuell auch zusammen mit anderen Schlüsselwortargumenten. Zu diesem Zweck müsste dann das `sys`-Modul zu Beginn des Programms importiert werden. Andere Möglichkeiten wären, das Objekt innerhalb des Programms immer wieder bezüglich der verschiedenen Argumente ‚manuell‘ anzupassen und dann das Programm aus der IDE heraus laufen zu lassen, oder Verben mit verschiedenen Stammauslauten in einer Liste zu speichern und diese mittels einer Schleife abzuarbeiten. Die hier – nur zur Vollständigkeit – gezeigte Lösung, verwendet die Listenversion fügt der Übersicht halber auch vor den Indikativformen des Verbs.

```

if __name__ == '__main__':
    for inf in ['fassen', 'duschen', 'lächeln', 'füttern',
               'flüstern', 'reden', 'trennen', 'prüfen']:
        verb = Verb(infinitiv=inf,
                    person='1',
                    form_grad='formell',
                    numerus='plural')
        print(f'Partizip 1: {verb.partizip1()} , '
              f'Partizip 2: {verb.partizip2()}')

```

```

if verb.person == '1':
    if verb.numerus == 'singular':
        pronomen = 'ich'
    else:
        pronomen = 'wir'
elif verb.person == '3':
    if verb.numerus == 'singular':
        pronomen = 'er/sie/es'
    else:
        pronomen = 'sie'
else:
    if verb.numerus == 'singular' and \
        verb.form_grad == 'informell':
        pronomen = 'Du'
    else:
        pronomen = 'Sie'
print(f'Stamm: {verb.stamm}; Präsens Indikativ für '
      f'{verb.person}. Person {verb.numerus.capitalize()}, '
      f'Formalitätsgrad {verb.form_grad}: {pronomen} '
      f'{verb.praesens()}')

```

Bei dieser Lösung sollte schon die Präsens-Indikativbildung implementiert sein, die wir im Folgenden besprechen werden, da sonst der Aufruf der `praesens`-Methode oben einen Fehler hervorruft. Die Ausgabe des Stammes dient nur zum Debuggen, damit man sehen kann, wie sich die Form des Stammes auf die einzelnen Flexionsarten auswirkt. Der Übersicht halber wird auch vor den Indikativformen des Verbs jeweils das passende Pronomen angezeigt, weshalb der Testblock etwas länger ausfällt.

Bei der Implementierung der `praesens`-Methode müssen wir zunächst prüfen, ob es sich bei unserem Verbojekt um ein reguläres Verb handelt, und falls nicht, idealerweise eine Meldung ausgeben, um dem Benutzer anzuzeigen, dass irreguläre Formen noch nicht modelliert werden können. Danach können wir die verschiedenen Präsens-Indikativformen implementieren. Hier müssen wir zunächst eine Unterscheidung nach Singular und Plural treffen. Innerhalb des Singulars muss dann zwischen den drei Personen unterschieden werden. Die erste Person ist am einfachsten, da immer nur ein {e} an den Stamm angehängt werden muss. Bei der zweiten Person wird es komplizierter. Hier muss nicht nur zwischen informellem und formellem Formalitätsgrad unterschieden werden, sondern bei der informellen Form auch noch, je nach Stammauslaut, entweder {est}, {st} oder {t} angehängt werden. Bei der formellen Form kann an den Stamm immer ein {en} angehängt werden, es sei denn der Stamm endet auf einen Liquid, so dass nur ein {n} angefügt werden darf. Zuletzt muss bei der dritten Person, basierend auf dem Stammauslaut, entweder noch {et} oder {t} angefügt werden.

Im Plural müssen wir nur testen, ob wir eine Verbform in der 2. Person mit informellem Formalitätsgrad generieren wollen, da bei allen anderen Personen immer nur mit {en}, bzw. {n} suffigiert wird. Liegt die entsprechende informelle Form vor, können wir denselben regulären Ausdruck wie beim Partizip 2 anwenden, um zu testen, ob nur ein {t} angehängt werden soll oder ein {et}. Die komplette Methode sollte dann in etwa so aussehen.

```
def praesens(self):
    if self.typ=='r':
        if self.numerus=='singular':
            if self.person=='1':
                return self.stamm + 'e'
            elif self.person=='2':
                if self.form_grad=='informell':
                    if re.search(r'[mndt]',
                                self.infinitiv[-3:-2]) \
                        and not re.search(r'[nm]{2}$', self.stamm):
                        return self.stamm + 'est'
                    else:
                        if re.search(r's$', self.stamm):
                            return self.stamm + 't'
                        else:
                            return self.stamm + 'st'
                else:
                    if re.search(r'[rl]$', self.stamm):
                        return self.stamm + 'n'
                    else:
                        return self.stamm + 'en'
            else:
                if re.search(r'[mndt]',
                            self.infinitiv[-3:-2]) \
                    and not re.search(r'[nms]{2}$', self.stamm):
                    return self.stamm + 'et'
                else:
                    return self.stamm + 't'
        if self.numerus=='plural':
            if self.person=='2' and \
                self.form_grad=='informell':
                if re.search(
                    r'(?c?h|(?ie|(?[aoäöü]h|r)n))$'
                    r'|[pbmfgklnsz]$', self.stamm):
                    return self.stamm + 't'
```

```
        else:
            return self.stamm + 'et'
    else:
        if re.search(r'[rl]$', self.stamm):
            return self.stamm + 'n'
        else:
            return self.stamm + 'en'
    else:
        return 'Irreguläres Verb. Noch nicht implementiert.'
```

Zum Testen können Sie z.B. die Infinitive verwenden, die ich in meinem Testblock verwendet hatte, aber natürlich auch eigene wählen.

Wie Sie durch diese Übung gesehen haben, können sprachliche Objekte sehr kompliziert sein, insbesondere da wir nur einen relativ kleinen Teil der Verbmorphologie behandelt haben. Allerdings müssen nicht alle Objekte, die wir in der Sprachwissenschaft verwenden, so kompliziert sein.

8 Wortlisten, Frequenzen und Grundlagen der Sortierung



Sprachliche Phänomene zu quantifizieren ist ein wesentlicher Bestandteil von Sprachanalysen. Deshalb soll in diesem Kapitel besprochen werden, wie verschiedene Arten von Listen solcher Phänomene, von einfachen Wortlisten bis zu Frequenzlisten für Wörter oder N-gramme, mit Python erstellt werden können. Da solche Listen für verschiedene Zwecke in der jeweils sinnvollsten Form ausgegeben werden müssen, werden hier einige wichtige Wege besprochen, wie man dies durch verschiedene Sortieroptionen erreichen kann. Wir beginnen unsere Diskussion mit einfachen Wortlisten, zunächst noch ohne Frequenzen.

8.1 Wortlisten

Wort-, aber auch N-Grammlisten, also Listen von jeweils mehreren Wörtern in Folge, sind ein essenzieller Bestandteil lexikographischer, korpuslinguistischer und computerlinguistischer Arbeit. So z.B. kann man durch die Generierung solcher Listen Vokabellisten für den schulischen oder eigenen Lernbedarf erstellen, den Wortschatz einer Sprache erfassen, phraseologische oder idiomatische Strukturen entdecken oder sogar Gesprächsthemen erkennen.

Methodisch basieren solche Listen – zumindest theoretisch – immer auf der festen Abfolge bestimmter Schritte. Man beginnt mit der Zerlegung von Texten in einzelne Wörter oder N-Gramme, den sogenannten **Tokens**, welche die einzelnen zählbaren Formen darstellen. Da jedes Token – wie z.B. die Wortform *das* oder das Bigramm *zu Hause* – mehrfach im Text auftreten kann, muss man zählen, wie oft jede einzelne Tokenform auftritt und eine neue Liste erzeugen, in der jede Form, der sogenannte **Type**, nur einmal erscheint, wobei jedem dieser Types wiederum die Anzahl der einzelnen Tokens zugeordnet ist, z.B. *das*: 235 oder *zu Hause* 23. Zuletzt muss die so erzeugte Liste noch sortiert und in geeigneter Form ausgegeben werden.

Reine Wortlisten – d.h. Listen, die nur Types ohne deren Häufigkeiten beinhalten – sind meist weniger informativ als Frequenzlisten, da die Häufigkeiten, zumindest bei Inhaltswörtern oder phraseologischen N-Grammen, auf wichtige Sprachfunktionen hindeuten. Sie können theoretisch als einfache Listen generiert werden, was jedoch in vielen Programmiersprachen, wie auch in Python, nicht die effizienteste Methode darstellt. Deshalb verwenden wir normalerweise Dictionaries dafür. Frequenzlisten hingegen sind informativer und repräsentativer, insbesondere da man sie nicht nur mit absoluten, sondern auch relativen Frequenzangaben generieren kann. Dies macht sie

besser vergleichbar mit anderen Listen, wodurch Schlüsselfunktionen oder -themen besser zu erkennen sind.

8.1.1 Wortlisten generieren

Um einfache Wortlisten zu erzeugen, müssen Texte gegebenenfalls zunächst bereinigt und dann in einzelne Tokens zerlegt werden, was man als **Segmentierung** oder **Tokenisierung** bezeichnet. Bei der ineffizienteren Methode über Listen würde man dabei zunächst jedes Token identifizieren und an die Liste anhängen, um somit eine unsortierte Liste mit Duplikaten zu erzeugen. Danach müsste diese Liste durch Vergleich der Elemente sortiert, dabei alle Duplikate entfernt und in eine neue Liste geschrieben werden.

Um auf etwas effizientere Weise schon existierende unsortierte Listen in sortierte Listen ohne Duplikate zu überführen, bietet Python mittels `sorted()`, `sort()` und `set()` spezielle Funktionen oder Methoden. Die `set()`-Funktion sorgt dafür, dass automatisch in einer Liste nur Unikate vorkommen, was für eine Wortliste optimal ist, da man sich dadurch das manuelle Aussortieren erspart. Die zwei Sortieroptionen `sorted()` und `sort()` werden wir im Folgenden etwas genauer besprechen.

8.1.2 Grundlagen der Sortierung

Python bietet zwei verschiedene Optionen zum Sortieren von listenartigen Objekten. Die `sort()`-Methode des Listenobjekts funktioniert nur mit Listen, da sie eine Methode dieses speziellen Objekttyps darstellt. Beim Aufruf dieser Methode wird die ursprüngliche Liste selbst verändert, so dass die Originalliste verloren geht. Die `sorted()`-Funktion hingegen funktioniert mit allen listenartigen Objekten und erstellt immer eine Kopie der ursprünglichen Liste.

Beide Sortieroptionen können über zwei optionale Schlüsselwortargumente (präziser) gesteuert werden, nämlich `reverse` und `key`. Ersteres erlaubt es, wie der Name schon sagt, die Liste rückwärts zu sortieren, wenn als Wert `True` angegeben ist. Die Voreinstellung ist jedoch `False`, so dass die Liste normalerweise aufsteigend sortiert wird. Die Verwendung von `key` bietet die Möglichkeit, einen zusätzlichen **Sortierschlüssel** anzugeben, also eine speziell definierte Art, in welcher Reihenfolge die Elemente sortiert bzw. bei der Sortierung miteinander verglichen werden sollen. So wird bei Angabe der Zeichenkettenmethode `str.lower` (allerdings ohne darauffolgende Klammern) als Schlüssel eine transformierte Kopie der Originalelemente in Kleinschreibung als Vergleichsbasis herangezogen, womit sich groß- und kleingeschriebene Elemente zusammen sortieren lassen, so wie in einem Wörterbuch. Neben Methoden wie `str.lower` und `str.upper`, können auch Funktionen wie `len`, aber auch komplexere Verfahren zur Generierung von Sortierschlüsseln verwendet werden. Mehr dazu besprechen wir in Abschnitt 8.3. In der folgenden Übung wollen wir jetzt erst eine einfache Wortliste erzeugen.



Übung 37 - Wortliste generieren

Schreiben Sie ein Programm `21_wortliste.py`, in dem Sie eine Wortliste aus Kafkas Erzählung *Die Verwandlung* mittels der Datei `kafka_verwandlung.txt` erstellen.

Darin importieren Sie zunächst das `re`-Modul und legen jeweils eine Variable für die Wortliste, die Eingabedatei und eine Ausgabedatei für die Wortliste an.

Die Variable für die Eingabedatei soll dabei schon den Dateinamen zugewiesen bekommen, mit dessen Hilfe Sie einen passenden Namen für die Ausgabedatei generieren. Um dies zu sicher zu erreichen, sollten Sie darauf achten, dass der Pfad zur Eingabedatei u.U. relativ oder auch länger sein könnte. Sie sollten daher die passenden Funktionen des `os.path`-Untermoduls verwenden, um den Pfad für die Ausgabedatei zu erzeugen.

Öffnen Sie dann, mit Fehlerbehandlung, die Eingabedatei und iterieren Sie über die Zeilen.

Verwenden Sie danach zwei reguläre Ausdrücke, um die Zeile zu bereinigen. Im Ersten löschen Sie alle Sonderzeichen, die Ihnen beim Sichten des Textes auffallen. Ersetzen Sie dann im zweiten mindestens zwei Vorkommen von ‚Leerzeichen‘ durch ein einzelnes. Wenden Sie dann die `strip()`-Methode auf die Zeile an, um etwaige Leerzeichen am Zeilenanfang oder -ende zu löschen.

Da durch die Bereinigung Leerzeilen entstanden sein können, testen Sie mithilfe des `not`-Operators, ob die Zeile leer ist. Tipp: eine leere Kette ist immer `False`. Falls die Bedingung erfüllt ist, verwenden Sie die `continue`-Anweisung, um die Zeile zu überspringen und die Schleife mit der nächsten Zeile weiter abzarbeiten.

Erweitern Sie dann die Wortliste, indem Sie mittels `re.split()` eine Liste generieren, die die Zeile an Leerzeichen aufspaltet.

Öffnen Sie danach, wieder mit geeigneter Fehlerbehandlung, die Ausgabedatei. Iterieren Sie über die Wortliste, wobei Sie innerhalb der Iteration diese zuerst durch die `set()`-Funktion von Duplikaten bereinigen und dann mit der `sorted()`-Funktion und `str.lower` als Schlüssel sortieren.

Schreiben Sie innerhalb der Schleife dann das jeweilige Wort und einen Zeilenumbruch in die Ausgabedatei.

Überprüfen Sie das Ergebnis in der Ausgabedatei.

Bei der Durchsicht der Wortliste ist Ihnen hoffentlich aufgefallen, dass hier manche Formen sowohl mit initialem Groß- als auch Kleinbuchstaben auftreten, was rein durch die Position im Satz bedingt ist. Auch finden wir oft mehrere flektierte Formen für ein Wort, was wir leider nicht so einfach vermeiden können, da eine Lemmatisierung sehr viel Aufwand erfordern würde. Zudem kann man leider auch nicht sehen, wie häufig die einzelnen Wortformen auftreten, weshalb wir auch nicht beurteilen können, wie

wichtig sie im Text sind. Letzteres ist jedoch bei den Frequenzlisten, die wir als nächstes besprechen werden, möglich.

8.2 Einfache Wortfrequenzlisten generieren

Das Vorgehen bei der Generierung von Wortfrequenzlisten ist dem bei der Generierung von Wortlisten sehr ähnlich, nur dass die gefundenen Types als Schlüssel mit dazugehörigem Zähler angelegt werden. Da Python es jedoch nicht erlaubt, den Wert zu einem Schlüssel zu inkrementieren, wenn dieser noch gar nicht existiert, muss zunächst (prophylaktisch) ein Standardwert festgelegt werden, um dann bei jedem weiteren Auftreten eines Wortes den Zähler erhöhen zu können. Dies kann über die vorher schon angesprochene Methode `setdefault()` des Dictionary-Objekts geschehen. Dieser muss als erstes Argument der Schlüsselnamen und als zweites ein Wert übergeben werden, der als Standard vorgegeben wird, falls noch kein Wert vorliegt.

Zur Ausgabe müssen die Dictionary-Schlüssel mittels der `keys()`-Methode abgerufen und dabei mit `sorted()` in eine sinnvolle Reihenfolge gebracht werden. Sonst werden sie nach der Reihenfolge des Eintrags ausgegeben, was aber meist keinen Sinn ergibt, da wir normalerweise Einträge erzeugen, wenn das erste Mal ein Token gefunden wird. Mit unserem bisherigen Wissen können wir allerdings nur alphabetisch auf- oder absteigend sortieren, was leider noch nicht optimal ist. In der folgenden Übung wollen wir zunächst eine alphabetisch sortierte Frequenzliste erzeugen.



Übung 38 - Wortfrequenzliste generieren

Speichern Sie das vorherige Programm als `22_frequenzliste.py` ab und passen Sie es so an, dass es eine Frequenzliste in einer entsprechend benannten Datei generiert.

Legen sie dabei statt der leeren Liste einen leeren Dictionary an und generieren Sie einen passenderen Namen für die Ausgabedatei.

Dort, wo Sie vorher die Ergebnisse der `re.split()`-Methode an die Liste angehängt hatten, verwenden Sie dieselbe Methode, iterieren aber diesmal in einer Schleife über die Ergebnisse und tragen das jeweilige Wort als Schlüssel ein. Dabei weisen Sie ihm mittels

```
Dictionaryname.setdefault(Schlüssel, 0) + 1
```

einen um 1 erhöhten Zähler zu, wobei `Dictionaryname` und `Schlüssel` natürlich durch Ihren eigenen Variablennamen und das jeweilige Wort ersetzt werden.

Bei der Ausgabe müssen Sie diesmal mit der passenden Methode über die sortierten Schlüssel des Dictionary iterieren.

Geben Sie dann innerhalb der Schleife mithilfe eines `f-strings` das jeweilige Wort, einen Tabulator (`\t`), den Wert des Wortes und einen Zeilenumbruch aus. Bei

Ausgabe von Frequenzen sollten Werte idealerweise untereinander stehen, auch wenn Wörter unterschiedlich lang sind. In unserem Fall soll deshalb die Breite der Wortausgabe immer so lang sein wie das längste Wort in der Liste. Sie müssen also bei jedem Wort, welches gefunden wird, auch immer prüfen, ob es das längste ist und gegebenenfalls diesen Wert speichern, um die Ausgabe richtig formatieren zu können.

8.3 Lambda-Funktionen

Bisher haben wir nur Funktionen kennengelernt, die vor ihrer Verwendung definiert werden müssen. In einigen Programmiersprachen gibt es aber die Möglichkeit, auch anonyme Funktionen zu definieren, also Funktionen, die ad hoc im Code definiert sind und direkt dort verwendet werden können, weshalb sie keinen Namen benötigen. In Python heißen diese Funktionen **Lambda-Funktionen** oder **Lambda-Ausdrücke**. Sie sind extrem nützlich für Filteroperationen oder die Erzeugung von Sortierschlüsseln, z.B. um Frequenzlisten nach Werten zu sortieren, so wie wir dies für unsere Wortfrequenzlisten benötigen. Die allgemeine Syntax für diese Ausdrücke ist

```
lambda Variable:Ausdruck
```

und wir können zum Beispiel mithilfe von

```
lambda wort:woerter[word]
```

auf den Wert von `wort` im Dictionary `woerter` zugreifen, um diesen als Sortierschlüssel für aufsteigende Frequenzen zu verwenden. Um eine rückläufige Sortierung, also nach absteigenden Frequenzen zu erreichen, können wir einfach den Wert negieren, also

```
-woerter[word]
```

schreiben, wobei das negative Vorzeichen hier ähnlich fungiert wie das `reverse`-Schlüsselwort für die allgemeine Sortierung. Um komplexere Sortierschlüssel zu generieren, kann man auch ein Tupel angeben, z.B.

```
lambda wort:(-woerter[word], wort.lower)
```

um neben der Sortierung nach der Frequenz auch eine alphabetische Sortierung zu erzwingen.

Im Folgenden wollen wir die Verwendung von Lambda-Funktionen üben. Dazu schreiben wir ein Objekt, welches eine Frequenzliste erzeugt und diese dann mit verschiedenen Sortieroptionen, sowohl alphabetisch als auch nach Frequenzen, ausgibt.

Dabei wollen wir es uns auch zunutze machen, dass Python eine stabile Sortierung generiert, was heißt, dass ein einmal sortierter Dictionary in derselben Reihenfolge von Schlüsseln erhalten bleibt.



Übung 39 - Klasse für Frequenzlisten

Schreiben Sie ein neues Modul `frequenzen.py`, in dem Sie eine Frequenzliste als Klasse implementieren.

Diese Klasse soll als Argumente im Konstruktor eine Eingabedatei, eine Ausgabedatei und eine Sortierreihenfolge übergeben bekommen, wobei als Standardwert für Letztere `n-1`, also rückwärtig nach Frequenz, angegeben sein soll.

Danach sollten Sie zwei Instanzvariablen, eine für die Frequenzliste und eine für eine sortierte Schlüsselliste, mit geeigneter Initialisierung, anlegen.

Im Konstruktor soll danach zunächst überprüft werden, ob eine Eingabedatei angegeben wurde und falls nicht, soll mithilfe der Anweisung `raise NameError(...)` und mit passender Meldung ein Eingabefehler an das Programm gemeldet werden, welches die Klasse verwendet.

Falls kein Fehler auftritt, soll hier auch wieder, falls nötig, der Ausgabedateiname erzeugt werden, wie wir dies vorher schon getan hatten. Liegt allerdings schon ein Ausgabedateiname, bzw. -Pfad, vor, dann soll dieser direkt verwendet werden.

Schreiben Sie danach eine Methode, um die Liste zunächst im Arbeitsspeicher zu generieren, wobei diese eine geeignete Fehlerbehandlung beinhalten sollte, und einen möglichen Betriebssystemfehler einfach mit `raise OSError`, wie oben besprochen, weiterleitet.

Innerhalb der Methode sollen Sie auch erfassen, wie lang das längste Wort und die längste Zahl sind und dies in Instanzvariablen der Klasse speichern, damit diese später für die Formatierung der Ausgabe verwendet werden können. Die Länge der längsten Zahl benötigen wir deshalb, weil wir die Frequenzen diesmal rechtsbündig ausgeben wollen.

Als nächstes sollten Sie, je nach angegebener Sortierreihenfolge, eine sortierte Liste der Schlüssel, jedoch ohne dazugehörige Werte, mittels einer weiteren Methode erzeugen. Diese soll später zur Steuerung der Ausgabe verwendet werden. Um deutsche ‚Sonderzeichen‘ besser handhaben zu können, verwenden Sie anstelle der `lower`-Methode die `casefold`-Methode. Bedenken Sie auch, dass Sie für die nach Frequenzen sortierte Ausgabe Lambda-Funktionen mit Tupeln im Sortierschlüssel verwenden müssen, wobei das erste Element des Tupels den Wert darstellt, aber danach noch einmal alphabetisch nach Schlüsseln sortiert werden muss. Für die Sortierung sollten Sie als Optionen zumindest alphabetisch auf- und absteigend und nach Frequenz auf- und absteigend implementieren. Außerdem können Sie selbst noch ein wenig überlegen, ob andere Sortierungen ebenfalls sinnvoll wären, und diese gegebenenfalls als zusätzliche Übung implementieren.

Danach fügen Sie eine Ausgabemethode hinzu, in der Sie, mit entsprechender Fehlerbehandlung, die Ausgabedatei öffnen und die sortierte Frequenzliste, wieder mit geeigneter f-String-Formatierung, ausgeben.

Zuletzt sollen Sie dann noch innerhalb der Datei einen Block zum Testen, wie unter 7.5.1 besprochen, schreiben, wobei Sie wieder Optionen zur Behandlung von Fehlern, die unter Umständen aus der Klasse gemeldet werden, mit implementieren.

8.4 Relative Frequenzen

Bisher haben wir nur rohe Frequenzen berechnet und uns anzeigen lassen. Durch sie bekommen wir zwar schon einen relativ guten Eindruck davon, welche Tokens am häufigsten auftreten und vielleicht für unsere Analysen interessanter sind, aber sie vermitteln uns noch keinen idealen Eindruck davon, welchen Anteil an unserem Text oder unseren Texten diese Tokens tatsächlich haben. Deshalb sind Rohfrequenzen auch schwerer interpretierbar und erlauben es uns auch nicht, sinnvolle Vergleiche zwischen Dateien oder Korpora anzustellen. Um einen besseren Eindruck von der Bedeutung der Types innerhalb von Dateien oder Korpora gewinnen zu können, müssen wir relative Frequenzen berechnen. Diese erhält man, indem man die Frequenzen der einzelnen Wort- oder Ngramm-Types durch die Anzahl aller gefundenen Tokens teilt. Durch Multiplikation mit 100 erhält man, falls erwünscht, dann auch Prozentangaben, die anfänglich vielleicht einfacher zu interpretieren sind, vor allem, da relative Frequenzen sehr niedrig sein können.

Zum direkten Vergleich zwischen verschiedenen Dateien oder Korpora können relative Frequenzen auch mit einem sinnvollen Faktor normiert werden, der am besten dem größten gemeinsamen Nenner der zu vergleichenden Dateien oder Korpora entspricht. Zwar ist es mathematisch korrekt, einen beliebigen (großen) Faktor, wie z.B. per Tausend oder Million Wörter/Tokens zu verwenden, was leider sehr häufig ‚blind‘ getan wird. Jedoch kann dies, falls der Faktor zu hoch angesetzt wird, zu einem verzerrenden Eindruck führen, so dass etwaige Differenzunterschiede größer erscheinen als sie tatsächlich sind.

Die Gesamtanzahl aller Tokens sollte idealerweise schon bei der Erstellung der Liste(n) durch eine global verfügbare Zählervariable erfasst werden, da sonst im Nachhinein alle Werte aufsummiert werden müssen. Bei Einsatz eines Objektes sollte diese Variable eine Instanzvariable sein, so dass sie von allen Programmteilen aus einfach erreichbar ist. Bei der Ausgabe relativer Frequenzen muss man auch darauf achten, dass diese vernünftig formatiert sind, so dass die normalerweise auftretenden Nachkommastellen deutlich zu sehen sind, insbesondere bei Wörtern mit sehr niedriger Frequenz.

Das Einbauen der nötigen Funktionalität, um relative Frequenzen durch das Objekt zu erzeugen, überlasse ich Ihnen als erweiterte Übung. Auch besprechen wir hier nicht

im Detail, wie man N-Gramme erzeugt. Im Prinzip sollte es schon klar sein, dass Sie dafür jeweils über die Inhalte von Eingabedateien iterieren müssen und mithilfe von Slices passender Länge die N-Gramme extrahieren. Da N-Gramme sich normalerweise nicht über Satzgrenzen hinweg erstrecken sollten, müssten Sie dabei auch die jeweiligen Texte in sinnvolle Einheiten zerlegen, aus denen dann die N-Gramme extrahiert werden.

Nachdem wir jetzt gesehen haben, wie wir quantitative Analysen für Wörter durchführen, beschäftigen wir uns im nächsten Kapitel damit, wie wir uns oder anderen Benutzern unserer Programme den Zugriff auf Daten oder Analysefunktionalität durch die Verwendung grafischer Benutzeroberflächen vereinfachen können.



8.5 Lösungen zu den Aufgaben

Lösung 37 - Wortliste generieren

Die ersten zwei Schritte bei der Erstellung dieses Programms dürften mittlerweile keinerlei Probleme mehr darstellen, so dass die ersten vier Zeilen wie folgt aussehen könnten.

```
import re
import os.path
woerter = []
eingabe_datei = './kafka_verwandlung.txt'
pfad, dateiname = os.path.split(eingabe_datei)
ausgabe_datei = os.path.join(pfad, 'wortliste_' + dateiname)
```

Die Zuweisung des Dateinamens an die Variable `eingabe_datei` macht das Programm zwar etwas unflexibel, da es immer nur aus der einen Datei eine Liste erzeugen kann, dient aber hier dazu, diesen Programmteil etwas zu vereinfachen. Deshalb müssen wir nicht auch das `sys`-Modul importieren und eine Fehlerbehandlung für das Einlesen des Dateinamens schreiben, sondern können uns stattdessen auf die wesentlichen Teile der Erzeugung einer Wortliste konzentrieren. In einem Programm, das Sie des Öfteren zur Generierung solcher Listen verwenden oder anderen zur Verfügung stellen wollen, sollten diese zusätzlichen Teile jedoch immer flexibel implementiert werden.

Für den Namen der Ausgabedatei bietet es sich an, wie im obigen Code, `wortliste_` vorne an den Eingabedateinamen anzuhängen. Dies zeigt zum einen direkt an, um was für eine Datei es sich handelt, behält jedoch gleichzeitig auch die Informationen zur Eingabequelle bei.

Zum Öffnen der Datei verwenden Sie idealerweise wieder den `with`-Operator und sichern dies mit einem `try`-Block ab. Im `except`-Block fangen Sie auch wieder, falls nötig, einen `OSError` ab und brechen das Programm mit Ausgabe des Fehlers ab. Das Iterieren über die Zeilen hatten wir schon in Übung 25 geübt, so dass das Einrichten

der nötigen `for`-Schleife kein Problem darstellen sollte. Innerhalb der Schleife sollten die Operationen zur Bereinigung in etwa so aussehen:

```
zeile = re.sub(r'>><<.,;!?:-]', '', zeile)
zeile = re.sub(r'\s{2,}', ' ', zeile)
zeile = zeile.strip()
```

Wie in der ersten Zeile oben zu sehen ist, sollten Sie auf jeden Fall die normalen Satzzeichen, aber auch die speziellen Formen der doppelten Anführungszeichen, » und «, die in diesem Text verwendet werden, löschen. Die weiteren zwei Operationen sollten keiner weiteren Erklärung bedürfen.

Der Test, ob noch Zeichen in der Zeile stehen, sollte am einfachsten mit `if not zeile:` durchgeführt werden. Nach dem Bedingungsblock, also falls tatsächlich die Zeile nicht leer ist, sollten Sie die Wortliste mittels der `extend()`-Methode des Listenobjekts um alle Wörter, die durch die `re.split()`-Operation zurückgeliefert werden, ergänzen. Ein typischer Fehler wäre es, stattdessen zu versuchen, die `append()`-Methode anzuwenden. Da jedoch durch die Aufspaltung der Zeile eine Liste erzeugt wird, würde in diesem Fall anstelle jedes einzelnen Wortes auf der Zeile diese Liste als ein einziges Element in der Wortliste angehängt, da `append()` immer ein Objekt anhängt, wohingegen `extend()` alle Elemente, die übergeben werden, einzeln anfügt. Mit anderen Worten arbeitet also `extend()` die Liste, die durch `re.split()` zurückgeliefert wird, einzeln ab, und fügt nach und nach alle einzelnen Wörter an die Wortliste an. Die richtige Anweisung sollte also

```
woerter.extend(re.split(r'\s', zeile))
```

sein.

Die `try`- und `except`-Blocks für die Ausgabe sind dieselben wie für die Eingabe, so dass Sie nur die Ausgabeschleife neu definieren müssen. Hier übergeben wir an die `sorted()`-Funktion die durch `set()` schon bereinigte Wortliste als erstes Argument und als zweites das Schlüsselwortargument `key` mit Wert `str.lower`, so dass wir innerhalb der Schleife nur noch das Wort, das durch den Iterator zurückgeliefert wird, und einen Zeilenumbruch in die Ausgabedatei schreiben müssen. Dadurch steht dann am Ende der Ausgabedatei noch ein Zeilenumbruch, aber das sollte bei der Verwendung der Wortliste oder eventuellen Weiterverarbeitung keine Probleme bereiten. Die vollständige Schleife sollte dann so aussehen:

```
for wort in sorted(set(woerter), key=str.lower):
    datei.write(wort + '\n')
```

Lösung 38 - Wortfrequenzliste generieren

Um einen leeren Dictionary anstelle der Liste anzulegen, müssen Sie nur die Zuweisung an die `woerter`-Variable ändern, entweder zu


```
woerter = {}
```

oder

```
woerter = dict()
```

Der Name der Ausgabedatei sollte diesmal statt mit `wortliste_` passender mit `frequenzliste_` beginnen.

Zur Verarbeitung der einzelnen Zeilen können wir natürlich diesmal nicht nur die jeweilige Zeile in einzelne Wörter aufspalten und an eine Liste anhängen, sondern müssen bei jedem einzelnen Wort im Dictionary ‚nachschiessen‘, ob es dort schon existiert, und darauf basierend unseren Zähler für dieses Wort hochsetzen. Dazu verwenden wir die `setdefault`-Methode, die prüft, ob dem Schlüssel schon ein Wert zugeordnet ist, und falls nicht, den von uns für den Schlüssel vorgegebenen Standardwert setzt. Existiert der Schlüssel bisher noch nicht, geben wir als Standardwert 0 an, um erst einmal einen passenden Eintrag zu erzeugen. Da wir jedoch im Rest der Anweisung immer 1 dazuzählen, wird effektiv beim Anlegen des Schlüssels, also beim ersten Auftreten des Wortes, auch dieser Wert zugeordnet. Existiert der Eintrag schon im Dictionary, dann liefert die `setdefault`-Methode jedoch den aktuellen Wert, den wir wiederum einfach um 1 erhöhen und dann dem Schlüssel zuweisen können. Die Schleife sollte also, zumindest anfänglich, wie folgt aussehen.

```
for wort in re.split(r'\s', zeile):
    woerter[word] = woerter.setdefault(word, 0) + 1
```

Allerdings wollen wir bei der Ausgabe die richtige Ausrichtung in Spalten erreichen, weshalb wir noch eine zusätzliche Variable benötigen, die die Länge des längsten Wortes beinhaltet. So können wir innerhalb des f-strings die Breite des Ausgabewortes (linksbündig) definieren. Diese Variable sollte zu Anfang des Programms, am besten gleich hinter den Variablen für die Ein- und Ausgabedatei definiert werden und anfänglich mit 0 initialisiert werden. Innerhalb der Zählschleife für die Frequenzermittlung sollte dann bei jedem Wort überprüft werden, ob es länger ist als das bisher längste Wort. Ist dies der Fall, dann muss dieser längere Wert der Variable zugewiesen werden. Wenn wir als Variable hier `laengstes_wort` verwenden, dann sollte die entsprechende Bedingungsanweisung folgendermaßen aussehen.

```
if len(wort) > laengstes_wort:
    laengstes_wort = len(wort)
```

Da die Schlüssel im Dictionary sowieso Unikate sind, müssen wir bei der Ausgabe kein Set mehr erzeugen, weshalb die Verwendung dieser Funktion entfällt. Allerdings müssen wir diesmal mithilfe von `keys()` über die Schlüssel iterieren. Dabei müssen wir diese noch sortieren und beim Vergleich wieder die Groß- und Kleinschreibung ignorieren, wozu wir wieder die `str.lower`-Methode als Sortierschlüssel verwenden. Allerdings können wir, wie oben angesprochen, bisher nur eine alphabetisch auf- oder

absteigende Sortierung erreichen. Die Liste wird also noch nicht optimal dargestellt, da wir ja eigentlich nach Frequenzen sortieren sollten. Wie wir dies erreichen können, werden wir im nächsten Abschnitt besprechen.

Lösung 39 - Klasse für Frequenzlisten

Um die Klasse effizient implementieren zu können, müssen wir wieder das `re`-Modul sowie wieder `os.path` importieren. Danach können wir mit der Implementierung der Klasse, zunächst dem Konstruktor, beginnen. Wichtig ist, dass als erstes Argument `self` übergeben wird, wobei für die Namen der Ein- und Ausgabedatei am besten als Standardwert `None` verwendet wird. Als Argumentvariable für die Eingabedatei verwenden wir sinnvollerweise `eingabe_datei`, für die Ausgabedatei `ausgabe_datei` und für die Sortierung einfach `sortierung`. Den Standardwert `n-1` für letztere setzen wir deshalb, weil er für Frequenzlisten im Allgemeinen der sinnvollste ist und zum anderen auch, damit wir ihn nicht jedes Mal beim Anlegen eines Objekts angeben müssen. Danach legen wir die zwei Variablen für die Wortliste (`woerter`) und eine sortierte Liste von Schlüsseln (`sortiert`) als Instanzvariablen an und initialisieren diese jeweils als leeren Dictionary und leere Liste.

Im nächsten Schritt testen wir idealerweise mittels einer Bedingung, ob kein Name für eine Eingabedatei übergeben wurde und leiten dann, falls nötig, den `NameError` an das Programm weiter, welches das Objekt verwendet. Falls ein Name vorliegt, sollte der Wert des Schlüsselwortarguments in einer Instanzvariable abgespeichert werden, nachdem wir zunächst vorsorglich den Pfad extrahiert haben. Da wir unseren Benutzern durch das Schlüsselwortargument für die Ausgabedatei auch erlauben, einen Dateinamen bzw. einen Pfad selbst anzugeben, überprüfen wir im nächsten Schritt wiederum, ob kein Name angegeben wurde und erzeugen gegebenenfalls wieder einen aus dem Namen der Eingabedatei. Dieser Name sollte ebenfalls wieder in einer Instanzvariable gespeichert werden. Zuletzt sollte im Konstruktor noch die Sortierreihenfolge als Instanzvariable abgelegt werden. Der komplette Konstruktor sollte dann so aussehen:

```
def __init__(self, eingabe_datei=None, ausgabe_datei=None,
             sortierung='n-1'):
    self.woerter = {}
    self.sortiert = []
    if not eingabe_datei:
        raise NameError('Keine Eingabedatei angegeben!'
                        'Unmöglich, Frequenzliste anzulegen...')
    else:
        pfad, dateiname = os.path.split(eingabe_datei)
        self.eingabe_datei = eingabe_datei
    if not ausgabe_datei:
        self.ausgabe_datei = os.path.join(pfad,
```

```

        'frequenzliste_' + eingabe_datei)
    else:
        self.ausgabe_datei = ausgabe_datei
        self.sortierung = sortierung

```

Die Methode, um die ‚rohe‘, unsortierte Frequenzliste zu generieren, sollte am besten so etwas wie `liste_erzeugen` heißen. Innerhalb dieser Methode muss zunächst der Dictionary, der die Liste enthält, über die `clear()`-Methode geleert werden, damit nicht bei einem versehentlichen weiteren Aufruf der Methode die Frequenzen fälschlicherweise vervielfacht werden. Danach sollten erst einmal die zwei Variablen, die die Länge des längsten Wortes und der längsten Zahl beinhalten, mit 0 initialisiert werden. Bitte beachten Sie dabei, dass diese Variablen nicht im Konstruktor angelegt werden müssen, sondern, solange Sie als Instanzvariablen definiert sind, so wie hier irgendwo im Programm definiert und initialisiert werden können.

Das Erzeugen der Liste selbst geschieht genauso wie in der letzten Übung, nur dass diesmal ein möglicher Fehler beim Öffnen der Datei nicht zum Programmabbruch führen, sondern der Fehler wieder mittels `raise` weitergeleitet werden sollte. Außerdem müssen die verschiedenen Variablen, die im gesamten Modul zugänglich sein sollen, auch als Instanzvariablen verwendet werden, so dass sie jedes Mal mit `self.` präfigiert sein sollten. Ein weiterer Unterschied zum vorherigen Programm ist auch, dass jedes Mal, nachdem ein Wert einem Wort zugeordnet wurde, überprüft werden muss, ob die Länge der jeweiligen Zahl größer ist als die der bisher erfassten. Auch wenn das Verfahren hier ähnlich der Ermittlung der Länge des längsten Wortes ist, gibt es dabei das Problem, dass die Zahl dafür immer erst in eine Zeichenkette gewandelt werden muss, um die Länge zu bestimmen. Sonst tritt ein `TypeError` auf, weil `len` nicht für Zahlen definiert ist. Die vollständige Methode sollte dann in etwa so aussehen.

```

def liste_erzeugen(self):
    self.woerter.clear()
    self.laengstes_wort = 0
    self.max_laenge_zahl = 0
    try:
        with open(self.eingabe_datei, 'r', encoding='utf-8')
        as datei:
            for zeile in datei:
                zeile = re.sub(r'[\«.,;!?:-]', '', zeile)
                zeile = re.sub(r'\s{2,}', ' ', zeile)
                zeile = zeile.strip()
                if not zeile:
                    continue
                for wort in re.split(r'\s', zeile):
                    if len(wort) > self.laengstes_wort:

```

```

        self.laengstes_wort = len(wort)
    self.woerter[word] =
    self.woerter.setdefault(word, 0) + 1
    if len(str(self.woerter[word])) >
    self.max_laenge_zahl:
        self.max_laenge_zahl =
        len(str(self.woerter[word]))
except OSError as fehler:
    raise OSError(fehler)

```

Die Sortiermethode, welche wir hier `liste_sortieren` nennen, ist relativ einfach zu implementieren. Es muss darin eigentlich nur abgefragt werden, welche Sortierreihenfolge vorgegeben wurde. Darauf basierend wird die `sorted`-Funktion auf die Schlüssel der ursprünglichen Wortliste angewandt, wobei das Ergebnis der sortierten Liste zugewiesen wird. Für jeden der vier Fälle, die wir minimal abdecken wollen, muss dafür ein entsprechender, mehr oder weniger komplexer, Sortierschlüssel angegeben werden. Die Methode sollte dann in etwa so aussehen:

```

def liste_sortieren(self):
    if self.sortierung=='a-z':
        self.sortiert = sorted(self.woerter.keys(),
                                key=str.casefold)
    elif self.sortierung=='z-a':
        self.sortiert = sorted(self.woerter.keys(),
                                key=str.casefold, reverse=True)
    elif self.sortierung=='n-1':
        self.sortiert = sorted(self.woerter.keys(),
                                key=lambda wort:(-self.woerter[word], wort.casefold()))
    else:
        self.sortiert = sorted(self.woerter.keys(),
                                key=lambda wort:(self.woerter[word], wort.casefold()))

```

Die alphabetischen Sortierungen sind hierbei am einfachsten, da wir nur bei beiden als Sortierschlüssel die `casefold`-Methode der Zeichenkette angeben, und bei der umgekehrten Sortierreihenfolge das Schlüsselwortargument `reverse` auf `True` setzen müssen. Bei der Sortierung nach Frequenzen müssen wir mit einer Lambda-Funktion und einem Tupel für den Sortierschlüssel arbeiten, in dem mit dem ersten Element auf die Frequenzinformation zugegriffen und mit dem zweiten die `casefold`-Methode aktiviert wird, damit auch die Schlüssel noch in geeigneter Weise sortiert werden. Bei der nach Frequenz absteigenden Sortierung muss natürlich ein `-` vor dem Wertezugriff stehen, damit diese Sortierung erreicht werden kann.

Die Sortieroptionen könnten hier je nach Bedarf noch erweitert werden. Zum Beispiel könnte man auch nach der Länge der Wörter (auf- oder absteigend) sortieren,

um zu sehen, wie häufig Wörter mit bestimmten Längen sind. Oder man könnte eine rückläufige Sortierung implementieren, um nach Endungen zu sortieren, was für morphologische Analysen sinnvoll sein kann. Letzteres lässt sich damit erreichen, dass man innerhalb eines Lambda-Ausdrucks im Sortierschlüssel mithilfe von `word[::-1]` auf das umgedrehte Wort zugreift.

Die Ausgabemethode für die Liste, die wir hier einfach `liste_ausgeben` nennen, sollte wieder dieselbe Fehlerbehandlungsroutine verwenden, wie beim Einlesen der Eingabedatei. Zur Ausgabe selbst müssen Sie nur über die sortierte Liste iterieren und jeweils das Wort von dort ausgeben, dann aber die dazugehörige Frequenz aus dem unsortierten Dictionary holen. Für die Ausgabe in die Datei verwenden Sie am einfachsten wieder einen `f-string`, da sich dort sowohl das Wort als auch die Frequenz mit passender Formatierung interpolieren lassen. Die Ausgabe innerhalb der Schleife sollte also folgendermaßen aussehen.

```
datei.write(
    f'{word: {self.laengstes_wort}}\t'
    f'{self.woerter[word]:>{self.max_laenge_zahl}d}\n')
```

Wichtig ist hierbei, dass zur rechtsbündigen Formatierung der ‚Pfeil‘ nach rechts (>) angegeben und die Frequenz durch die Angabe von `d` als formatierte Ganzzahl markiert ist.

Innerhalb des Test-Blocks müssen Sie zunächst ein Frequenzlistenobjekt anlegen, was Sie nach Belieben initialisieren können, um auf verschiedene Dateien zur Ein- oder Ausgabe zuzugreifen, oder die Sortieroptionen zu überprüfen. Danach müssen Sie zum Testen die verschiedenen Methoden nacheinander aufrufen, um Listen zu generieren und auszugeben. Die Initialisierung und der Aufruf der Methoden sollten zur Fehlerbehandlung in einen `try`-Block eingebettet sein. Da in unserer Klasse zwei verschiedene Arten von Fehlern auftreten können, nämlich ein `NameError` bei einem fehlenden Argument für die Eingabedatei, und ein `OSError` bei Fehlern, die die Ein- oder Ausgabedatei betreffen, sollten auch zwei `except`-Blocks dafür vorhanden sein, die Letztere behandeln können. Der komplette Test-Block könnte dann z.B. folgendermaßen aussehen.

```
if __name__ == '__main__':
    try:
        f_liste = Frequenzliste(eingabe_datei=
            'kafka_verwandlung.txt', ausgabe_datei=
            'test_frequenzliste.txt', sortierung='n-1')
        f_liste.liste_erzeugen()
        f_liste.liste_sortieren()
        f_liste.liste_ausgeben()
    except OSError as fl:
        print('Falsche Ein- oder Ausgabedatei angegeben!')
```

```
    str(f1).split(' ')[-1])  
except NameError as f2:  
    print(str(f2))
```

Der obige Code sollte, bis auf eine Ausnahme, einfach zu verstehen sein. Die Ausgabe für den `OSError` enthält allerdings das zweite Argument für die `print`-Anweisung

```
str(f1).split(' ')[-1],
```

welches aus dem Fehler das letzte Feld extrahiert, nämlich den Dateinamen oder -pfad, der den Fehler ausgelöst hat. Dabei wird der Fehler selbst zunächst mittels `str()` in eine Zeichenkette gewandelt, dann die einzelnen Felder mit `split(' ')` an Leerzeichen aufgespalten und in eine Liste gewandelt, von der wir nur auf das letzte Element zugreifen. Dieses liefert die Informationen, die wir an den Benutzer weitergeben wollen, nämlich welche Eingabe den Fehler ausgelöst hat.

9 Einfache grafische Benutzeroberflächen



Grafische Benutzeroberflächen (GUIs) erleichtern die Arbeit mit Daten erheblich, insbesondere da wir mit ihrer Hilfe während der Laufzeit eines Programms diese Daten bequem auswählen oder verschiedene Optionen zur Darstellung verändern können. Deshalb vermittelt Ihnen dieses Kapitel den Einstieg in die Erstellung solcher Fensterprogramme mit der plattformübergreifend einsetzbaren Bibliothek PyQt. Dabei besprechen wir erst, wie PyQt aufgebaut ist, erörtern dann den allgemeinen Ansatz zur Entwicklung von GUI-Programmen, welche PyQt-Programmteile (Widgets) am nützlichsten für unsere Arbeit sind, und wie diese erzeugt werden. Danach sehen wir uns an, wie man mit Layouts arbeitet, um Programme übersichtlich und benutzerfreundlich zu gestalten, welche Eigenschaften die verschiedenen Widgets aufweisen, und wie durch deren Manipulation interaktive Funktionalität erreicht werden kann. Zu guter Letzt werden wir noch lernen, wie durch PyQt auch die Arbeit mit Dateien und Verzeichnissen erleichtert wird.

9.1 Grafische Benutzeroberflächen

Grafische Benutzeroberflächen, meist abgekürzt zu GUIs (engl. *graphical user interfaces*), ermöglichen vereinfachte Interaktion mit Daten und/oder Benutzern, z.B. zur Auswahl von Programmoptionen, die sonst als beim Start als Argumente übergeben werden müssten, auch nachdem das Programm schon gestartet wurde. Diese Daten können etwa zu bearbeitende Dateien, verschiedene Darstellungsoptionen für angezeigte Daten (Sortieroptionen, Formatierungen etc.) sein, oder Möglichkeiten für die interaktive Bearbeitung von Daten. Dabei warten GUI-Programme innerhalb einer Ereignisschleife (engl. *event loop*) ständig auf Benutzeraktionen oder andere Ereignisse (engl. *events*), wie etwa Mausklicks zum Starten von Aktionen oder der Auswahl von Optionen, die Auswahl oder das Öffnen von Dateien oder Verzeichnissen, das Eintippen von Text in Editor- oder Eingabefenster, oder die Ausgabe von Ergebnissen.

Der als Zusatzmodul installierbare QtDesigner erlaubt es zwar, reine GUIs (ohne benutzerdefinierte Funktionen) interaktiv ‚zusammenzubasteln‘, aber die resultierende `.ui`-Datei muss dann erst noch konvertiert, in das Programm eingebunden und auch jedes Mal neu kompiliert werden, wenn Änderungen am Programm vorgenommen wurden. Wir erstellen aber hier unsere GUIs nur rein manuell, um dadurch ein besseres Verständnis für die einzelnen Komponenten und Interaktion mit der grafischen Schnittstelle zu erreichen. Um zusätzliche Optionen, wie z.B. den QtDesigner,

zu installieren, können Sie später `pip(3) install pyqt5-tools` verwenden und selbst damit experimentieren.

9.1.1 PyQt-Grundlagen

Wie alle GUI-Pakete (engl. *toolkits*) bietet auch PyQt Steuerelemente (engl. *widgets*), die, Lego-artig, für die Interaktion mit Benutzern zusammengebaut werden. Diese Elemente repräsentieren meist Fenster oder grafische Objekte, die diverse Arten von Funktionalität bieten, wobei Qt allerdings auch abstrakte Dokumentenmodelle, wie z.B. für Text- oder Multimediadokumente, Listen, Tabellen etc., bietet.

Bei Programmfenstern können wir zwischen zwei Haupttypen unterscheiden, zum einen Dialoge, die für einfache Programme mit beschränktem Funktionsumfang und simplem Layout verwendet werden, aber dabei normalerweise keine Menüs oder Statuszeilen bieten, und zum anderen Hauptfenster, die für komplexere Programme benötigt werden. Letztere haben gewöhnlich Menüs, optional auch Knopfleisten, Statuszeilen, oder andockbare Leisten, und müssen in PyQt immer ein **zentrales Steuerelement** (engl. *central widget*) aufweisen. Die Kommunikation zwischen den Steuerelementen und dem Programm erfolgt in PyQt über Signale (engl. *signals*), wie z.B. Mausklicken oder -ziehen. Diese Signale werden mit Methoden (engl. *slots*) verknüpft, um die Interaktion zu erreichen.

Um PyQt-Programme zu erstellen, benötigt man generell Importe aus drei verschiedenen Modulen, die wir im Folgenden kurz besprechen werden.

Modul	Funktionalität
PyQt5.QtCore	bietet die Kernfunktionalität und Elemente für Interaktion wie Signale und slots, Koordinaten über <code>QPoint</code> , Größen mittels <code>QSize</code> sowie Ein- und Ausgabemethoden und -operationen
PyQt5.QtGui	liefert die Basisfunktionalität für Widgets, Farben (<code>QColor</code>) und Schriftarten (<code>QFont</code>), Tastenkombinationen (<code>QKeySequence</code>), Mauspositionen (<code>QCursor</code>), Texteingenschaften und -operationen
PyQt5.QtWidgets	stellt die Funktionalität zum Erstellen von Programmen über das Programm selbst als <code>QApplication</code> , für Hauptfenster und Dialoge über <code>QMainWindow</code> oder <code>QDialog</code> , verschiedene Layoutoptionen, Steuerelemente und Menüs sowie Aktionen mittels <code>QAction</code> , zur Verfügung

Tabelle 16: PyQt5-Submodule

Für Importe aus 1) und 2) bietet sich `from Modul import *` an, bei 3) eher selektivere Optionen, nur für die benötigten Steuerelemente, um Speicherplatz zu sparen.

9.2 Allgemeiner Ansatz zur Entwicklung von GUI-Programmen

Der allgemeine Ansatz zur Entwicklung von GUI-Programmen besteht aus einer Reihe von einfachen Schritten. Zunächst muss man sich für eine Anfangsoption entscheiden, und zwar entweder die direkte Verwendung einer Dialogklasse, was sich eher für kleine oder einfache Programme anbietet, oder die Ableitung und Anpassung einer Hauptfensterklasse über die `__init__`-Methode, die diese Klasse initialisiert. Wird eine Klasse abgeleitet, dann muss, wie auch bei der Ableitung anderer Objekte, der Name der Elternklasse als Argument in der Klassendefinition angegeben werden.

Nach dieser Entscheidung erfolgt die Auswahl von Layoutoptionen sowie das Anlegen und die Konfiguration von GUI-Elementen. Oft geschieht dies bei Ableitung über eine selbstdefinierte `initUI`-Methode, wobei das separate Anlegen der Elemente die grafische Schnittstelle besser von den eigentlichen Programmroutinen, bzw. der Programmlogik, trennt. Zwar muss man sich dabei schon im Voraus über die Funktionalität des Programms bewusst sein, aber noch nicht alle Funktionen oder Methoden vollständig implementiert haben. Oder man kann, bei schon bestehenden externen Modulen, deren Methoden, die nur vom Programm aufgerufen werden und ihre Ergebnisse an diese zurückliefern, einfach später einbinden. Um eine bessere Vorstellung davon zu erhalten, wie das Programm aussehen sollte, bietet es sich an, eine grobe Skizze des gesamten Layouts anzulegen. Hierbei muss man gut überlegen, welche Programmoptionen unter Umständen während der Laufzeit des Programms verändert werden müssen und über welche Arten von Steuerelementen dies am besten erreicht werden kann.

Zuletzt muss man dann noch Aktionen und Methoden mit Steuerelementen verknüpfen, was entweder mithilfe von `QActions` geschieht, die dann sowohl in Menüs als auch Knopfleisten verwendbar sind, oder über benutzerdefinierte Methoden, die mit Signals verbunden werden.

9.2.1 Nützliche PyQt-Steuerelemente

Wie schon oben angesprochen, muss man sich bei der Erstellung von GUIs auch genau Gedanken darüber machen, welche Steuerelemente die geplanten Aufgaben am besten erfüllen, damit das Programm später so übersichtlich wie nur möglich wird. Deshalb besprechen wir im Folgenden erst einmal einige der wichtigsten Steuerelemente kurz, explorieren diese danach in einer Übung weiter und machen uns erste Gedanken über ein einfaches Programm.

Steuerelement	Beschreibung
QWidget	Basisklasse für alle Steuerelemente, von der alle anderen Steuerelemente abgeleitet sind, weshalb sie eher seltener direkt verwendet wird
QFrame (engl. <i>frame</i> = Rahmen)	oft nicht sichtbarer Behälter für Layouts, über die wir in Abschnitt 9.3 mehr erfahren werden
QLabel	nicht-editierbares Textfeld zum Anzeigen von Informationen oder Nachrichten, dessen Text jedoch, je nach Bedarf, über das Programm verändert werden kann. Für Informationen, die für Benutzer kopierbar sein sollten, eignen sie sich allerdings nicht
QLineEdit	einzeiliges editierbares Textfeld zur Eingabe von Argumenten oder Ausgabe von kopierbaren Ergebnissen
QTextEdit	mehrzeiliges Textfeld zum Arbeiten mit kompletten Texten, auch HTML
QPushButton	Knopf zum Ausführen von Aktionen
QListWidget	Listenfeld, das untereinander Einträge platzsparend auflistet, was sich vor allem zur Auswahl von mehreren einzelnen Optionen anbietet, die von Benutzern nicht veränderbar sein sollen
QComboBox	editierbare, aufklappbare Auswahlliste; Vorteil gegenüber <code>QListWidget</code> : Benutzer können auch eigene Werte für Optionen eingeben
QSpinBox	Feld zur Zahleneinstellung über Knöpfe, die die Werte darin herauf- oder herabsetzen können
QCheckBox	Feld zum An- oder Abwählen von Optionen, in dem ein Häkchen zu Anwahl gesetzt werden kann; kann nützlich sein, um mehrere Optionen, auch parallel, anzuschalten
QRadioButton	runder Auswahlknopf (normalerweise in einer <code>QButtonGroup</code>), der normalerweise eine von einer relativ geringen Anzahl an Optionen auswählt
QMenuBar	Menüleiste, die meist mehrere Menüs beinhaltet, die wiederum auch Untermenüs haben können, wobei alle dieser Menüs als <code>QMenu</code> angelegt werden
QToolBar	Behälter für <code>QPushButtons</code> oder <code>QActions</code>
QStatusBar	Statuszeile zur Ausgabe von Einstellungen, Optionen etc.
QFileDialog	vorgefertigter Dialog zur Auswahl von Dateien oder Verzeichnissen, der normalerweise zum Öffnen/Laden oder Speichern verwendet wird

Tabelle 17: Auswahl nützlicher Steuerelemente

Wie Sie hier sehen, ist selbst diese noch relativ beschränkte Liste einigermaßen lang. Es empfiehlt sich also durchaus, nicht immer alle Elemente, die durch `PyQt5.QtWidgets` zur Verfügung gestellt werden, automatisch mit `*` zu importieren.

Übung 40 - Steuerelemente verstehen lernen

Laden Sie zunächst das Archiv unter http://martinweisser.org/tools/zips/Widget_demo.zip herunter und entpacken Sie es. Starten Sie das Programm `widget_demo.py` und machen Sie sich mit den verschiedenen Typen von Widgets etwas vertraut, ohne jedoch genauer auf die zugeordneten Beschreibungen einzugehen.

Überlegen Sie, teilweise basierend auf den Programmen, die wir bisher schon entwickelt haben, für welche Zwecke die verschiedenen Typen bei unserer Arbeit nützlich sein könnten.

Welche Steuerelemente würden Sie verwenden, um ein Programm zu erstellen, mit dem wir unsere syntaktische Inversion einfacher Deklarativsätze darstellen könnten?

Zeichnen Sie sich dafür ein Layout und überlegen Sie grob, wie die einzelnen Elemente mit verschiedenen Aktionen verknüpft werden sollten und welche potenziellen Fehler man behandeln müsste.

9.2.2 Ein minimales PyQt-Programm

Ein minimales PyQt-Programm kann einfach in einem `__main__`-Block implementiert werden, insofern zuerst die benötigten Module importiert wurden. Wir wollen dies hier an einem kurzen Beispiel nachvollziehen, welches Sie Schritt für Schritt in den Editor eintippen sollten, um es später austesten zu können.

```
import sys
from PyQt5.QtWidgets import QApplication, QDialog, QLabel
```

Das `sys`-Modul importieren wir, um ein Kommandozeilenargument einzuholen und um später das Programm ordnungsgemäß zu beenden und etwaige Fehler dabei auszugeben. Als Steuerelemente importieren wir `QApplication`, was immer benötigt wird, `QDialog`, da wir nur ein Dialog- und kein Hauptfenster erzeugen wollen und `QLabel`, damit wir in dem Dialogfenster auch eine Nachricht ausgeben können. Elemente, bzw. Funktionalität, aus `QtCore` oder `QtGui` benötigen wir nicht.

```
if __name__ == '__main__':
```

Nach dem Anlegen des `__main__`-Blocks, den wir benötigen, weil wir ja das Programm eigenständig starten wollen, legen wir zunächst eine Objektvariable für das GUI-Programm selbst an, wofür wir `QApplication` verwenden müssen. Dieses Programmobjekt ist allerdings nicht für Benutzer sichtbar. Als Argumente an das Objekt übergeben wir



einfach alles, was als Kommandozeilenargumente existiert. Hierüber könnten wir z.B. bestimmte Optionen für den Programmstart übergeben.

```
app = QApplication(sys.argv)
```

Im nächsten Schritt wird das (später) sichtbare Fenster des Programms als Dialogobjekt angelegt, aber noch nicht angezeigt.

```
fenster = QDialog()
```

Danach erzeugen wir ein Label, dessen **Elternteil** (engl. *parent*), also dasjenige Steuerelement, dem es zugeordnet ist, unser Dialogfenster ist, was als Argument explizit angegeben ist. Allerdings könnten wir hier auch schon andere Eigenschaften des Labels, wie z.B. seinen Text, festlegen.

```
label = QLabel(fenster)
```

Der Fenstertext, den wir vorher nicht festlegen wollten, um ihn variabel halten zu können, wird jetzt mit der Methode `setText` gesetzt, wobei wir als Text das erste Argument auf der Kommandozeile verwenden.

```
label.setText(sys.argv[1])
```

Da wir (noch) kein spezielles Layout verwenden, müssen wir jetzt das Label explizit durch `move` innerhalb des Fensters positionieren, und zwar in unserem Fall 100 Pixel nach rechts, also auf der x-Achse eines imaginären Koordinatensystems, und 20 Pixel nach unten (auf der y-Achse).

```
label.move(100,20)
```

Danach positionieren wir das Fenster selbst 100 Pixel nach rechts und 100 Pixel nach unten, gemessen vom linken oberen Bildrand, und machen es 250 Pixel breit und 50 Pixel hoch.

```
fenster.setGeometry(100,100,250,50)
```

Über die folgende Anweisung setzen wir den Text in der Titelzeile des Programms, der normalerweise als Programmname fungiert.

```
fenster.setWindowTitle('Einfacher Dialog')
```

Nachdem wir das Dialogfenster komplett konstruiert haben, können wir es mit `show()` zum Anzeigen vorbereiten. Bei einem schon laufenden Programm würde es dann auch gleich angezeigt.

```
fenster.show()
```

Allerdings erfolgt die eigentliche Anzeige erst, nachdem wir das Programm tatsächlich gestartet haben, was mit `app.exec_()` ausgeführt wird. Hierbei dient die Einbindung

in `sys.exit()` nur zur besseren Kontrolle des Betriebssystems über das Programm und würde sonst nicht benötigt.

```
sys.exit(app.exec_())
```

Wenn Sie das Programm soweit eingetippt haben, können Sie es unter einem selbstgewählten Namen abspeichern und ausführen. Beachten Sie aber bitte, dass Sie dabei eine Zeichenkette als Argument übergeben müssen, die dann auf dem Label dargestellt wird, da ansonsten das Programm nie startet und ein `IndexError` aufgrund des fehlenden Arguments auftritt. Wenn Sie wollen, können Sie das Programm selbstständig so anpassen, dass der Fehler abgefangen wird.

Wie Sie gerade gesehen haben, ist es durchaus möglich, ein kleines, einfaches Programm in dieser Form zu gestalten. Allerdings würde dies, sobald die Komplexität auch nur ein bisschen steigt, schnell recht unübersichtlich werden. Im Normalfall ist es sinnvoller, den größten Teil der Schnittstelle und die Programmfunktionalität in einem abgeleiteten Hauptprogramm mit eigenen Methoden zu entwickeln und dann nur noch den Programmaufruf im `__main__`-Block erfolgen zu lassen. Außerdem ermöglicht uns die Entwicklung im Hauptprogramm auch, unser Programm unter Umständen später als Komponente in ein anderes Programm einzubetten.

9.2.3 Ableitung eines Hauptfensters

Neue Steuerelementklassen für größere Fensterprogramme werden (nach den `import`-Anweisungen) durch die Definition einer Klasse mit Argument `QMainWindow` als Elternklasse kreiert. Natürlich ist es auch möglich, neue Elemente basierend auf `QDialog` zu entwerfen, oder spezielle Formen anderer Steuerelemente mit veränderter Funktionalität, worauf wir aber hier nicht gesondert eingehen, da es im Prinzip genauso funktioniert wie die Ableitung der Hauptklasse. Durch Ableitung kann übrigens nicht nur neue Funktionalität durch zusätzliche Methoden oder Eigenschaften entwickelt werden, sondern es können auch schon bestehende Methoden umgeschrieben werden, indem sie mit demselben Namen innerhalb der abgeleiteten Klasse neu implementiert werden. Letzteres ist ein allgemeines Prinzip bei der Ableitung von Objekten in Programmiersprachen, welches uns erlaubt, spezielle Unterkategorien von Objekten anzulegen.

Nach diesem kurzen Exkurs wollen wir jetzt jedoch wieder zurück zur Ableitung unserer neuen Hauptfensterklasse kommen. Nach erfolgter Klassendefinition muss zunächst die Initialisierungsfunktion des Objekts `__init__(self)` angelegt werden, worin wiederum als erstes die Initialisierungsfunktion der Oberklasse mittels

```
super().__init__()
```

aufgerufen werden muss. Durch diesen Aufruf stehen alle Eigenschaften und Methoden der Oberklasse zur Verfügung und können nach Belieben an die eigenen Bedürfnisse angepasst werden. Daraufhin können potenzielle Argumente abgearbeitet und das

Anwendungsfenster mittels einer Funktion, die oft `initUI` genannt ist, konfiguriert werden, was wir Schritt für Schritt in den folgenden Übungen lernen werden.



Übung 41 - Inversion mit GUI, Teil 1

Legen Sie ein neues Programm namens `GUI_syn_inversion.py` an, in dem wir jetzt tatsächlich unsere syntaktische Inversion einfacher Deklarativsätze simulieren wollen.

Importieren Sie darin zunächst die Module `sys` und `re` komplett.

Importieren Sie außerdem von `PyQt5.QtWidgets` `QApplication`, `QMainWindow`, `QHBoxLayout`, `QVBoxLayout`, `QLabel`, `QLineEdit`, `QPushButton`, `QFrame`, und `QMessageBox`, und von `PyQt5.QtGui` `QFont`.

Legen Sie jetzt eine neue Klasse `Inverter`, abgeleitet von `QMainWindow`, an.

Definieren Sie danach die `__init__()`-Methode, wie gerade beschrieben und fügen Sie noch die folgenden zwei Zeilen hinzu:

```
self.setFont(QFont("Courier", 12))
self.initUI()
```

Erstere sorgt dafür, dass dem Fenster eine geeignete Fonteinstellung mit Fontnamen und Punktgröße hinzugefügt wird, und Letztere, dass die `initUI`-Methode, die wir als nächstes schreiben werden, aufgerufen wird, um das Layout und die Steuerelemente des Programms anzulegen. Bevor wir dies jedoch tun können, müssen wir noch etwas über die Verwendung von Layouts erfahren.

Im Prinzip haben wir mit dem Code aus Übung 41, bis auf die fehlende Implementierung der `initUI`-Methode, schon eine lauffähige Klasse für unser Programm entwickelt. Diese könnten wir, wenn wir temporär den Aufruf der Methode auskommentieren, innerhalb eines `__main__`-Blocks instanziiieren und aufrufen, so wie wir das oben für den einfachen Dialog gesehen haben. Da wir jedoch einige der oben besprochenen Schritte entweder nicht benötigen bzw. die Layoutgestaltung innerhalb unserer Klasse implementiert werden soll, könnte jetzt unser verkürzter `__main__`-Block wie folgt aussehen.

```
if __name__ == '__main__':
# Instanziierung des Programms
    app = QApplication(sys.argv)
# Anlegen des Fensterobjekts
    inverter = Inverter()
    inverter.setWindowTitle('Einfache syntaktische Inversion')
    inverter.setGeometry(15, 35, 600, 100)
# Anzeigen des Fensters
    inverter.show()
```

```
# Programmaufruf
sys.exit(app.exec_())
```

Einen ähnlichen `__main__`-Block werden Sie wahrscheinlich für die meisten Ihrer GUI-Programme anlegen, so dass Sie sich diesen schon als Vorlage abspeichern können, die Sie dann nur noch in andere Programme hineinkopieren und anpassen müssen.

9.3 Mit Layouts arbeiten

Layouts erlauben es, Steuerelemente so zu gruppieren und anzuordnen, dass man zum einen nicht alle Elemente fest positionieren muss, aber auch Fenster flexibel vergrößern oder verkleinern kann, ohne die Elemente explizit selbst neu positionieren zu müssen. Wir wollen hier nur drei Haupttypen besprechen:

1. das `QVBoxLayout`, welches eine vertikale Anordnung in einer Spalte erlaubt;
2. das `QHBoxLayout`, was die horizontale Anordnung in einer Reihe ermöglicht;
3. das `QGridLayout` (engl. *grid* = Gitter), was die Eigenschaften der beiden Vorhergehenden kombiniert und beide Anordnung mit festen Positionen oder Bereichen innerhalb eines Gitters gestattet.

Da Gitterlayouts komplizierter werden können, besprechen wir hier nur die allgemeine Methode, um Steuerelemente im Gitter einzufügen, was über

```
addWidget(Steuerelement, Zeilenindex, Reihenindex, Zeilenspanne,
Reihenspanne)
```

erfolgt. Zeilen- und Reihenspanne definieren, über wie viele Zellen im Gitter sich das Element erstrecken soll, sind aber optional. Außerdem kann zusätzlich noch eine Ausrichtung als letztes Argument angegeben werden. Um komplexere Layouts zu erreichen, können anstelle von einzelnen Steuerelementen auch ganze Layouts mit `addLayout()` und denselben Argumenttypen eingefügt werden.

Wenn nicht ein Gitterlayout verwendet wird, sind die anderen Typen meist ineinander verschachtelt, um etwas komplexere Layouts erzeugen zu können und auch Elemente mit zusammengehörigen Funktionen zusammenzufassen. Bei Hauptfenstern eignet sich oft am besten ein `QFrame` als zentrales Steuerelement und Behälter für Layouts, so dass dieses zuerst angelegt werden muss. Allerdings wäre es auch möglich, ein `QWidget` zu verwenden. In unserem Fall aber wollen wir tatsächlich einen Rahmen verwenden, weshalb wir

```
behaelter = QFrame()
```

schreiben können. Danach müssen wir unseren Behälter als zentrales Steuerelement unserer Hauptfensterklasse festlegen, was mittels

```
self.setCentralWidget(behaelter)
```

geschehen kann. Wie wir schon vorher gesehen haben, taucht hier für das Setzen einer Eigenschaft wieder *set* als Bestandteil der Methode auf, was bei vielen Methoden der Fall ist, die solche Eigenschaften setzen oder verändern.

Im nächsten Schritt definieren wir jetzt unsere Haupt- und Unterlayouts. Bei relativ kleinen Programmen, in denen normalerweise nicht viele Elemente nebeneinander angeordnet werden müssen, so wie unserem Fall, fängt man oft mit einem vertikalen Hauptlayout an, da die meisten Elemente untereinander auftreten werden. Dieses können wir mit

```
haupt_layout = QVBoxLayout()
```

anlegen und dann unsere Unter-Layouts zum Hauptlayout hinzufügen, z.B.

```
haupt_layout.addLayout(dekl_layout)
```

Zuletzt müssen wir noch unserem Behälter das Hauptlayout zuweisen, was wir mit der Anweisung

```
behaelter.setLayout(haupt_layout)
```

erreichen.



Übung 42 - Inversion mit GUI, Teil 2

Definieren Sie jetzt die `initUI`-Methode.

Fügen Sie darin, wie gerade beschrieben, einen Behälter, ein Hauptlayout und drei Unterlayouts hinzu.

Zur Erinnerung: dabei soll das erste Unterlayout später die Elemente beinhalten, um einen einfachen, veränderlichen Deklarativsatz anzuzeigen, das zweite, um das interrogative Gegenstück (kopierbar) auszugeben, und das dritte, um den Knopf zum Konvertieren aufzunehmen.

Die Steuerelemente werden wir dann im nächsten Schritt hinzufügen.

9.4 Steuerelemente definieren und Layouts zuordnen

Alle Steuerelemente, außer dem Hauptfenster, müssen normalerweise einem Behälter zugewiesen sein. Der Name des Behälters kann als letztes Element explizit über


```
Parent=Elternname
```

beim Anlegen übergeben werden. Bei der Verwendung von Layouts wird jedoch automatisch das Layout zum Behälter, so dass dieser nicht mehr angegeben werden muss. Wichtig ist die Angabe aber unter Umständen bei Elementen, wie z.B. Nachrichtendialogen, die nicht fest in ein Layout eingebunden sind, sondern nur kurzzeitig angezeigt werden.

Steuerelemente, die im ganzen Programm zugänglich sein sollten, müssen als Instanzvariablen definiert sein, beispielsweise wenn ihre Eigenschaften ausgelesen oder verändert werden sollen, wie in unseren Ein- und Ausgabeelementen. Wir legen das erste mit

```
self.dekl_eingabe = QLineEdit('Dies ist ein Deklarativsatz.')
```

an, da ja der Text des Elements vom Benutzer veränderbar sein soll. Andere Elemente, die nur ‚lokale Funktion‘ haben, können einfach als normale Variablen angelegt werden. So können wir den Knopf zum Konvertieren einfach als

```
konvert = QPushButton('Konvertieren')
```

anlegen, da wir später direkt danach ein Signal mit dessen Slot verknüpfen werden, um ihn klickbar zu machen.

Das Hinzufügen der Elemente zu einem Layout erfolgt über die Methode `addWidget()`. Dabei kann man einfache Steuerelemente ohne veränderbare Eigenschaften oder Funktionen auch direkt beim Hinzufügen anlegen, ohne vorher eine Variable erzeugen zu müssen, z.B. für Beschreibungstexte wie in

```
dekl_layout.addWidget(QLabel('Deklarativsatz:\t'))
```

9.5 Eigenschaften, Methoden und Signale von Steuerelementen

Steuerelemente haben verschiedene Eigenschaften, die sich, je nach ihrem Zweck, mehr oder weniger kontrollieren lassen, z.B. Text oder Werte(bereiche), der/die sich auslesen oder setzen lassen, Listen zur Übersicht oder zu Auswahlzwecken, bei denen man auf die einzelnen Positionen zugreifen oder diese löschen kann, oder Optionen, die an- oder abgewählt werden können. Manche von ihnen können beim Erstellen über Argumente direkt initialisiert werden, andere nur über spezielle Methoden, was etwas umständlicher ist und mehr Code erfordert. Leider muss man dies teilweise selbst austesten, da die PyQt-Dokumentation oft nur die Zugriffsmethoden beschreibt, aber nicht die Namen der Argumente selbst. Allerdings habe ich schon versucht, einige der wichtigsten Argumente in dem Programm `widget_demo` aufzulisten, so dass Sie dieses schon teilweise als Referenz verwenden können.



Übung 43 - Inversion mit GUI, Teil 3

Starten Sie erneut das Programm `widget_demo`.

Klicken oder Doppelklicken Sie auf die jeweiligen Elementtypen, oder wählen Sie jeweils eine Option aus, so dass die Informationen dazu im *QTextEdit*-Fenster auf der linken Seite erscheinen.

Machen Sie sich, so gut es geht, mit den Eigenschaften, Methoden und nützlichsten Signalen der einzelnen Typen vertraut, insbesondere denen, die wir für unser aktuelles Programm benötigen.

Wenn Sie später selbst GUI-Programme erstellen, können Sie diese Informationen jederzeit wieder ‚nachschnellen‘, oder zusätzliche Informationen unter <https://doc.qt.io/qtforpython-5/> finden, da diese, auch wenn sie eigentlich für das etwas andere Modul `PySide2` gedacht sind, auch generell für `PyQt` zutreffen sollten.

Legen Sie nun die Steuerelemente, die wir für unser Programm benötigen, an. Diese sind:

- ein beschreibender Text für unser Eingabeelement für den Deklarativsatz
- das Eingabeelement mit Anfangstext `Dies ist ein Deklarativsatz`.
- ein beschreibender Text für die Ausgabe
- ein Ausgabefeld
- ein Knopf zum Auslösen der Konvertierung

Fügen Sie diese den jeweiligen Layouts hinzu.

9.6 Interaktive Funktionen hinzufügen

Steuerelemente reagieren auf verschiedene Ereignisse. Manche, so wie Knöpfe, (normalerweise) auf Klicks, Editoren auf Klicks, Doppelklicks oder Veränderungen im eingegebenen Text, Auswahllisten oder -knöpfe auf die Auswahl oder Veränderung von Optionen und Umschaltknöpfe auf das Umschalten von Zuständen. Solche Ereignisse können entweder über vordefinierte Signale des Elements zugeordnet werden oder den `mousePressEvent` für Klicks, falls kein passendes Signal vorhanden ist. Beim Aufruf von Methoden ohne Argumente ist die allgemeine Syntax

```
Elementvariable.Signalname.connect(Methodenname)
```

oder

```
Elementvariable.mousePressEvent = Methodenname
```

Bei Methoden, die Argumente verlangen, müssen allerdings Lambda-Anweisungen verwendet werden, und die Syntax ist entweder

```
Elementvariable.Signalname.connect(lambda x:[Klassenname.]Methoden-
name(Argument(e)))
```

oder

```
Elementvariable.mousePressEvent = lambda x:[Klassenname.]Methoden-
name(Argument(e))
```

Unser `QPushButton` hat als Standardsignal `clicked`, welches wir mittels des `connect()`-Slots mit einer Methode verbinden können. Diese Methode (*inversion*) definieren wir als Teil unseres `Inverter`-Objekts. Innerhalb dieser Methode, die später einfach als Argument an `connect()` übergeben wird, können wir auf den Text des Eingabefelds zugreifen, ihn umformen und dann dem Ausgabefeld zuweisen.

Übung 44 - Inversion mit GUI, Teil 4

Implementieren Sie die Methode `inversion`.

Extrahieren Sie hierfür den aktuellen Text aus dem Eingabefeld mittels der passenden Methode und verwenden Sie einen regulären Ausdruck mit geeigneten Gruppierungen, um über eine Suche das erste Wort, das zweite Wort und den Rest des Textes bis auf das Satzzeichen in einer Variable zu speichern.

Testen Sie dann ob der Eingabetext überhaupt eine Zeichenkette beinhaltet hatte und die `re`-Suche erfolgreich war.

Falls ja, setzen Sie die einzelnen Teile, die durch die Gruppierungen gefunden wurden, unter Beachtung der korrekten Groß- und Kleinschreibung und einem Fragezeichen zusammen, und weisen Sie das Ergebnis mittels der passenden Methode dem Ausgabefeld zu.

Falls nein, zeigen Sie die folgende Fehlermeldung mittels des vorgefertigten `Nachrichtendialogs (QMessageBox)` an:

```
QMessageBox(text = 'Bitte einen kompletten einfachen
Deklarativsatz eingeben!!', windowTitle = 'Eingabefehler', icon =
QMessageBox.Critical).exec_()
```

Testen Sie das Programm, wobei Sie wahlweise auch Fehler einbauen.



9.7 Aktionen

Aktionen in PyQt sind häufiger verwendete Arten von Funktionalität, die unter Umständen an mehreren Stellen im Programm verwendet werden, wie z.B. in Menüs oder auf Knöpfen. Sie werden über

```
QAction ([QIcon(Icon)], 'Menütext' [, shortcut='Ctrl+ X'][,
statusTip='Beschreibung für Statuszeile'][, tooltip='Beschreibung
als Tooltip'])
```

als Objekte angelegt, dann mit

```
addAction(Aktion)
```

oder

```
addAction(Aktionen)
```

Menüs oder Knopfleisten – siehe unten – zugewiesen, und über das Signal `triggered` mit Methoden verknüpft. Bei Verwendung in Knopfleisten (`QToolBar`) wird entweder der Menütext oder die Tooltip-Beschreibung als Tooltip verwendet, abhängig davon, ob Letztere definiert ist. Dabei sollten Menütext und Tooltip eher kurz gehalten sein, wohingegen die Ausgabe in der Statuszeile (falls Letztere vorhanden ist) auch länger sein kann.

9.7.1 Menüs, Knopf- und Statuszeilen erstellen

Menü und Knopfleisten sind normalerweise ein integraler Bestandteil größerer Fensterprogramme, wobei sie bei einfachen Dialogen, die ja nur für Rückmeldungen an die Benutzer dienen, nicht vorgesehen sind. Die Menüzeile, die unter Windows und Linux direkt unterhalb der Titelseite des Programms erscheint, wird einfach mithilfe

```
Hauptfenster.menuBar()
```

angelegt, wonach dann noch die einzelnen Menüs definiert und hinzugefügt werden müssen. MacOS jedoch zeigt das Menü normalerweise nicht als Teil des jeweiligen

Programmfensters an, sondern im Betriebssystemmenü. Um dies zu unterdrücken, können Sie die Anweisung

```
hauptMenu.setNativeMenuBar(False)
```

verwenden. Dadurch werden Mac-Benutzer sich zwar etwas umstellen müssen, was aber den Vorteil hat, dass ihr Programm unter allen Betriebssystemen gleich aussieht. Dies ist besonders von Vorteil, da man in einer eventuell zu schreibenden Dokumentation auch jeweils immer nur einen Screenshot benötigt, um die Funktionalität von Menüs zu beschreiben.

Haupt-, aber ebenso Untermenüs werden mithilfe von `addMenu()` angelegt, z.B.

```
dateiMenu = hauptMenu.addMenu('&Datei')
```

Wie oben beschrieben, fügt man dann Aktionen direkt mit

```
addAction()
```

oder

```
addActions(Liste)
```

dort hinzu. Da Menüs zwar hauptsächlich mit der Maus angewählt werden, aber im Notfall – und aus Effizienzgründen – auch über die Tastatur zu erreichen sein sollten, kann man einzelne Buchstaben innerhalb des jeweiligen Menüeintrags durch `&` markieren, so dass durch Drücken des jeweiligen Buchstabens der Menüeintrag ausgelöst bzw. bei Hauptmenüs das Menü selbst geöffnet wird. Um ein Hauptmenü anzuwählen, drückt man unter Windows und Linux die `Alt`-Taste und auf dem Mac die `Options`-Taste, woraufhin die auslösenden Buchstaben für die Hauptmenüs unterstrichen angezeigt und durch Drücken die Hauptmenüs aufgeklappt werden können. Einträge in Untermenüs kann man bei geöffnetem Hauptmenü direkt über den jeweiligen Buchstaben anwählen. Beim Anlegen solcher Zugriffsoptionen empfiehlt es sich, sich bei Standardeinträgen möglichst immer an etablierte Konventionen zu halten und bei nicht-standardisierten Menüeinträgen möglichst sinnvolle, mnemonische, Optionen zu wählen.

Ähnlich funktioniert das Anlegen von Knopfleisten, die meist mithilfe von

```
Hauptfenster.addToolBar('Name')
```

angelegt werden, wobei sie aber nicht unbedingt dem Hauptfenster zugeordnet sein müssen, sondern auch an anderen Stellen in Layouts eingebunden sein können. Auch kann man beliebig viele Knopfleisten in einem Programm anlegen, wodurch aber

die Arbeitsfläche verkleinert wird, die für andere Elemente zur Verfügung steht. Außer Aktionen können Knopfleisten auch andere Steuerelemente mit `addWidget()` zugewiesen werden, die dann über elementspezifische Signale oder das allgemeine Signal `actionTriggered` mit Methoden verknüpft werden.

Statuszeilen werden mit

```
Hauptfenster.statusBar()
```

angelegt und zeigen entweder die Statusinformationen von Aktionen an, oder können mittels `showMessage()` angewiesen werden, Nachrichten auszugeben. Auch ist es möglich, in ihnen permanente Steuerelemente anzulegen, um bestimmte Zustände anzuzeigen. So z.B. zeigt die Statuszeile in Textverarbeitungsprogrammen oft die Anzahl der Seiten, Wörter, und der jeweiligen Eingabesprache an. Allerdings haben nicht alle Fensterprogramme auch eine Statuszeile.

Wie wir oben gesehen haben, sind alle Hauptmenüs und Statuszeilen normalerweise immer dem jeweiligen Hauptfenster zugeordnet. Ein Ausnahme machen dabei komplette Programme, die als komplexe Steuerelemente in andere Programme eingebettet sein und dann Ihre eigenen Menüs oder Statuszeilen haben können.

9.8 Mit Dateien und Verzeichnissen in PyQt arbeiten

Um in PyQt-Programmen Benutzern zu erlauben, Datei- oder Verzeichnisnamen auszuwählen bzw. zu vergeben, verwendet man das `QFileDialog`-Steuerelement mit verschiedenen Methoden und Optionen zum Öffnen, Speichern, oder Erstellen von Dateien. Solche Dateiauswahldialoge sollten Sie eigentlich aus Ihrem eigenen Umgang mit Programmen kennen. Zum Auswählen von einzelnen Dateien gibt es die Methode `getOpenFileName` und, um mehrerer Dateien gleichzeitig auswählen zu können, `getOpenFileNames`, wohingegen zum Speichern, diesmal nur einzelner Dateien, `getSaveFileName` verwendet wird. Alle drei Methoden erwarten die gleichen (Schlüsselwort-)Argumente, eine Referenz auf eine aufrufende Fensterklasse (`parent`), einen Titel für das Dialogfenster (`caption`), einen Anfangspfad (`directory`) und optional einen Dateifilter (`filter`), z.B.

```
'XML-Dateien (*.xml)'
```

Für die Methoden zum Öffnen kann dieser Filter auch aus mehreren Elementen bestehen, die durch zwei Strichpunkte getrennt sind, etwa

```
'XML-Dateien (*.xml);; Textdateien (*.txt)'
```

Ist dies der Fall, dann kann man als weiteres Argument einen standardmäßig ausgewählten Filter (`initialFilter`) aus der Gruppe von Filtern angeben. Zu guter Letzt

gibt es auch noch eine Reihe von Flags, mit denen man den Dialog weiter einschränken kann und von denen wir später noch eines kennenlernen werden. Da diese Methoden jedoch in PyQt5 als Ergebnis ein Tupel zurückliefern, nämlich den/die Dateinamen und den gesetzten Filter, muss man diesen Tupel entpacken, um nur den/die Dateinamen zu erhalten. Dies tut man, indem man das zweite Element einer ‚Dummy-Variable‘ () zuweist, also einer Variable, die nie verwendet wird. Die allgemeine Syntax für die drei Optionen ist also jeweils:

```
dateiName, _ = QFileDialog.getOpenFileName(self, 'Datei öffnen',
    '.', filter='Extension;;Extension');
dateiNamen, _ = QFileDialog.getOpenFileNames(self, 'Datei(en) öff-
nen', '.', filter='Extension;;Extension');
dateiName, _ = QFileDialog.getSaveFileName(self, 'Datei speichern',
    '.', filter='Extension;;Extension', initialFilter='Extension').
```

Wie Sie in den Syntaxbeschreibungen sehen können, müssen die Schlüsselwörter dabei nicht angegeben werden, insofern das jeweilige Argument eindeutig aufgrund der Position erkannt werden kann. Zur Auswahl von Verzeichnissen können wir am einfachsten die folgende Syntax verwenden:

```
verzeichnisName = QFileDialog.getExistingDirectory(self, 'Verzeich-
nis auswählen', '.', QFileDialog.ShowDirsOnly)
```

Hier verwenden wir nicht nur die Methode `getExistingDirectory`, sondern auch das Flag `QFileDialog.ShowDirsOnly`.

Sobald die Datei- oder Verzeichnisnamen, die wir über den `QFileDialog` ausgewählt haben, in einfachen Variablen oder Listen abgespeichert sind, kann man mithilfe von Standarddatei- und Verzeichnisoperationen damit arbeiten. Falls ausgewählte Dateien nicht direkt für Operationen geöffnet werden sollen, bietet es sich jedoch meist an, diese in einem `QListWidget` zur weiteren Auswahl/Verwendung aufzulisten, wobei bei Verzeichnissen normalerweise ausgelesen und ggf. gefiltert werden muss.

Nachdem wir jetzt gelernt haben, wie wir unsere Programme durch GUIs flexibler und benutzerfreundlicher gestalten, wollen wir uns im letzten Hauptkapitel damit beschäftigen, wie man mit Webdaten und Annotationen umgeht.



9.9 Lösungen zu den Aufgaben

Lösung 40 - Steuerelemente verstehen lernen

Um ein Programm zu entwickeln, welches uns erlaubt, die syntaktische Inversion einfacher Deklarativsätze zu simulieren, so wie wir es vorher auf der Kommandozeile getan haben, benötigt man minimal Ein- und Ausgabeelemente sowie eine Möglichkeit, die Inversion tatsächlich auch auszulösen. Man könnte dafür zwar theoretisch auch nur ein einziges Element für die Ein- und Ausgabe verwenden, aber das wäre weniger übersichtlich, da man dann entweder immer nur den Ausgangssatz oder das Endprodukt der Inversion sehen würde. Zudem sollte man idealerweise auch noch Texte haben, die die Ein- und Ausgabeelemente beschreiben. So ist auf den ersten Blick erkennbar, welche Steuerelemente wozu dienen und die Benutzer und Benutzerinnen benötigen keine zusätzliche Dokumentation für ein relativ einfaches Programm.

Zur Ein- und Ausgabe bieten sich jeweils zwei `QLineEdit`s an, da wir nur kurze, einzeilige Texte haben, zum Auslösen der Inversion ein `QPushButton` und für die Beschreibungen jeweils ein `QLabel`. Dabei sollten die Beschreibungen per Konvention jeweils links vom dazugehörigen Ein- oder Ausgabeelement stehen, die Ausgabe unter der Eingabe und der Auslöseknopf ganz unten. Damit ergibt sich folgendes Layout.

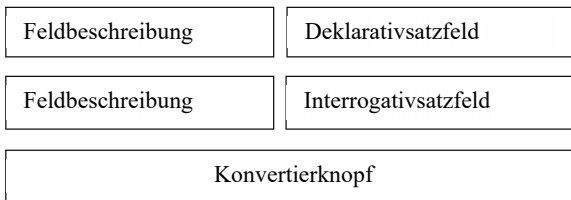


Abb. 5: Layout für GUI-Inversion

Beim Klicken des Knopfes muss zunächst der Text im Eingabefeld (mit Fehlerbehandlung) ausgelesen werden. Dann wird eine Methode für die Inversion ausgelöst, die diesen Text als Argument übergeben bekommt und invertiert zurückliefert, damit der Text ins Ausgabefeld eingetragen oder eine schon bestehende Ausgabe überschrieben werden kann.

Die Fehlerbehandlung sollte erfassen, ob überhaupt ein Text in dem Deklarativfeld auftritt und dieser auf einen Punkt endet.

Lösung 41 - Inversion mit GUI, Teil 1

Die Importe von `sys` und `re` haben wir jetzt schon sehr oft durchgeführt, so dass Sie mittlerweile keinerlei Schwierigkeiten mehr bereiten sollten. Auch die anderen Importe mittels `from Modulname import` dürften Ihnen nicht weiter schwerfallen. Sollten Sie dabei ein Element vergessen oder falsch schreiben, wird Ihnen die WingIDE später

dafür keine automatische Vervollständigung anbieten und spätestens beim Starten des Programms eine Fehlermeldung auftreten. Anfänglich jedoch werden die meisten Elemente, insofern sie noch nicht im weiteren Programmcode auftreten, erst einmal gelb unterringelt erscheinen. Bei Ihren eigenen Programmen können Sie sich später diese Funktionalität auch zunutze machen, indem Sie eventuell zu viel geladene Modulteile, die nach Fertigstellung und Testen des Programms immer noch so markiert sind, löschen, da sie ja nicht im Programm gebraucht wurden.

Die Klassendeklaration sollte, da wir von einem Hauptfenster ableiten, wie folgt aussehen:

```
class Inverter(QMainWindow):
```

Und die Definition der `__init__`-Methode:

```
def __init__(self):
    # Konstruktor der Elternklasse aufrufen
    super().__init__()
    self.setFont(QFont("Courier", 12))
    self.initUI()
```

Dieses Format werden Sie wahrscheinlich fast immer verwenden, wenn Sie aus einer bestehenden Steuerelementklasse ableiten. Sie können es sich unter Umständen auch als Vorlage in Textformat abspeichern, da die Personal Edition der WingIDE leider die Verwendung von Textschnipseln, die innerhalb des Programms direkt eingefügt werden können, nicht bietet. Allerdings wird unter Umständen der Teil, wo die Fonteigenschaften gesetzt werden, bei größeren Programmen, insbesondere wenn Benutzer selbst das Aussehen festlegen sollen, vielleicht nicht benötigt, sondern eher in einer `initUI`-Methode definiert oder sogar über Steuerelemente veränderbar sein.

Lösung 42 - Inversion mit GUI, Teil 2

Die ersten Schritte bis zum Anlegen des Unterlayouts für die Eingabeelemente des Deklarativsatzes können Sie im Prinzip einfach von der Beschreibung oben im Text übernehmen. Sie müssen dann nur noch die Teile zum Anlegen des Layouts für die Ausgabe und das Auslösen der Konversion selbst zwischendrin schreiben, so dass das komplette Anlegen der Layoutstruktur wie folgt aussehen könnte:

```
# Behälter für Layouts anlegen
behaelter = QFrame()

# Behälter als zentrales Steuerelement setzen
self.setCentralWidget(behaelter)

# Haupt- und Unterlayouts definieren
haupt_layout = QVBoxLayout()
dekl_layout = QHBoxLayout()
```

```

interr_layout = QHBoxLayout()
konv_layout = QHBoxLayout()
# Unter-Layouts zum Hauptlayout hinzufügen
haupt_layout.addLayout(dekl_layout)
haupt_layout.addLayout(interr_layout)
haupt_layout.addLayout(konv_layout)
# Behälter Hauptlayout zuweisen
behaelter.setLayout(haupt_layout)

```

Damit bleiben dann nur noch zwei Schritte zum Anlegen der Elemente übrig, die Definition der Steuerelemente selbst und deren Hinzufügen zu den jeweiligen Layouts. Außerdem benötigen wir, um das Programm fertigzustellen, noch die Methode, die tatsächlich die Inversion durchführt und die Ausgabe produziert. Wenn Sie wollen, können Sie das Programm auch schon austesten, wobei das Layout noch nicht deutlich zu sehen ist, da sich noch keine Steuerelemente in den einzelnen Unterlayouts befinden. Ihnen wird daher nur ein Programmfenster mit angezeigtem Text aus dem `__main__`-Block und der dort definierten Positionierung angezeigt. Je mehr wir das Programm jedoch weiterentwickeln, umso mehr wird dort auch Schritt für Schritt zu sehen sein.

Lösung 43 - Inversion mit GUI, Teil 3

Beim Definieren und Anlegen der Steuerelemente müssen wir als erstes überlegen, welche Arten von Elementen wir sinnvollerweise verwenden sollten, und inwiefern wir auf diese über die Methoden der Klasse zugreifen können müssen. Falls die Eigenschaften von Elementen im ganzen Programm lesbar oder veränderbar sein sollen, dann müssen diese jeweiligen Elemente auch als Instanzvariablen mit `self.Elementname` angelegt werden.

In unserem Programm verwenden wir am sinnvollsten, wie wir ja schon oben angesprochen haben, jeweils ein `QLabel` für die beschreibenden Texte, ein `QLineEdit` für die Ein- und Ausgabe sowie einen `QPushButton`, um die Konversion auszuführen. Dabei müssen aber nur die Ein- und Ausgabeelemente auch als Klassenvariablen definiert werden, da auf deren jeweiligen Text lesend oder schreibend zugegriffen werden muss. Das heißt, wir müssen später den Text im Eingabefeld mit der passenden Methode auslesen und den im Ausgabefeld mit deren Gegenstück setzen können. Der Knopf zum Auslösen selbst muss dabei nicht als Klassenvariable angelegt werden. Wir können ihm später direkt nach der Deklaration einen entsprechenden Slot zuweisen, was wir im nächsten Unterabschnitt besprechen werden.

Variablen müssen überhaupt nur für die Ein- und Ausgabefelder und den Knopf angelegt werden. Nur diese müssen weiterhin im Programm ansprechbar sein und somit die entsprechenden Instanzvariablen dem Layout hinzugefügt werden. Dahingegen können die beschreibenden Texte einfach direkt beim Hinzufügen zum Layout

,anonym‘ angelegt werden. Dieser Programmteil könnte dann folgendermaßen aussehen:

```
#Steuerelemente definieren
self.dekl_eingabe = QLineEdit('Dies ist ein Deklarativsatz.')
self.interr_ausgabe = QLineEdit()
konverter = QPushButton('Konvertieren')
# Signal mit Slot verknüpfen
# hier muss noch später die Verknüpfung erfolgen

# Steuerelemente in Layouts einfügen
dekl_layout.addWidget(QLabel('Deklarativsatz:\t'))
dekl_layout.addWidget(self.dekl_eingabe)
konv_layout.addWidget(konverter)
interr_layout.addWidget(QLabel('Interrogativsatz:'))
interr_layout.addWidget(self.interr_ausgabe)
```

Wie oben im Kommentar angezeigt, müssen wir später noch die Verknüpfung des Slots mit dem passenden Signal für den Knopf einfügen.

Lösung 44 - Inversion mit GUI, Teil 4

Unseren Ausdruck sollten wir als rohe Zeichenkette definieren, um nicht zu viele Sonderzeichen maskieren zu müssen. Er sollte so aussehen, damit Sie die erforderlichen Gruppen extrahieren können:

```
r'^\s*(\w+\b) (\b\w+\b) (.+?)\.$'
```

Dabei verankern wir die Gruppe für das erste Wort am Anfang, die zweite nach dem ersten Leerzeichen und die dritte deckt alle Zeichen ab, die vor einem Punkt am Zeilenende stehen. Diesen Ausdruck können wir direkt innerhalb einer Regex-Suche auf den Text im Eingabefeld anwenden, auf den wir mittels

```
self.dekl_eingabe.text()
```

zugreifen, da die Methode `text()` ja den eingegebenen Text in einem `QLineEdit`-Element zurückliefert. Wenn wir diese Variable `teile` nennen, dann können wir später leicht über

```
teile.group(n)
```

auf die einzelnen Gruppen zugreifen, wobei mit `n=1` auf das erste Wort, `n=2` das zweite und `n=3` auf den Rest des Satzes zugegriffen werden kann, um die ersten zwei Wörter zu vertauschen, den Satz dann zusammenfügen und mittels der `setText()`-Methode dem Ausgabefeld zuweisen zu können. Dabei sollten wir auch das erste Wort mit

großgeschriebenem Anfangsbuchstaben ausgeben, das zweite verkleinert, und am Ende noch ein Fragezeichen anfügen. Dies können wir mit

```
self.interr_ausgabe.setText(
    f'{teile.group(2).capitalize()} '
    f'{teile.group(1).lower()}{teile.group(3)}?')
```

elegant erreichen.

Bevor wir allerdings den Text zusammenfügen und ausgeben, müssen wir sicherstellen, dass auch tatsächlich Text im Eingabefeld steht und dieser einen Deklarativsatz enthält. Dafür benötigen wir nur eine einfache Fehlerbehandlung über den Bedingungsblock

```
if self.dekl_eingabe.text() and teile:
```

Dabei wird im `else`-Block dann die `QMessageBox` verwendet, um den Benutzer darauf hinzuweisen, dass die Eingabe nicht korrekt war und was tatsächlich erwartet wurde. Der erste Teil der `if`-Anweisung überprüft, ob überhaupt Text im Eingabefeld steht, da ansonsten `None` zurückgeliefert würde, und der zweite, ob unser Ausdruck ein Ergebnis zurückgeliefert hat.

10 Webdaten und Annotationen



Die Arbeit mit Daten aus dem Web wird bei Sprachwissenschaftlern aufgrund der Einfachheit, wie man dort Daten finden kann, aber auch anderen Gründen, immer populärer. Deshalb wird in diesem Kapitel zunächst aufgezeigt, wie solche Daten aufgebaut und mit HTML ausgezeichnet sind. Sobald dann ein grundlegendes Verständnis für Auszeichnungssprachen besteht, wird danach im Vergleich das für sprachwissenschaftliche Zwecke nützlichere XML als Möglichkeit zur Annotierung von Textdaten ebenfalls vorgestellt. Außerdem lernen wir, wie mithilfe geeigneter Bibliotheken Daten aus dem Internet heruntergeladen werden und schließlich Texte zu XML konvertiert werden können.

10.1 Webdaten und Annotierungen

Daten, die aus dem Internet heruntergeladen werden können, werden aufgrund leichter und umfangreicher Verfügbarkeit nicht nur immer beliebter, sondern stellen z.T. auch schon einen eigenen Forschungsbereich dar. Python bietet zum Arbeiten mit Webdaten oder URLs das `urllib`-Paket, und Qt sogar Komponenten, die als Browser fungieren, so dass man sich damit einen auf eigene Bedürfnisse zugeschnittenen Browser programmieren kann.

Webseiten selbst sind normalerweise in **HTML (HyperText Markup Language)** geschrieben, einer im Umfang relativ beschränkten **Auszeichnungssprache** (engl. **markup language**), aus der sich relativ einfach Textdaten extrahieren lassen. Natürlich können aber auch andere Daten, wie z.B. PDFs, direkt aus dem Internet heruntergeladen werden. Allerdings ist die Extraktion von Texten daraus, selbst wenn es dafür auch Module gibt, nicht immer ganz so einfach oder qualitativ hochwertig, so dass man bei derartigen Daten häufig noch sehr viel manuell nachbessern muss. Im Gegensatz zu den HTML-Auszeichnungen auf Webseiten, die hauptsächlich zur Darstellung der Informationen dienen, verwendet man für linguistische **Annotierungen**, das heißt Auszeichnung und Anreicherung mit linguistisch relevanten Informationen oder Interpretationen, meist **XML (eXtensible Markup Language)**. Das Wort *extensible*, also *erweiterbar*, deutet hier schon darauf hin, dass diese Auszeichnungssprache nicht nur eine begrenzte Anzahl an festgelegten Funktionselementen zur Verfügung stellt, sondern nahezu beliebig erweiterbar ist, vorausgesetzt dass einige grundsätzliche formale Bedingungen erfüllt sind.

10.2 Auszeichnungssprachen

Auszeichnungssprachen wie HTML oder XML verwenden Kennzeichnungen in spitzen Klammern (`<...>`), genannt **Elemente** oder **Tags** (nicht zu verwechseln mit morpho-syntaktischen Tags), zur strukturellen und linguistischen Auszeichnung. Diese eignen sich sehr gut für die Auszeichnung, da sie selten selbst in Texten auftreten und einfach zu erkennen sind, insbesondere weil sie paarweise, mit einem öffnenden und einem schließenden Teil, auftreten. Sollten sie dennoch einmal im Text vorkommen, müssen sie durch sogenannte **Entities**, mithilfe von `<` für `<` und `>` für `>`, maskiert werden. Dies gilt ebenfalls für das Ampersand (`&`), da es ja innerhalb der Entities auftreten kann, sowie auch einige andere Sonderzeichen.

Es gibt zwei Typen von Elementen. Einen, den wir als **Tagklammern** bezeichnen können, und einen, der als **leere Elemente** bezeichnet wird. Tagklammern enthalten andere Tags oder Text und müssen immer paarweise auftreten; das heißt, sie umschließen Inhalte. Das allgemeine Format ist also `<Tagname>...</Tagname>`, wobei das Ende des Elements, der sogenannte **End-Tag**, im Gegensatz zum Anfang, dem **Start-Tag**, durch den Schrägstrich direkt hinter der öffnenden spitzen Klammer gekennzeichnet ist. Leere Elemente jedoch stellen primär Instruktionen für den Browser dar oder beinhalten ‚Nicht-Text-Informationen‘ und keine anderen Tags. Ihr allgemeines Format ist `<Tag>` oder `<Tag />`, z.B. `
` für einen Zeilenumbruch in HTML, wobei der Schrägstrich vor der schließenden Klammer in XML verbindlich ist, aber das Leerzeichen davor nicht. Jedoch bietet es sich an, das Leerzeichen der Übersichtlichkeit halber immer zu schreiben.

Beide Typen können zusätzliche Informationen in Form von Attribut-Wertepaaren beinhalten, bei Tagklammern allerdings nur im Start-Tag. Die Attribute sind hier mit ihren Werten über ein Istgleich-Zeichen (`=`) verknüpft, und die Werte müssen, wie bei Zeichenketten in Python, entweder in (komplementären) einfachen oder doppelten Anführungszeichen stehen.

10.3 HTML Kurzeinführung

Webseiteninhalte bestehen normalerweise aus zwei Elementen, einem Seitenkopf (`<head>...</head>`), der Meta-Informationen wie Seitentitel, Zeichensatz etc. enthält, und dem eigentlichen Seitentext (`<body>...</body>`). Die Haupttextelemente des Seitentextes, die eigentlich auf fast jeder Seite vorkommen, sind Absätze (engl. *paragraph*; `<p>...</p>`) und Überschriften (engl. *heading*; `<h>...</h>`), wobei das *n* Werte von eins bis sechs annehmen kann, da bis zu sechs Überschriftentiefen zur Textstrukturierung definiert sind. In der Praxis werden jedoch selten mehr als drei oder vier verwendet.

Zur zusätzlichen Strukturierung gibt es auch andere Arten von Textabschnitten. Hier finden wir zum einen Abschnitte/Sektionen (engl. *division*; `<div>...</div>`), die mehrere Absätze gruppieren können, und Spannen (engl. *span*; `...`), die

dazu dienen, kürzere Texteinheiten, z.B. zur Hervorhebung oder besonderen Formatierung, auszuzeichnen. Zum anderen existieren aber auch Listen und Tabellen. Bei Ersteren unterscheidet man zwischen geordneten Listen (engl. *ordered lists*; `...`), in denen die Listenelemente nummeriert sind, und ungeordneten (engl. *unordered lists*; `...`), wo die Listenelemente nur als solche markiert sind. Innerhalb der Listenstrukturen werden die einzelnen Listenelemente (engl. *list items*) durch `...` ausgezeichnet. Eine Verschachtelung eines oder beider Typen ist ebenso möglich, so dass verzweigte Strukturen mit verschiedenen Ebenen erzeugt werden können, wie z.B. bei Menüs zur Navigation.

Eine weitere Art, Text strukturiert darzustellen, bieten Tabellen, welche minimal mit dem Tag `<table>...</table>` ausgezeichnet werden und darunter in Reihen (engl. *table rows*; `<tr>...</tr>`) unterteilt sind. Diese wiederum beinhalten einzelne Zellen mit Tabellendaten (engl. *table data*; `<td>...</td>`). In neueren Versionen von HTML gibt es auch noch zusätzliche Elemente, die bestimmte Teile der Tabelle beschreiben, auf die wir aber hier nicht näher eingehen wollen.

Als Basis für Hyperlinks, die auf Elemente innerhalb und außerhalb der Seite verweisen und diese beim Anklicken aufrufen, dienen Anker (engl. *anchor*). Diese haben die allgemeine Form `<a [id="..."] href="URL">Beschreibungstext`, wobei das optionale `id`-Attribut, falls vorhanden, normalerweise Positionen innerhalb von Seiten markiert und das `href`-Attribut (engl. *hypertext reference*) das Ziel des Links in Form einer URL (engl. *Uniform Resource Locator*), also einer Webadresse, beinhaltet. Auf ähnliche Weise können auch Multimedia-Inhalte in Webseiten eingebunden werden. Für Bilder gab es hier schon immer einen eigenen Tag, ``, wo hier anstelle des `href`-Attributs das `src`-Attribut (engl. *source*) zur Verwendung kommt und optional das `alt`-Attribut (engl. *alternative*) einen alternativen Beschreibungstext anzeigt, falls die Anzeige von Bildern im Browser deaktiviert ist oder diese lange zum Laden benötigen. In älteren Versionen von HTML wurden Audio-/Videodateien einfach mittels Hyperlinks in die Seite eingebettet. Seit HTML5 jedoch gibt es separate Tags, die Audio- (`<audio>...</audio>`) und Videoinhalte (`<video>...</video>`) kennzeichnen, wobei dann über eingebettete `<source src='URL'>...</source>` Tags auf die eigentlichen Inhalte verwiesen und zugegriffen wird.

Neben den oben besprochenen strukturellen Optionen gibt es zur Formatierung von Webseiten diverse Möglichkeiten, um Text- oder Seitenteile besonders hervorzuheben oder visuell voneinander abzugrenzen. Zur direkten Zeichenformatierung kann man die Tags `...` (fett; engl. *bold*), `...`, oder `<i>...</i>` (kursiv; engl. *italics*) verwenden, wobei die ersten beiden normalerweise von Browsern standardmäßig gleich dargestellt werden. Mithilfe von internen oder externen **Stylesheets** und/oder Formatierungsanweisungen für Tags oder Spans kann man jedoch das genaue Aussehen von Textteilen noch mehr beeinflussen oder umdefinieren, z.B. in Bezug auf Hintergrund- oder Textfarbe, Umrahmungen etc. Das Aussehen einzelnen Textteile oder bestimmter Tagtypen kann man dabei auch über `class`- oder `id`-Attribute

gezielt steuern, was wir aber hier nicht genauer besprechen werden. Dies ist für die Textextraktion normalerweise nicht relevant, weil Textformatierungen selten auch ein Ausdruck der Textsemantik sind, wohingegen strukturelle Absätze oder Überschriften spezielle Funktionen innerhalb von Texten erfüllen.

10.4 Webseiten herunterladen

Bevor wir jedoch lernen, wie wir in Python mit HTML-Seiten arbeiten können, müssen wir erst noch besprechen, wie man diese herunterladen kann. Zwar kann man für einfache und begrenzte Zwecke solche Seiten einfach aus dem Browser speichern, aber Python bietet ebenso Möglichkeiten, auf Dateien im Internet zuzugreifen, ähnlich, wie wir dies schon für lokale Dateien getan haben. Am einfachsten geht dies für unsere Zwecke mithilfe des Moduls `urllib.request`, welches über die Methode `urlopen` und ein `http.client.HTTPResponse`-Objekt eine Verbindung zu einer Webseite herstellt oder, falls dies nicht möglich ist, einen `urllib.error.URLError` generiert.

Das Response-Objekt besteht allerdings nicht aus einer Zeichenkette, sondern aus Bytes, weshalb man es beim Zugriff darauf über die Methode `read()` erst einmal dekodieren muss. Das heißt, man muss also die von der Methode zurückgelieferten Bytes in eine Zeichenkette wandeln, indem man die `decode()`-Methode des Zeichenkettenobjekts darauf anwendet, welche als Argument eine Kodierung zum Dekodieren erwartet. Um diese Kodierung zu identifizieren, muss man dem Response-Objekt zunächst die relevanten Header-Informationen extrahieren, was man mittels

```
Response-objekt.headers.get_content_charset([failobj="Kodierung"])
```

erreichen kann. Hierbei erlaubt uns die Verwendung des optionalen `failobj`-Schlüsselworts, eine Kodierung als Notfalloption anzugeben, falls im Objekt die nötige Information nicht mitgeliefert wird. Wie wir dies tun, probieren wir in der nächsten Übung aus.



Übung 45 - Webseiten-Download

Schreiben Sie das Programm `23_seite_holen.py`, welches mithilfe `urllib.request` die Webseite `http://martinweisser.org/pract_cl/HTML_download_test_de.html` herunterlädt und abspeichert. Dabei sollten auch immer bei Zwischenschritten geeignete Meldungen an den Benutzer ausgegeben werden. Legen Sie zunächst eine Variable für die Adresse an, die Sie später unter Umständen auch verändern können.

Verwenden Sie dann die Methode `urlopen` des Moduls mit der Adressvariable als Argument innerhalb eines `try`-Blocks, um eine Verbindung zur Seite herzustellen und in einer geeigneten Variable abzulegen.

Fangen Sie einen eventuellen `urllib.error.URLError`-Fehler in einer Ausnahme auf und beenden Sie das Programm mit einer passenden Fehlermeldung. Der Grund für den Fehlschlag kann dabei aus dem vom `URLError` zurückgelieferten Fehler über das Attribut `reason` des Fehlerobjekts extrahiert werden.

Versuchen Sie dann, die Header-Information über die Kodierung abzufragen und in geeigneten Variable abzuspeichern. Als Notfalloption geben Sie dabei an, da die Mehrzahl der Seiten ja heutzutage darin kodiert ist.

Versuchen Sie jetzt, die Methodensequenz `read().decode()` mit der Kodierung auf die Seitenvariable anzuwenden, um den Seiteninhalt auszulesen und einer Zeichenkettenvariable zuzuweisen. Falls dies nicht funktionieren sollte, fangen Sie einen Fehler der Fehleroberklasse `Exception` ab und brechen Sie das komplette Programm mit einer geeigneten Fehlermeldung ab, da wir dann keine Möglichkeit mehr haben, die Seite abzuspeichern.

Versuchen Sie dann, den Titel mittels `re`-Suche aus dem Seiteninhalt zu extrahieren, falls dieser vorhanden ist. Um ihn zu finden, verwenden Sie einen Ausdruck, der diesen über den Inhalt des entsprechenden Tags extrahiert.

Falls ein Titel gefunden wurde, erzeugen Sie daraus einen Dateinamen, wobei Sie eventuelle Leerzeichen darin durch Unterstriche ersetzen und die Extension `html` anfügen sollten.

Sonst ermitteln Sie mittels der Methode `rindex()` und der Adresse die Position des Seitennamens und extrahieren diesen aus der URL, um ihn für den Dateinamen zu verwenden.

Speichern Sie danach den Seiteninhalt in einer Datei ab.

Testen Sie dann das Programm.

Wenn wir Webseiten herunterladen, dann wollen wir daraus natürlich auch nützliche Textteile zur Analyse extrahieren. Zu diesem Zweck könnten wir selbstverständlich unser oben erworbenes Wissen über die möglichen Tags verwenden, um mithilfe regulärer Ausdrücke zum Ziel zu gelangen. Da aber manche HTML-Seiten aufgrund des Auftretens vieler Attribute und starker Verschachtelungen der Elemente sehr komplex sein können, bietet es sich an, ein schon existierendes Modul zu verwenden, das uns einen großen Teil dieser Arbeit erleichtert.

Das Modul, welches wir hier verwenden wollen, heißt *Beautiful Soup* und existiert mittlerweile schon in seiner vierten Version. Sie können es mit `pip(3) install beautifulsoup4` installieren, wobei Sie aber zum späteren Import in Ihre Programme den abgekürzten Namen `bs4` verwenden müssen. Für die Verarbeitung von Webseiten erzeugt man aus der jeweiligen Seite ein `BeautifulSoup`-Objekt, das einen geparsten

Baum der Seite beinhaltet. Es genügt für unsere Zwecke, wenn Sie nur dieses eine Objekt mit `from bs4 import BeautifulSoup` importieren. Um das Seitenobjekt selbst anzulegen, übergibt man als erstes Argument für den Konstruktor die Seite und als zweites einen Parsertypen. BeautifulSoup unterstützt verschiedene Parsertypen, von denen aber nicht alle von vornherein installiert sind und deshalb bei Bedarf nachinstalliert werden müssen. Für unsere nächste Übung verwenden wir den schon vorinstallierten `html.parser`.

Beautiful Soup kann verwendet werden, um Elemente oder Text in Webseiten zu finden, aber auch zu manipulieren oder darin zu navigieren. Da wir aber nur an Analysen interessiert sind, ignorieren wir Letztere hier und besprechen nur einige wenige Möglichkeiten, Textdaten aufzufinden und zu extrahieren. Die einfachste Methode, um relevante Textteile zu finden, ist `find`, was die jeweils erste Instanz eines gesuchten Tags, Attributs oder Texts findet, wobei neben Zeichenketten auch `re`-Objekte zur Suche angegeben werden können. Falls nichts gefunden wird, dann ist der Rückgabewert `None`, bei Suchen nach Tags oder Attributen ein `Tag`-Objekt und bei Textsuchen – mittels des Schlüsselwortarguments `string` – ein `NavigableString`-Objekt. Dabei versucht `find`, das erste Argument als Tag zu interpretieren, falls ein entsprechender Tagname – ohne umschließende spitze Klammern – definiert ist. Falls darauf ein anderes Argument folgt, welches kein Schlüsselwortargument ist, wird dies normalerweise als Attribut interpretiert. In HTML fest definierte Attributnamen, wie z.B. `class`, `id` oder `name` können auch als Schlüsselwortargumente verwendet werden. Allerdings muss bei der Suche nach dem `class`-Attribut `class_` als Schlüsselwort verwendet werden, da `class` selbst ja ein Schlüsselwort in Python ist.

Dieselben Argumente kann man auch für die Methode `find_all()` angeben, welche eine Liste von Ergebnissen zurückliefert. Zudem erlaubt es diese Methode auch, nach Listen von Tags zu suchen sowie die Suche durch die Angabe von `limit` zu begrenzen. Sobald man mithilfe einer dieser Suchmethoden geeignete Textteile gefunden hat, kann man den Text der einzelnen gefundenen Elemente über die Methode `get_text()` extrahieren. Will man nur den gesamten Text extrahieren, lässt sich diese Methode übrigens auch auf das gesamte Baum-Objekt anwenden.

In unserer nächsten Übung wollen wir mit `find_all()` nur Absätze aus der in der letzten Übung heruntergeladenen Webseite extrahieren und einfach ausgeben lassen. Da wir jedoch dafür die Methode `get_text()` auf jedes einzelne gefundene Element anwenden müssen, um den Text selbst aus den einzelnen Listenelementen zu extrahieren, lernen wir zunächst noch eine effizientere Option kennen, alle Listenelemente auf einmal zu transformieren.

10.5 List und Dictionary comprehension

Um alle Elemente in Listen zu transformieren und/oder zu filtern, bietet Python eine sehr elegante Methode. Diese ermöglicht es, aus bestehenden Listen neue, veränderte,

zu generieren, ohne dass Sie dabei umständlich eine oder mehrere Schleifen schreiben müssen. Dies nennt man im Englischen *list comprehension*, zu Deutsch also ‚Listenverständnis‘, und lässt sich am besten durch ein einfachstes Beispiel erklären.

Nehmen wir an, dass wir eine Liste von Wörtern haben, die wir zum Vergleich, wie in unseren Wortlisten, verkleinern wollen, aber nur dann, wenn diese Wörter auch mit einem bestimmten Buchstaben anfangen. Der eine Teil dieser Aufgabe wäre also eine Transformation und der andere eine Filterung, wobei die Filterung normalerweise zuerst erfolgen sollte. Um das Ergebnis zu erzielen, könnten wir eine `for`-Schleife über die ursprüngliche Wortliste laufen lassen, in der dann mittels einer verschachtelten `if`-Abfrage nur die in Frage kommenden Wörter herausgefiltert und transformiert würden. Schematisch gesehen würde dies also folgendermaßen aussehen:

```
for wort in wortliste1:
    if wort.startswith('Buchstabe'):
        wortliste2.append(wort.lower())
```

Dies kann man in Python allerdings viel knapper und konziser als Einzeiler formulieren, nämlich als

```
wortliste2 = [wort.lower() for wort in wortliste1 if
              wort.startswith('Buchstabe')]
```

Da hiermit eine neue Liste erzeugt wird, muss der gesamte Ausdruck auf der rechten Seite in eckigen Klammern stehen.

Auf ähnliche Weise könnten wir aus derselben Liste und mit demselben Filter auch einen Dictionary erzeugen, bei dem jedem Wort seine verkleinerte Form zugeordnet wird. Dazu müssten wir das schematische Beispiel nur leicht zu

```
wort_dict = {wort:wort.lower() for wort in wortliste1 if
             wort.startswith('Buchstabe')}
```

abändern. In diesem Fall müssen wir natürlich geschweifte Klammern verwenden, da wir einen Dictionary erzeugen. Mit diesem Wissen können wir jetzt unser nächstes Programm angehen.

Übung 46 - Textextraktion aus Webseiten

Schreiben Sie ein neues Programm `24_absaetze_extrahieren.py`, in dem Sie alle Absätze aus der vorher heruntergeladenen Seite extrahieren.

Importieren Sie dazu zunächst das benötigte Objekt aus dem `Beautiful Soup`-Modul.

Öffnen Sie dann, diesmal ohne Fehlerbehandlung, die heruntergeladene Webseite und erzeugen Sie ein Baumobjekt, wobei Sie als Parser `html.parser` angeben.



Erzeugen Sie daraufhin mittels *list comprehension* eine Absatzliste, indem Sie aus allen Absätzen, die Sie mit `find_all` aus dem Baum extrahieren, den Text extrahieren, so wie wir dies oben besprochen hatten. Geben Sie zuletzt diese Liste zeilenweise in einer einzelnen `print`-Anweisung aus.

Nachdem wir jetzt wissen, wie wir mit HTML-annotierten Daten umgehen können, wenden wir uns nun der linguistischen Annotierung mit XML zu.

10.6 Kurzeinführung in XML

In XML sind die Tags, ungleich HTML, nicht fest definiert, sondern können nach Bedarf erzeugt werden, insofern man nicht ein schon vordefiniertes Standardformat verwendet, womit wir uns hier nicht näher beschäftigen werden. Um es Ihnen selbst und anderen, die später vielleicht mit Ihren Annotierungen arbeiten werden, zu vereinfachen, sollten Ihre Tags möglichst selbsterklärend sein. Dabei muss man heutzutage, wo Speicherplatz und die Geschwindigkeit von Computern keine Probleme mehr darstellen, auch nicht mehr so kurze Tags verwenden, wie die, die schon in den 1990iger Jahren für HTML definiert wurden, wo diese Ressourcen noch äußerst knapp waren. Allerdings sollte man trotzdem relativ konzis bleiben, wie auch bei der Definition von Variablen in Python.

Aber selbst wenn man bei der Definition der Tags sehr flexibel ist, muss man bei der Erzeugung eines XML-Dokuments immer noch einige wenige formelle Dinge beachten. Das wichtigste Konzept hier ist die **Wohlgeformtheit**, die besagt, dass ein Dokument

1. mit der XML-Deklaration `<?xml version="1.0"?>` beginnen muss;
2. einen ‚Behälter‘-Tag um den Inhalt, z.B. `<text>...</text>`, haben muss;
3. keine überlappenden Tags beinhalten darf;
4. und leere Elemente vor schließender Klammer einen `/` haben müssen, z.B.
5. `<pause />`.

Ist ein XML-Dokument wohlgeformt, so ist es auch ein gültiges XML-Dokument, insofern seine **Validität** nicht durch Zusatzregeln weiter eingeschränkt ist, was hier aber nicht besprochen wird.

In der linguistischen Annotierung mit XML spiegeln die Namen von Tagklammerungen meist linguistische Einheiten oder Konzepte wider, wohingegen leere Tags meist ‚Nicht-Wörter‘ repräsentieren, wie z.B. Interpunktion oder Meta-Informationen. Attribute können sowohl strukturelle Informationen, wie z.B. laufende Nummern, darstellen, als auch linguistische Konzepte. Um dies besser verstehen zu lernen, wandeln wir in der nächsten Übung ein längeres Textdokument in XML um, in dem verschiedene Arten von Überschriften, Absätze sowie Sätze annotiert und durchnummeriert sind.

Bevor wir jedoch diese Übung machen können, müssen wir noch lernen, wie man Funktionen beim Ersetzen mit Regexes verwendet und mit globalen Variablen arbeitet.

10.7 Ersetzen mit Funktionen und globale Variablen

Wie wir schon in Kapitel 6 angesprochen hatten, aber noch nicht ausführlich erläutert, erlaubt es das `re`-Modul auch, anstelle einer Regex im Ersetzungsteil der `sub`-Methode eine Funktion anzugeben. Dies kann man verwenden, um mit Suchergebnissen bestimmte Transformationen durchzuführen, aber auch, um eine komplexere Zeichenkette zu erzeugen, die das Ergebnis beinhaltet und dann zur Ersetzung verwendet wird. Wir machen uns dies in unserer nächsten Übung zunutze, indem wir damit die Satznummerierung über eine Zählervariable kontrollieren.

Bei der Ersetzung mit Funktionen definiert man eine eigene Funktion, deren Name anstelle eines Musters als Ersetzungsbegriff angegeben wird. Man verwendet also die Syntax

```
re.sub('Muster', Funktionsname, Zeichenkette)
```

Als Argument für die Funktion wird dabei immer implizit das aktuelle `match`-Objekt der jeweiligen Regex übergeben, so wie dies bei Objektmethoden immer mit der Referenz auf das Objekt geschieht. Auf dieses Objekt kann man dann in derselben Weise zugreifen, wie auf andere Suchergebnisse auch.

Wie wir ebenfalls schon gelernt hatten, sind Variablen in Funktionen immer lokal für die jeweilige Funktion definiert, so dass wir denselben Namen auch mehrmals im Programm vergeben können, ohne dass dadurch Konflikte auftreten. Wenn wir aber dennoch einmal auf eine Variable zugreifen wollen, die im Hauptprogramm definiert wurde, so können wir dies tun, indem wir sie zunächst innerhalb unserer Funktion `neu` mit dem Schlüsselwort `global`, gefolgt vom Variablennamen, anlegen. Alle weiteren Operationen auf die so deklarierte Variable innerhalb der Funktion greifen dann auf die globale Variable zu.

10.8 Text zu XML konvertieren

Bevor wir einen Text nach XML konvertieren können, müssen wir uns zunächst über seinen strukturellen und inhaltlichen Aufbau Gedanken machen. Danach können wir entscheiden, wie die einzelnen Teile sinnvoll ausgezeichnet werden können. Dazu muss man sich den Text erst einmal genau anschauen, um seine Eigenschaften zu verstehen. In unserer nächsten Übung konvertieren wir den Text von Hermann Hesses *Siddhartha*, in eine XML-Datei, um damit die Struktur besser zum Ausdruck bringen zu

können, und um es uns unter Umständen später zu erlauben, nur bestimmte Textteile zu durchsuchen etc.

Wenn wir uns den Text genau anschauen, dann sollte uns dabei auffallen, dass dieses Buch zunächst in Teile strukturiert ist, die eigene Überschriften aufweisen, dann Kapitelüberschriften, Absätze und, innerhalb der Absätze, wiederum Sätze. Somit ergibt sich für uns innerhalb des Behältertags für den Gesamttext, den wir logischerweise `text` nennen können, eine Struktur, die die Elemente `teil`, `kapitel`, `absatz`, und `satz` beinhalten sollte. Dabei enthält jeder Absatz ein oder mehrere Sätze. Auch wenn die Überschriften für Teile und Kapitel formell gesehen ebenfalls Absätze darstellen, bestehen diese naturgemäß nur aus einzelnen Einheiten, die normalerweise auch nicht durch Interpunktion abgeschlossen sind. Deshalb müssen wir sie nicht extra als Sätze annotieren, was wir aber theoretisch tun könnten.

Bei genauerer Analyse des Textes erkennen wir, dass die einzelnen Überschriften und Absätze im Text voneinander durch mindestens zwei Zeilenumbrüche getrennt sind, weshalb wir dies als Kriterium für die Aufspaltung in Absätze anwenden, die wir als Ausgangsbasis für die Annotierung benötigen. Innerhalb der (echten) Absätze jedoch enthalten die einzelnen Sätze auch unnötige Zeilenumbrüche, die die Verarbeitung erschweren, so dass wir sie entfernen müssen, um den Textfluss (wieder)herzustellen.

Bei der Analyse der Überschriften ist leicht erkennbar, dass diese durchgängig in Großbuchstaben geschrieben sind, wobei auch Leerzeichen oder Bindestriche auftreten. Dieses Merkmal können wir als Kriterium zur Unterscheidung zwischen Überschriften und ‚normalen‘ Absätzen verwenden. Zudem kommt in den Überschriften für die Teile auch immer das Wort *Teil* vor, was wir wiederum als Unterscheidungsmerkmal zwischen Teil- und Kapitelüberschriften anwenden. Mit diesem Wissen können wir jetzt unser Konvertierprogramm schreiben.



Übung 47 - Text zu XML konvertieren

Erstellen Sie ein weiteres Programm (`27_txt_nach_xml.py`), mit dem Sie den Text von Hesses *Siddhartha* in XML konvertieren. Dabei sollen alle Teil- und Kapitelüberschriften des Buches eigene Tags zugewiesen bekommen, sowie alle Absätze und Sätze darin ebenfalls passend ausgezeichnet werden. Außerdem sollen alle dieser Tag-Arten auch getrennt, aber durchlaufend, nummeriert werden.

Legen Sie erst alle benötigten Zählervariablen an und initialisieren Sie sie mit 0.

Öffnen Sie dann zunächst mit geeigneter Fehlerbehandlung die Datei `hesse_siddhartha.txt` zur Eingabe.

Spalten Sie diese anhand mehrerer aufeinanderfolgender Zeilenumbrüche auf und weisen Sie das Ergebnis einer Liste von Absätzen zu.

Öffnen Sie dann die Datei `hesse_siddhartha.xml`, natürlich wieder mit Fehlerbehandlung, zur Ausgabe.

Schreiben Sie zuerst als ‚Header‘ die XML-Deklaration und einen öffnenden `text`-Tag mit den Attributen `id`, `corpus` und `lang` (für engl. *language*, also Sprache) und passenden beschreibenden Werten in die Ausgabedatei.

Starten Sie daraufhin eine Schleife über alle Absätze.

Fügen Sie hier zunächst alle Zeilen im Absatz zu einer einzigen zusammen, damit wir später darin die Sätze markieren können. Bedenken Sie aber, dass nicht alle Zeilen immer auf ein Leerzeichen enden, da die Umbrüche im Text anscheinend immer eingefügt wurden, um eine Zeilenlänge von ca. 70 bis 71 Zeichen zu erreichen.

Prüfen Sie dann, ob das 1. Wort des Absatzes nur aus Großbuchstaben besteht. Falls ja, müssen Sie wiederum unterscheiden, ob eine Teil- oder eine Kapitelüberschrift vorliegt, und können einfach die Zeile, ‚umwickelt‘ mit passenden Tags, inklusive korrekter Zählernummer als Attribut, ausgeben und zum nächsten Schleifendurchlauf springen.

Falls nein, liegt ein einfacher Absatz vor, der ein oder mehrere Sätze beinhalten kann, die dann ebenfalls mit Tags versehen werden müssen.

Um die Sätze zu markieren, verwenden Sie `re.sub` mit einem passenden Ausdruck, der alle Zeichen (nicht-gierig) bis entweder zu einem Satzzeichen, gefolgt von einem Leerzeichen, oder einem Zeilenende findet und diese mithilfe einer Funktion mit einer `satz`-Tagklammer mit dem entsprechenden Zählerattribut ersetzt. Diese Funktion sollten Sie ebenfalls an einer geeigneten Stelle im Programm schreiben. Danach können Sie den Absatz, zusammen mit einem `absatz`-Tag mit Zählerattribut, ausgeben.

Zuletzt müssen Sie nur noch nach der Schleife den End-Tag des `text`-Behälters ausgeben, wobei am Ende diesmal kein Zeilenumbruch stehen sollte.

10.9 Lösungen zu den Aufgaben

Lösung 45 - Webseiten-Download

Im ersten Schritt des Programms müssen Sie das Modul `urllib.request` importieren. Danach sollte die Variable für die Adresse angelegt und mit der URL instanziiert werden, wobei sich für den Variablennamen `adresse` eignet. Da wir im darauffolgenden Teil mit Fehlerbehandlung arbeiten und gegebenenfalls das Programm beenden wollen, sollten Sie wieder das `sys`-Modul importieren.

Im `try`-Block müssen Sie, nachdem Sie idealerweise eine Meldung darüber ausgegeben haben, dass jetzt versucht wird, die Webseite herunterzuladen, mit der Methode `urllib.request.urlopen` und der Adresse als Argument, die Seite zu holen und in der dafür vorgesehenen Variable, passenderweise wahrscheinlich `seite`, abzuspeichern. Dieser Teil der Übung könnte also wie folgt aussehen.



```

adresse=\
'http://martinweisser.org/pract_cl/HTML_download_test_de.html'
try:
    print(f'Versuche {adresse} herunterzuladen...')
    seite=urllib.request.urlopen(adresse)
    print(f'Type des Objekts = {type(seite)}')
except urllib.error.URLError as fehler:
    sys.exit(f'Fehler beim Herunterladen.\n{adresse}'
            f'{fehler.reason}')
print(f'Seite {adresse} erfolgreich heruntergeladen')

```

Danach sollten Sie wieder eine Nachricht ausgeben, was als nächstes geschehen soll, z.B.

```
print('Suche nach Kodierung & Seitentitel...')
```

Die Extraktion der Kodierung sieht dann in etwa so aus:

```
kodierung=page.headers.get_content_charset(failobj="utf-8")
```

Jetzt können wir die tatsächlich gefundene oder angenommene Kodierung verwenden, um zu versuchen, den Seitentext auszulesen, im ungünstigsten Fall aber das Programm abubrechen.

```

try:
    seiteninhalt=page.read().decode(kodierung)
except Exception:
    sys.exit('Unable to decode URL. Aborting download.\n')
    return

```

Falls wir im letzten Schritt erfolgreich waren, können wir jetzt nach dem Titel suchen, was ungefähr so aussehen könnte;

```

titel=re.search(r'<title>\s*([<]+?)\s*</title>',
                seiteninhalt, re.M).group(1)

```

Im Normalfall sollte es eigentlich nicht nötig sein, auf führende oder folgende Leerzeichen, inklusive Zeilenumbrüchen, zu prüfen, so wie im obigen Code. Allerdings habe ich beim Austesten des Codes mit verschiedenen Seiten auch solche gefunden in denen der Titel-Tag eben solche Zeilenumbrüche beinhaltete und deshalb mehrzeilig war, weshalb sich auch hier die Verwendung des Flags `re.M` anbietet, da sonst u.U. nicht der gesamte Titel extrahiert wird. Daran zeigt sich leider einmal wieder, dass man sich bei Web-Inhalten nicht immer auf sinnvolle Annahmen verlassen kann. vor allem, da häufig die Inhalte nicht direkt von den Designern erstellt werden, sondern

über Programme aus Materialien anderer, die wenig Verständnis für die technischen Aspekte besitzen, ‚zusammengebastelt‘ werden.

Falls sowohl Kodierung als auch Titel gefunden wurde gefunden wurden, sollte gegebenenfalls eine geeignete Meldung ausgegeben werden.

```
if kodierung and titel:
    print('Kodierung:', kodierung, '& Titel:', titel)
```

Jetzt sollten wir kurz eine Information ausgeben, dass ein Dateiname erzeugt wird. Sollte ein Titel gefunden worden sein, dann können wir einfach ein oder mehrere nicht-Wort-Zeichen durch unterstriche ersetzen und die Extension anhängen, aber wenn nicht extrahieren wir den letzten Teil aus der URL wie folgt.

```
if titel:
    dateiName=re.sub(r'\W+',r'_',titel, re.M) + '.html'
else:
    dateiName=adresse[adresse.rindex('/')+1:]
```

Danach können wir wieder eine Meldung ausgeben, diesmal, dass wir versuchen, die Datei abzuspeichern, dann versuchen, dies zu tun, ansonsten das Programm wieder mit Fehlermeldung abbrechen, oder aber bei Erfolg dann eine dementsprechende Meldung ausgeben.

```
print('Versuche, Seiteninhalt von', adresse, 'abzuspeichern...')
try:
    with open(dateiName,mode='w',encoding='utf-8') as aus:
        aus.write(seiteninhalt)
except OSError as fehler:
    sys.exit(str(fehler))
print(adresse, 'erfolgreich als', dateiName, 'abgespeichert.')
```

Dieser letzte Teil dürfte im Prinzip sehr einfach gewesen sein. Insgesamt aber haben Sie hoffentlich bei dieser Übung gesehen, dass selbst relativ einfache Operationen gezielte und häufige Fehlerbehandlung erfordern.

Lösung 46 - Textextraktion aus Webseiten

Dieses Programm ist, da es nur die Textextraktion illustrieren soll, wieder um einiges kürzer als die, die wir zuletzt geschrieben hatten, nicht nur, da wir mit relativ wenig Objekten arbeiten, sondern auch, da ein großer Teil der Arbeit direkt vom `bs4`-Modul geleistet wird. Dies zeigt einmal wieder, dass man das Rad oft nicht neu erfinden muss, sondern es für Python schon viele sehr durchdachte und nützliche Module gibt, die man einfach in die eigenen Programme einbinden kann. Für fortgeschrittene Arbeiten mit HTML, bei der Sie noch mehr Kontrolle über die Webdaten haben, müssten Sie sich

jedoch trotzdem noch weiter in das Modul einarbeiten oder gegebenenfalls ein eigenes Modul schreiben.

Der Import des Objekts aus dem `bs4`-Modul war ja in Abschnitt 10.4 beschrieben, so dass wir hier nicht noch einmal darauf eingehen müssen. Ebenso sollte es mittlerweile nicht mehr nötig sein, das Öffnen der Webseite zu besprechen. Der erste wirklich neue Schritt ist die Erzeugung des Baums, welche Sie innerhalb des Datei-Kontexts, den Sie hoffentlich zum Öffnen der Datei mittels `with` etabliert haben, über

```
baum = BeautifulSoup(seite, "html.parser")
```

tun können, vorausgesetzt dass Ihre Dateivariablen `seite` heißt und das Baumobjekt der Variable `baum` zugewiesen wird.

Der etwas schwierigere Teil besteht darin, mithilfe der *list comprehension* den Text aus den Absätzen des Baum-Objekts zu extrahieren und in eine Liste zu schreiben. Dafür können wir aber einfach

```
absaetze = [absatz.getText() for absatz in baum.find_all('p')]
```

verwenden, wobei wir direkt über die Absatzliste, die `find_all` zurückliefert, iterieren und den Text aus allen Absätzen extrahieren. Als ‚Suchbegriff‘ müssen wir nur `'p'` angeben, da es sich um einen einfachen Tag handelt. Um aber z.B. alle möglichen Überschriften herausziehen zu können, arbeiten wir mit einem `re`-Objekt, weil Überschriften verschiedener Gliederungstiefen vorliegen könnten, so dass unser Suchbegriff dann als

```
re.compile(r'h\d')
```

definiert werden könnte, vorausgesetzt, dass das `re`-Modul ebenfalls importiert wurde.

Zur Ausgabe in einer einzelnen `print`-Anweisung können wir einfach die `join`-Methode mit ‚Separator‘ `'\n'` und der zuvor erzeugten Liste verwenden.

Lösung 47 - Text zu XML konvertieren

Da wir hier wieder mit Regexes arbeiten, sollten Sie am Anfang des Programms erst das `re`-Modul importieren, und zum Abbrechen des Programms bei etwaigen Ein- oder Ausgabefehlern ebenfalls das `sys`-Modul. Danach können Sie die Zählervariablen initialisieren, wobei wir jeweils eine für Teilüberschriften, Kapitelüberschriften, Absätze und Sätze benötigen, die wir `n_teilueberschrift`, `n_kapitel`, `n_absatz`, und `n_satz` nennen können, da normalerweise per Konvention für Zähler-Attribute in XML `n` verwendet wird.

Das Einlesen der Quelldatei sollte, mit passendem `try`-Block und `with` – am besten in einer Zeile – die Zuweisung an die Absatzliste erzeugen, und zwar, wenn die Referenz auf das Dateiojekt `ein` und die Absatzliste `absaetze` heißt, mittels

```
absaetze = re.split(r'\n{2,}', ein.read())
```

wo der komplette Dateiinhalt mittels `read()` eingelesen und gleich als Argument an die `split`-Methode übergeben wird. Bei dem Muster in der Regex ist es wichtig, dass mindestens zwei Zeilenumbrüche auftreten. Es können aber auch mehrere sein, weshalb wir die Quantifizierung mit geschweiften Klammern mit Minimum 2 und offenem Maximum verwenden. Die Angabe mit `+` funktioniert hier nicht, da sie auch einzelne Zeilenumbrüche finden würde und `\n\n` ebenso nicht, da dies uns leere Absätze zurückliefern würde, die wir zusätzlich durch eine Bedingungsabfrage herausfiltern müssten. Im `except`-Block müsste ein eventueller `OSError` beim Einlesen abgefangen werden und das Programm gegebenenfalls mit `sys.exit` und der Ausgabe des Fehlers abgebrochen.

Die Fehlerbehandlung für die Ausgabe sollte auf genau dieselbe Art und Weise erfolgen, weshalb wir sie nicht näher besprechen. Um die XML-Deklaration und den öffnenden `text`-Tag zu schreiben, können Sie entweder ein oder zwei `write`-Anweisungen verwenden. Als `id` bietet sich einfach der kleingeschriebene Buchtitel an, für das `Korpus`-Attribut der kleingeschriebene Name des Autors, da wir unter Umständen ein Korpus aller Werke Hesses anlegen wollen, wobei sich innerhalb der Dateien nur die IDs unterscheiden. Das `lang`-Attribut sollte am besten den internationalen Sprachcode für Deutsch (`de`) als Wert haben, und kann dazu dienen, etwaigen Programmen, die die Daten rein automatisch verarbeiten, einen ‚Hinweis‘ darauf zu geben, welche Sprachressourcen, wie z.B. Lexika, zur Verarbeitung benötigt werden.

Zum Zusammenfügen der Zeilen benötigen wir zwei Substitutionen. Die eine, um aufgrund der möglicherweise fehlenden Leerzeichen am Zeilenende die Zeilenumbrüche zunächst in Leerzeichen umzuwandeln, und die zweite, um im nächsten Schritt alle dadurch potenziell verdoppelten Leerzeichen wieder in einzelne zurückzuwandeln.

Um danach zwischen Überschriften und regulären Absätzen zu unterscheiden, muss man zunächst eine Bedingungsabfrage durchführen, in der getestet wird, ob am Zeilenanfang mindestens zwei Großbuchstaben stehen. Die Regex dafür sollte am besten

```
^[A-ZÄÖÜ]{2,}\b
```

sein, damit – zumindest theoretisch – auch die Umlaute erfasst werden. Liegt eine Überschrift vor, müssen wir weiterhin zwischen Teil- und Kapitelüberschriften unterscheiden, was aber sehr einfach ist, da in Teilüberschriften immer das Wort *Teil* in Großbuchstaben auftritt. Je nach Ergebnis muss dann auch entweder der Zähler für Teil- oder Kapitelüberschriften hochgesetzt und der jeweilige Absatz mit einem passenden Tag und Attribut in das Ausgabedokument geschrieben werden. Ist dies erfolgt, muss zum nächsten Schleifendurchlauf gesprungen werden, da der entsprechende Absatz ja schon fertig abgearbeitet wurde.

Liegt ein regulärer Absatz vor, wird die Sache etwas komplizierter, weil wir diesen auch in potenzielle Sätze unterteilen müssen. Zunächst aber erhöhen wir erst den Absatzzähler. Der Zähler für die einzelnen Sätze ist jedoch selbstverständlich abhängig davon, wie viele Sätze überhaupt vorkommen, so dass wir diesen nicht einfach

hochzählen und als Teil der Ersetzungen verwenden können, mit denen wir die Sätze markieren wollen. Um dies zu ermöglichen, benötigen wir unsere benutzerdefinierte Funktion, die innerhalb unserer `re.sub`-Anweisung die aktuelle Zählernummer für den Satz berechnet und auch jeweils den fertig getaggten Satz, der mittels der Regex gefunden wurde, als Ersetzung zurückliefert. Angenommen, dass unsere Funktion `satz_erzeugen` heißt, kann dann der Ersetzungsausdruck folgendermaßen aussehen:

```
absatz = re.sub(r'([\w, "':;-]+?[.?!]"?) (?:\s+|$)',
               satz_erzeugen,
               absatz)
```

Hier wird der Satzinhalt selbst durch die erste Klammerung gespeichert. Das ist für die Alternativen in der zweiten Gruppe wegen dem nach der öffnenden Klammer auftretenden `?:`, nicht der Fall, wie wir im Abschnitt 6.8 gelernt haben, so dass wir nur auf die erste Gruppe zugreifen, um den Satzinhalt später mit dem Tag zu umklammern. Die Funktion `satz_erzeugen`, die Sie weiter oben im Programm definieren sollten, da sonst ein Fehler auftritt, sollte in etwa so aussehen:

```
def satz_erzeugen(match_objekt):
    satz = match_objekt.group(1)
    global n_satz
    n_satz += 1
    return f'<satz n="{n_satz}">\n{satz}\n</satz>\n'
```

Wie Sie aus der Angabe des Arguments sehen, müssen wir hier mit dem implizit übergebenen Treffer-Objekt arbeiten, von dem wir die erste Gruppe extrahieren und in der Variable `satz` speichern. Danach geben wir an, dass wir innerhalb der Funktion auf die global definierte Variable `n_satz`, also den Satzzähler, zugreifen wollen, und inkrementieren diesen Zähler bei jedem Auffinden eines Satzes. Dadurch verwenden wir ihn als Wert für das `n`-Attribut unseres `satz`-Tags und können den getaggtem Satz direkt an die `sub`-Methode zur Ersetzung zurückliefern.

Zu guter Letzt müssen wir nur noch den End-Tag des `text`-Behälters in die Ausgabedatei schreiben, wobei diesmal kein Zeilenumbruch angefügt werden sollte, da dieser Tag ja das Ende der Datei darstellt.

Mithilfe der so erstellten XML-Version von *Siddhartha* und geeigneten Regexes können Sie jetzt ganz gezielt nach, oder in, bestimmten Teilen des Texts suchen, oder auch z.B. nach jedem ersten oder letzten Wort innerhalb von Sätzen, und diese zur weiteren Analyse extrahieren. Für komplexere Suchen auf verschachtelten Ebenen müssten Sie allerdings einen XML-Parser, wie den aus dem schon vorinstallierten `xml.etree`-Modul, verwenden, was wir jedoch hier nicht genauer besprechen.

Auch wenn unser letztes Programm schon eine sehr nützliche Ausgabe erzeugt, unterliegt es dennoch bestimmten Beschränkungen. Zum einen haben wir die Ein- und Ausgabedateien komplett festgelegt, weshalb wir das Programm nicht einfach mit anderen Dateien verwenden können. Zum anderen ist auch das Herausfiltern

von Sonderzeichen auf solche festgelegt, die wir für dieses eine Werk in dieser speziellen Ausgabe identifiziert hatten. Diese Dinge ließen sich jedoch relativ leicht durch flexiblere Programmooptionen erreichen, idealerweise auch dadurch, dass wir das Programm als Objekt umschreiben würden. Was wir weggelassen haben, um das Programm bewusst etwas einfacher zu gestalten, ist auch, dass unter Umständen nicht nur die Überschriften für Textteile und Kapitel in Tags gehüllt sein sollten, sondern auch die Teile und Kapitel selbst, die dann selbstverständlich auch wieder durchnummeriert werden sollten.

11 Schlusswort

In diesem Buch habe ich versucht, Ihnen die wichtigsten Grundlagen für Sprachanalysen mit Python beizubringen. Dabei haben wir nicht nur die grundlegenden Konzepte der Python-Programmierung wie Datentypen und Kontrollstrukturen sowie Interaktionsmechanismen besprochen, sondern auch, wie man auf gespeicherte Daten zugreifen kann, um diese auf effiziente Weise zu analysieren und zu quantifizieren. Wir haben vor allem gelernt, zu erkennen, wie komplex die Muster sind, die in natürlicher Sprache vorkommen, aber auch, wie wir sie mithilfe von Regexes oder anderer Konstrukte modellieren.

Zudem haben wir geübt, wie man seine Programme modular aufbaut, um wiederkehrende Aktionen nicht jedes Mal neu implementieren zu müssen und Programmcode effizienter zu gestalten und zu organisieren. Zu guter Letzt haben wir auch besprochen, wie man den Zugang zu Analysedaten und -optionen mithilfe von GUIs für uns selbst oder die Benutzer unserer Analyseprogramme vereinfacht, Daten aus dem Internet herunterlädt und Text daraus extrahiert sowie Textdaten durch Annotierung noch nützlicher macht.

Im Verlauf dieses Buches haben wir uns auch von den anfänglich relativ einfachen, wenn auch für Nicht-Programmierer noch recht ungewohnten Konzepten bis hin zu einem fortgeschrittenen Level durchgearbeitet. Dabei konnten wir nicht immer in die Tiefe gehen, Sie haben aber hoffentlich immer genug gelernt, um Ihre Programmierfähigkeiten jetzt selbstständig weiterzuentwickeln. Selbst, wenn Sie jetzt die wichtigsten Grundlagen der Programmierung beherrschen, heißt das allerdings nicht, dass sie deshalb schon richtig programmieren können. Das lernt man nur durch ausgiebige und langjährige Übung wirklich erst beherrschen, und selbst dann macht man wahrscheinlich immer noch gelegentlich Fehler.

Wie schon oben angesprochen, konnten wir bei unseren Übungen nicht immer in die Tiefe gehen. Allerdings habe ich Ihnen so gut wie möglich versucht, für den Umgang mit Sprache sinnvolle ‚Projekte‘ zu geben, die sich teilweise zu großen, direkt anwendbaren Programmen ausbauen lassen.

12 Appendix - Python-Programme

Bitte beachten Sie, dass ich längere Zeilen im unten stehenden Programmcode teilweise manuell, so wie in Sektion 2.7 beschrieben, umbrochen habe, um Sie im Buch darstellen zu können. Dadurch unterscheiden sie sich teils von den Programmen, die Sie unter <https://meta.narr.de/9783823384564/Zusatzmaterial.zip> direkt herunterladen können.

01_tausch.py

```
#!/usr/bin/env python3
# 01_tausch.py
# Autor: Ihr Name
# Programm zum Vertauschen zweier Variablen,
# um syntaktische Inversion zu simulieren
# erstellt: Erstellungsdatum
# zuletzt bearbeitet: Bearbeitungsdatum

# Woerter deklarieren & instanziiieren
wort1 = 'hier'
wort2 = 'ist'
print('Vor Vertauschung, Anfang Deklarativsatz: Wort 1='
      + wort1 + '; Wort 2=' + wort2)
# Inhalt von wort1 zwischenspeichern
temp = wort1
# Wert von wort2 wort1 zuweisen
wort1 = wort2 # wort1 jetzt ist
# zwischengespeicherten Wert wort2 zuweisen
wort2 = temp # wort2 ist jetzt hier
print('Nach Vertauschung, Anfang Interrogativsatz: Wort 1='
      + wort1 + '; Wort 2=' + wort2)
```

02_woerterliste.py

```
#!/usr/bin/env python3
# 02_woerterliste.py

woerter = ['dies', 'ist', 'ein', 'Satz']
print(woerter[0], woerter[1], woerter[2], woerter[3]+'.')
print(woerter[1], woerter[0], woerter[2], woerter[3]+'?')
```

03_get_args_argv.py

```
#!/usr/bin/env python3
# 03_get_args_argv.py
# sys-Modul importieren
import sys

# Programmnamen, der in sys.argv an 1. Position steht, ausgeben
print('Der Programmname ist ' + sys.argv[0])
# 1. Wort von Kommandozeile holen & in Variable wort1 speichern
wort1 = sys.argv[1]
# 2. Wort von Kommandozeile holen & in Variable wort2 speichern
wort2 = sys.argv[2]
# Wörter mit Erklärung ausgeben
print('Wort1: ' + wort1, '; Wort2: ' + wort2)
```

04_get_args_input.py

```
#!/usr/bin/env python3
# 04_get_args_input.py
# sys-Modul importieren
import sys

# Programmnamen, in sys.argv an 1. Position, ausgeben
#print('Der Programmname ist ' + sys.argv[0])
# 1. Wort von Kommandozeile holen; in Variable wort1 speichern
wort1 = input('Bitte 1. Wort eingeben...\n')
# 2. Wort von Kommandozeile holen; in Variable wort2 speichern
wort2 = input('Bitte 2. Wort eingeben...\n')
# Wörter mit Erklärung ausgeben
print('\n\nWort1: ' + wort1, '\nWort2: ' + wort2)
```

05_wortvergleich.py

```
#!/user/bin/env python3
# 05_wortvergleich.py
import sys

wort1 = sys.argv[1]
wort2 = sys.argv[2]

if wort1 == wort2:
    print("Die Wörter sind gleich.\nWort 1\t" + wort1)
```

```

        + '\nWort 2\t' + wort2)
elif wort1 < wort2:
    print("Wort 1 kommt vor Wort 2.\nWort 1\t" + wort1
        + '\nWort 2\t' + wort2)
else:
    print("Wort 2 kommt vor Wort 1.\nWort 1\t" + wort1
        + '\nWort 2\t' + wort2)

```

06_satz_eingeben.py

```

#!/usr/bin/env python3
# 06_satz_eingeben.py

wort = input(
    'Bitte geben Sie Wörter ein, um einen Satz zu bilden.\n'
    'Zum Abschluss des Satzes geben Sie einfach . oder ? '
    'ein.\n')
satz = wort
while wort != '.' and wort != '?':
    #print('Das aktuelle Wort ist', wort)
    wort = input('Weiteres Wort oder Satzzeichen?\n')
    if wort != '.' and wort != '?':
        satz += ' ' + wort
    #print('Der bisherige Satz ist:', satz)
print('Der Satz war: ' + satz + wort)

```

07_wort_verkleinerung.py

```

#!/usr/bin/env python3
# 07_wort_verkleinerung.py

satz = input('Bitte geben Sie einen vollständigen Satz ohne '
    'Satzzeichen ein.\n')
woerter = satz.split()
for wort in woerter:
    print(wort.lower())
print('Der ursprüngliche Satz war:\n' + satz)

```

08_bereinigung.py

```
#!/usr/bin/env python3
# 08_bereinigung.py

kette = '    Wort1  Wort2    '
print('Mit strip()-Methode ohne Argumente: >>')
    + kette.strip() + '<<')
print('Mit replace()-Methode: >>')
    + kette.replace(' ', ' ') + '<<')
print('Mit beiden Methoden: >>')
    + kette.replace(' ', ' ').strip() + '<<')
```

09_stamm_bildung.py

```
#!/usr/bin/env python3
# 09_stamm_bildung.py

import sys

praefixe = tuple(sys.argv[1].split(','))
verb_kette = 'abschütteln angeben anhören abholen abtreten '
            'beleidigen betreten entfernen entleeren übergeben '
            'überlegen wiederholen 'wiedergeben verlieren verteilen '
            'zerstören zuhören'
for verb in verb_kette.split():
    if verb.startswith(praefixe):
        for praefix in praefixe:
            if verb.startswith(praefix):
                #print(verb[-2:-1])
                if verb[-2:-1] == 'l'
                    or verb[-2:-1] == 'r':
                    print('Der Stamm von', verb, ' ist:',
                        verb[len(praefix):-1])
                else:
                    print('Der Stamm von', verb, ' ist:',
                        verb[len(praefix):-2])
```

10_infix_loeschen.py

```
#!/usr/bin/env python3
# 10_infix_loeschen.py

infix = 'zu'
infixLaenge = len(infix)
woerter = 'abzunehmen,anzuhören,auszutragen,herauszugeben, '
        'hineinzulegen,mitzunehmen,wegzunehmen,weiterzugeben'
for wort in woerter.split(','):
    indexPos = wort.index('zu')
print('Wort mit Infix:', wort + '; ohne Infix:',
      wort[0:indexPos] + wort[indexPos+infixLaenge:])
```

11_datei_lesen_a.py

```
#!/usr/bin/env python3
# 11_datei_lesen_a.py

import sys

# Dateinamen als 1. Argument von Kommandozeile speichern
dateiName = sys.argv[1]
# versuchen, Datei zum Lesen mit Kodierung UTF-8 zu öffnen
try:
    datei = open('./' + dateiName, 'r', encoding='utf-8')
# bei Fehler, Fehlermeldung & Programm explizit beenden
except OSError as fehler:
    sys.exit(str(fehler))

# alle Zeilen aus Dateiobjekt auslesen
Zeilen = datei.readlines()
# Datei manuell schließen
datei.close()
'''Zeilen in Schleife abarbeiten, aber dabei jeweils
über enumerate ein Tupel von Zeilennummer & Zeile erzeugen'''
for (num, zeile) in enumerate(Zeilen):
    # Nachricht, Nummer (zu Kette gewandelt) & Zeile ausgeben
    zeile = zeile.strip()
print(f'Zeile Nummer {num+1}: {zeile}')
```

12_datei_lesen_b.py

```
#!/usr/bin/env python3
# 12_datei_lesen_b.py

import sys

# Dateinamen als 1. Argument von Kommandozeile speichern
dateiName = sys.argv[1]
# versuchen, über with...as Datei zum Lesen
# mit Kodierung UTF-8 zu öffnen
try:
    with open('./' + dateiName, 'r', encoding='utf-8')
        as datei:
        # Dateiinhalt als Kette speichern
        dateiInhalt = datei.read()
# bei Fehler, Fehlermeldung & Programm explizit beenden
except OSError as fehler:
    sys.exit(str(fehler))
for (num,zeile) in enumerate(dateiInhalt.splitlines()):
    print(f'Zeile Nummer {num+1:>2d}: {zeile}')
```

13_datei_lesen_c.py

```
#!/usr/bin/env python3
# 13_datei_lesen_c.py

import sys

dateiName = sys.argv[1]
try:
    with open('./' + dateiName, 'r', encoding='utf-8')
        as datei:
        for (num, zeile) in enumerate(datei):
            print(f'Zeile Nummer {num+1:>2d}:'
                f'{zeile.strip()}')
except OSError as fehler:
    sys.exit(str(fehler))
```

14_datei_kopieren.py

```
#!/usr/bin/env python3
# 14_datei_kopieren.py

import sys

(eingabeDatei, ausgabeDatei) = sys.argv[1:]
try:
    with open(eingabeDatei, 'r', encoding='utf-8') as ein,
        open(ausgabeDatei, 'w', encoding='utf8') as aus:
        for zeile in ein:
            aus.write(zeile)
except OSError as fehler:
    sys.exit(str(fehler))
```

15_verzeichnislesen.py

```
#!/usr/bin/env python3
# 15_verzeichnislesen.py

import os

dateien = []
verzeichnisse = []

for element in os.scandir():
    if element.is_file():
        dateien.append(element)
        #print('Datei:', element.name)
    elif element.is_dir():
        verzeichnisse.append(element)
        #print('Verzeichnis:', element.name)

print('Verzeichnisse:', end='\n\n')
for element in verzeichnisse:
    print(element.name)

print('\nDateien:', end='\n\n')
for element in dateien:
    print(element.name)
```

16_verzeichnis_erstellen.py

```
#!/usr/bin/env python3
# 16_verzeichnis_erstellen.py

import sys
from pathlib import Path

eingabeDatei = sys.argv[1]

pfad = Path.cwd()
neuesVerz = pfad / 'backup'
ausgabeDatei = neuesVerz / eingabeDatei

if not neuesVerz.exists():
    Path.mkdir(neuesVerz)
    print('Neues Verzeichnis', str(neuesVerz), 'angelegt')
else:
    print(str(neuesVerz), 'existiert schon.')

try:
    with open('./' + eingabeDatei, 'r', encoding='utf-8')
        as ein, \
        open(str(ausgabeDatei), 'w', encoding='utf8') as aus:
        for zeile in ein:
            aus.write(zeile)
except OSError as fehler:
    sys.exit(fehler)
```

17_einfache_muster.py

```
#!/usr/bin/env python3
# 17_einfache_muster.py

import sys
import re

suchbegriff = sys.argv[1]

try:
    with open('./beispiel_saetze.txt', 'r', encoding='utf-8')\
        as datei:
        for zeile in datei:
            ergebnis = re.search(suchbegriff, zeile)
```



```

        if ergebnis:
            (anfang,ende) = ergebnis.span()
            print(f'{zeile[:anfang]}[{ergebnis.group()}] '
                  f'{zeile[ende:]}', end='')
except OSError as fehler:
    sys.exit(str(fehler))

```

18_zeichenklassen_testen.py

```

#!/usr/bin/env python3
# 18_zeichenklassen_testen.py

import sys
import re

# Suchbegriff von Kommandozeile holen & abspeichern
try:
    suchbegriff = sys.argv[1]
    # falls keine Sonderzeichen im Suchbegriff vorkommen,
    # die Klassen einleiten, mit passender Fehlermeldung
    # abbrechen
    if not re.search('[.\\[\\]\\]', suchbegriff):
        sys.exit('Keine Zeichenklasse im Suchbegriff'
                  'definiert!')
# falls kein Argument angegeben wurde
except IndexError:
    # Programm mit geeigneter Fehlermeldung abbrechen
    sys.exit('Keine Suchbegriff definiert!')

# Datei öffnen
try:
    with open('beispiel_saetze.txt', 'r', encoding='utf-8')
        as datei:
            # über Zeilen iterieren
            for zeile in datei:
                # falls Suchbegriff mindestens einmal
                # gefunden wird...
                if re.search(suchbegriff, zeile):
                    # Variable zur Zusammensetzung der neuen
                    # Zeile mit Markierungen erstellen
                    zeileNeu = ''
                    # Startvariable anlegen

```

```

start = 0
# über alle Treffer auf der Zeile iterieren
for treffer in re.finditer(suchbegriff,zeile):
    # Anfangs- & Endpositionen des Treffers
    # bestimmen
    (anfang,ende) = treffer.span()
    # alles ab beginn des aktuellen Anfangs
    # bis vor Trefferanfang, linke Klammer,
    # Treffer & rechte Klammer an neue Zeile
    # anhängen
    zeileNeu += f'{zeile[start:anfang]}'
        f'[{treffer.group()}]'
    # Startvariable für nächsten Teil der Zeile
    # auf Ende des Treffers setzen
    start = ende
# am Ende der Schleife neue Zeile & Rest
# der Zeile, ohne Zeilenumbruch, ausgeben
print(zeileNeu + zeile[ende:-1])
except OSError as fehler:
    sys.exit(str(fehler))

```

19_regexes_testen.py

```

#!/usr/bin/env python3
# 19_regexes_testen.py

import sys
import re

# Suchbegriff von Kommandozeile holen & abspeichern
try:
    suchbegriff = re.compile(sys.argv[1])
# falls kein Argument angegeben wurde
except IndexError:
    # Programm mit geeigneter Fehlermeldung abbrechen
    sys.exit('Keine Suchbegriff definiert!')
except re.error as e:
    sys.exit(f'Regexfehler="{e.msg}" in Muster:"{e.pattern}"'
        f' an Position {e.pos}')

# Datei öffnen
try:

```

```

with open('beispiel_saetze.txt', 'r', encoding='utf-8')
    as datei:
    # über Zeilen iterieren
    for zeile in datei:
        # falls Suchbegriff mindestens einmal
        # gefunden wird...
        if suchbegriff.search(zeile):
            # Variable zur Zusammensetzung der neuen
            # Zeile mit Markierungen erstellen
            zeileNeu = ''
            # Startvariable anlegen
            start = 0
            # über alle Treffer auf der Zeile iterieren
            for treffer in suchbegriff.finditer(zeile):
                # Anfangs- & Endpositionen des Treffers
                # bestimmen
                (anfang,ende) = treffer.span()
                # alles ab beginn des aktuellen Anfangs
                # bis vor Trefferanfang, linke Klammer,
                # Treffer & rechte Klammer an neue Zeile
                # anhängen
                zeileNeu += f'{zeile[start:anfang]}'
                    f'[{treffer.group()}]'
                # Startvariable für nächsten Teil der
                # Zeile auf Ende des Treffers setzen
                start = ende
            # am Ende der Schleife neue Zeile & Rest der
            # Zeile, ohne Zeilenumbruch, ausgeben
            print(zeileNeu + zeile[ende:-1])
except OSError as fehler:
    sys.exit(str(fehler))

```

uebersetzer.py

```

#!/usr/bin/env python3
# uebersetzer.py
import re
import sys

def woerterbuch_lesen(woerterbuch_datei):
    woerterbuch = {}

```

```

try:
    with open(woerterbuch_datei, 'r', encoding='utf-8')
        as wb:
        for zeile in wb:
            zeile = zeile.strip()
            if zeile:
                wort, uebersetzung = zeile.split(':')
                woerterbuch[word] = uebersetzung
except OSError as fehler:
    sys.exit(str(fehler))
return woerterbuch

def saetze_lesen(satz_datei):
    saetze = []
    try:
        with open(satz_datei, 'r', encoding='utf-8') as sd:
            for zeile in sd:
                zeile = zeile.strip()
                if zeile:
                    saetze.append(zeile)
    except OSError as fehler:
        sys.exit(str(fehler))
    return saetze

def uebersetzen(woerterbuch, satz):
    uebersetzungen = list()
    satz = satz[0:1].lower() + satz[1:-1]
    for wort in re.split(r'\s', satz):
        if wort in woerterbuch:
            uebersetzungen.append(woerterbuch[word])
        else:
            uebersetzungen.append('???')
    return ' '.join(uebersetzungen).capitalize() + '.'

def uebersetzung_ausgeben(satz, uebersetzung):
    print('\n'.join(['Original:', satz, 'Übersetzung:',
        uebersetzung]))

```

20_uebersetzung.py

```
#!/usr/bin/env python3
# 20_uebersetzung.py

from uebersetzer import (woerterbuch_lesen, saetze_lesen,
                          uebersetzen, uebersetzung_ausgeben)

wb = woerterbuch_lesen('./woerterbuch_de_en.txt')
saetze = saetze_lesen('./saetze_de.txt')

for satz in saetze:
    uebersetzung_ausgeben(satz, uebersetzen(wb, satz))
```

wort.py

```
#!/usr/bin/env python3
# wort.py
import re
import sys

class Verb:
    """Klasse zur Modellierung von Verben

    infinitiv -- Infinitivform: nicht optional
    person -- Person: Standardwert '1'
    form_grad -- Formalitätsgrad: Standardwert 'informell'
    numerus -- Numerus: Standardwert 'singular'
    tempus -- Tempus: Standardwert 'praesens'
    modus -- Modus: Standardwert 'indikativ'
    typ -- Typ: Standardwert 'r' für regulär
    """

    def __init__(self,
                  infinitiv=None,
                  person='1',
                  form_grad='informell',
                  numerus='singular',
                  tempus='praesens',
                  modus='indikativ',
                  typ='r'):
        """Konstruktor"""
```

```

    if not infinitiv:
        raise NameError('Keine Infinitivform angegeben!')
    self.infinitiv = infinitiv
    self.person = person
    self.form_grad = form_grad
    self.numerus = numerus
    self.tempus = tempus
    self.typ = typ
    if not re.search(r'[lr]',self.infinitiv[-2:-1]):
        self.stamm = self.infinitiv[:-2]
    else:
        # Liquid vor Infinitiv-Endung
        self.stamm = self.infinitiv[:-1]

def partizip2(self):
    if self.typ=='r':
        # finaler Frikativ, Langvokal, Liquid,
        # nicht-alveolarer Plosiv, oder Doppelkonsonant
        # am Stammende
        if re.search(r'(?c?h|(?ie|(?[aoäöü]h|r)n?)|[pbfgklmsz])$',
            self.stamm)\
            or re.search(r'[nms]{2}$', self.stamm):
            return 'ge' + self.stamm + 't'
        elif re.search(r'(?[dtn])$', self.stamm):
            # finaler alveolarer plosiv
            return 'ge' + self.stamm + 'et'
        else:
            return 'ge' + self.stamm + 'en'
    else:
        return 'Irreguläres Verb. Noch nicht implementiert.'

def partizipl(self):
    return self.infinitiv + 'd'

def praesens(self):
    if self.typ=='r':
        if self.numerus=='singular':
            if self.person=='1':
                return self.stamm + 'e'
            elif self.person=='2':
                if self.form_grad=='informell':

```

```

        if re.search(r'[mndt]',
                    self.infinitiv[-3:-2]) \
        and not re.search(r'[nm]{2}$', self.stamm):
            return self.stamm + 'est'
        else:
            if re.search(r's$', self.stamm):
                return self.stamm + 't'
            else:
                return self.stamm + 'st'
    else:
        if re.search(r'[rl]$', self.stamm):
            return self.stamm + 'n'
        else:
            return self.stamm + 'en'
    else:
        if re.search(r'[mndt]',
                    self.infinitiv[-3:-2]) \
        and not re.search(r'[nms]{2}$', self.stamm):
            return self.stamm + 'et'
        else:
            return self.stamm + 't'
if self.numerus=='plural':
    if self.person=='2' and \
    self.form_grad=='informell':
        if re.search(
            r'(?<:c?h|(?<:ie|(?<:[aoäöü]h|<:r)n))$'
            r'|[pbmfgk<:lnsz]$', self.stamm):
            return self.stamm + 't'
        else:
            return self.stamm + 'et'
    else:
        if re.search(r'[rl]$', self.stamm):
            return self.stamm + 'n'
        else:
            return self.stamm + 'en'
    else:
        return 'Irreguläres Verb. Noch nicht implementiert.'

if __name__ == '__main__':
    for inf in ['fassen', 'duschen', 'lächeln', 'füttern',
                'flüstern', 'reden', 'trennen', 'prüfen']:

```

```

verb = Verb(infinitiv=inf,
            person='1',
            form_grad='formell',
            numerus='plural')
print(f'Partizip 1: {verb.partizip1()}, '
      f'Partizip 2: {verb.partizip2()}')
if verb.person == '1':
    if verb.numerus == 'singular':
        pronomen = 'ich'
    else:
        pronomen = 'wir'
elif verb.person == '3':
    if verb.numerus == 'singular':
        pronomen = 'er/sie/es'
    else:
        pronomen = 'sie'
else:
    if verb.numerus == 'singular' and \
       verb.form_grad == 'informell':
        pronomen = 'Du'
    else:
        pronomen = 'Sie'
print(f'Stamm: {verb.stamm}; Präsens Indikativ für'
      f'{verb.person}. Person {verb.numerus.capitalize()}, '
      f'Formalitätsgrad {verb.form_grad}: {pronomen} '
      f'{verb.praesens()}')

```

21_wortliste.py

```

#!/usr/bin/env python3
# 21_wortliste.py

import re
import sys
import os.path

woerter = []
eingabe_datei = './kafka_verwandlung.txt'
pfad, dateiname = os.path.split(eingabe_datei)
ausgabe_datei = os.path.join(pfad, 'wortliste_' + dateiname)

```



```

try:
    with open(eingabe_datei, 'r', encoding='utf-8') as datei:
        for zeile in datei:
            zeile = re.sub(r'[»«.,;!?:-]', '', zeile)
            zeile = re.sub(r'\s{2,}', ' ', zeile)
            zeile = zeile.strip()
            if not zeile:
                continue
            woerter.extend(re.split(r'\s', zeile))
except OSError as fehler:
    sys.exit(str(fehler))

try:
    with open(ausgabe_datei, 'w', encoding='utf-8') as datei:
        for wort in sorted(set(woerter), key=str.lower):
            datei.write(wort + '\n')
except OSError as fehler:
    sys.exit(str(fehler))

```

22_frequenzliste.py

```

#!/usr/bin/env python3
# 22_frequenzliste.py

import re
import sys
import os.path

woerter = {}
eingabe_datei = './kafka_verwandlung.txt'
pfad, dateiname = os.path.split(eingabe_datei)
ausgabe_datei = os.path.join(pfad, 'frequenzliste_'
    + dateiname)
laengstes_wort = 0

try:
    with open(eingabe_datei, 'r', encoding='utf-8') as datei:
        for zeile in datei:
            zeile = re.sub(r'[»«.,;!?:-]', '', zeile)
            zeile = re.sub(r'\s{2,}', ' ', zeile)
            zeile = zeile.strip()

```

```

        if not zeile:
            continue
        for wort in re.split(r'\s', zeile):
            woerter[word] = woerter.setdefault(word, 0)
                + 1
            if len(word) > laengstes_wort:
                laengstes_wort = len(word)
except OSError as fehler:
    sys.exit(str(fehler))

try:
    with open(ausgabe_datei, 'w', encoding='utf-8') as datei:
        for word in sorted(woerter.keys(), key=str.lower):
            datei.write(f'{word}:{laengstes_wort}} '
                        f'\t{woerter[word]}\n')
except OSError as fehler:
    sys.exit(str(fehler))

```

frequenzen.py

```

#!/usr/bin/env python3
# frequenzen.py

import re
import os.path

class Frequenzliste:

    def __init__(self,
                 eingabe_datei=None,
                 ausgabe_datei=None,
                 sortierung='n-1'):
        self.woerter = {}
        self.sortiert = []
        if not eingabe_datei:
            raise NameError(
                'Keine Eingabedatei angegeben! Unmöglich, '
                'Frequenzliste anzulegen...')
        else:
            pfad, dateiname = os.path.split(eingabe_datei)
            self.eingabe_datei = eingabe_datei
            if not ausgabe_datei:

```

```

        self.ausgabe_datei = os.path.join(pfad,
                                           'frequenzliste_' + eingabe_datei)
    else:
        self.ausgabe_datei = ausgabe_datei
    self.sortierung = sortierung

def liste_erzeugen(self):
    self.woerter.clear()
    self.laengstes_wort = 0
    self.max_laenge_zahl = 0
    try:
        with open('./' + self.eingabe_datei, 'r',
                  encoding='utf-8') as datei:
            for zeile in datei:
                zeile = re.sub(r'[»«.,;!?:-]', '', zeile)
                zeile = re.sub(r'\s{2,}', ' ', zeile)
                zeile = zeile.strip()
                if not zeile:
                    continue
                for wort in re.split(r'\s', zeile):
                    if len(wort) > self.laengstes_wort:
                        self.laengstes_wort = len(wort)
                    self.woerter[word] = \
                        self.woerter.setdefault(word, 0)
                        + 1
                    if len(str(self.woerter[word]))
                       > self.max_laenge_zahl:
                        self.max_laenge_zahl = \
                            len(str(self.woerter[word]))
    except OSError as fehler:
        raise OSError(fehler)

def liste_sortieren(self):
    if self.sortierung=='a-z':
        self.sortiert = sorted(self.woerter.keys(),
                                key=str.casefold)
    elif self.sortierung=='z-a':
        self.sortiert = sorted(self.woerter.keys(),
                                key=str.casefold,
                                reverse=True)
    elif self.sortierung=='n-1':

```

```

        self.sortiert = sorted(self.woerter.keys(),
                                key=lambda wort:(-self.woerter[word],
                                                  wort.casefold()))
    elif self.sortierung=='w_laenge':
        self.sortiert = sorted(self.woerter.keys(),
                                key=lambda wort:(-len(wort),-self.woerter[word],
                                                  wort))
    elif self.sortierung=='rueck':
        self.sortiert = sorted(self.woerter.keys(),
                                key=lambda wort:(wort[::-1],
                                                  len(wort)))
    else:
        self.sortiert = sorted(self.woerter.keys(),
                                key=lambda wort:(self.woerter[word],
                                                  wort.casefold()))

def liste_ausgeben(self):
    try:
        with open(self.ausgabe_datei,'w',encoding='utf-8')
            as datei:
            for wort in self.sortiert:
                datei.write(
                    f'{wort:{self.laengstes_wort}}\t'
                    f'{self.woerter[word]}'
                    f':>{self.max_laenge_zahl}d\n')
    except OSError as fehler:
        raise OSError(fehler)

if __name__ == '__main__':
    try:
        f_liste = Frequenzliste(
            eingabe_datei='kafka_verwandlung.txt',
            ausgabe_datei='test_frequenzliste.txt',
            sortierung='n-1')
        f_liste.liste_erzeugen()
        f_liste.liste_sortieren()
        f_liste.liste_ausgeben()
    except OSError as f1:
        print('Falsche Ein- oder Ausgabedatei angegeben!',
              str(f1).split(' ')[-1])
    except NameError as f2:
        print(str(f2))

```

minimalgui.py

```
#!/usr/bin/env python3
# minimalgui.py

import sys
from PyQt5.QtWidgets import QDialog, QApplication, QLabel

if __name__ == '__main__':
    app = QApplication(sys.argv)
    fenster = QDialog()
    label = QLabel(fenster)
    label.setText(sys.argv[1])
    label.move(100,20)
    fenster.setGeometry(100,100,250,50)
    fenster.setWindowTitle('Einfacher Dialog')
    fenster.show()
    sys.exit(app.exec_())
```

GUI_syn_inversion.py

```
#!/usr/bin/env python3
# GUI_syn_inversion.py

import sys
import re

from PyQt5.QtWidgets import (QMainWindow, QApplication,
                             QHBoxLayout, QVBoxLayout, QLabel, QLineEdit,
                             QPushButton, QFrame, QMessageBox)
from PyQt5.QtGui import QFont

class Inverter(QMainWindow):
    """GUI zur Illustration syntaktischer Inversion"""

    def __init__(self):
        # Konstruktor der Elternklasse aufrufen
        super().__init__()
        self.setFont(QFont("Courier", 12))
        self.initUI()

    def initUI(self):
        # Behälter für Layouts anlegen
        behaelter = QFrame()
```

```

# Behälter als zentrales Steuerelement setzen
self.setCentralWidget(behaelter)
# Haupt- und Unterlayouts definieren
haupt_layout = QVBoxLayout()
dekl_layout = QHBoxLayout()
interr_layout = QHBoxLayout()
konv_layout = QHBoxLayout()
# Unter-Layouts zum Hauptlayout hinzufügen
haupt_layout.addLayout(dekl_layout)
haupt_layout.addLayout(interr_layout)
haupt_layout.addLayout(konv_layout)
# Behälter Hauptlayout zuweisen
behaelter.setLayout(haupt_layout)

#Steuerelemente definieren
self.dekl_eingabe = QLineEdit(
    'Dies ist ein Deklarativsatz.')
self.interr_ausgabe = QLineEdit()
konverter = QPushButton('Konvertieren')
# Signal mit Slot verknüpfen
konverter.clicked.connect(self.inversion)

# Steuerelemente in Layouts einfügen
dekl_layout.addWidget(QLabel('Deklarativsatz:\t'))
dekl_layout.addWidget(self.dekl_eingabe)
konv_layout.addWidget(konverter)
interr_layout.addWidget(QLabel('Interrogativsatz:'))
interr_layout.addWidget(self.interr_ausgabe)

def inversion(self):
    teile = re.search(r'^(\w+\b) (\b\w+\b) (.+)\.$',
        self.dekl_eingabe.text())
    if self.dekl_eingabe.text() and teile:
        self.interr_ausgabe.setText(
            f'{teile.group(2).capitalize()} '
            f'{teile.group(1).lower()}{teile.group(3)}?')
    else:
        QMessageBox(
            text='Bitte einen kompletten einfachen'
            'Deklarativsatz eingeben!!',
            windowTitle='Eingabefehler',
            icon=QMessageBox.Critical).exec_()

```

```

if __name__ == '__main__':
    # Instantiierung des Programms
    app = QApplication(sys.argv)
    # Anlegen des Fensterobjekts
    inverter = Inverter()
    inverter.setWindowTitle('Einfache syntaktische Inversion')
    inverter.setGeometry(15, 35, 600, 100)
    # Anzeigen des Fensters
    inverter.show()
    # Programmaufruf
    sys.exit(app.exec_())

```

23_seite_holen.py

```

#!/usr/bin/env python3
# 23_seite_holen.py

import urllib.request
import re
import sys

adresse=\
'http://martinweisser.org/pract_cl/HTML_download_test_de.html'
try:
    print(f'Versuche {adresse} herunterzuladen...')
    seite=urllib.request.urlopen(adresse)
    print(f'Type des Objekts = {type(seite)}')
except urllib.error.URLError as fehler:
    sys.exit(f'Fehler beim Herunterladen.\n{adresse}'
            f'{fehler.reason}')
print(f'Seite {adresse} erfolgreich heruntergeladen')

print('Suche nach Kodierung & Seitentitel...')
# Kodierung über Header-Info finden
kodierung=seite.headers.get_content_charset(failobj="utf-8")
try:
    seiteninhalt=seite.read().decode(kodierung)
except Exception:
    sys.exit('Unable to decode URL. Aborting download.\n')
    return

# nach Titel suchen
titel=re.search(r'<title>\s*([^\<]+?)\s*</title>',

```

```

        seiteninhalt, re.M).group(1)
if kodierung and titel:
    print('Kodierung:', kodierung, '& Titel:', titel)

# Dateinamen generieren
print('Erzeuge Dateinamen...')
# entweder aus Titel, falls gefunden oder sonst Adresse
if titel:
    dateiName=re.sub(r'\W+',r'_',titel, re.M) + '.html'
else:
    dateiName=adresse[adresse.rindex('/')+1:]
print('Versuche Seiteninhalt von', adresse,'abzuspeichern...')
try:
    with open(dateiName,mode='w',encoding='utf-8') as aus:
        aus.write(seiteninhalt)
except OSError as fehler:
    sys.exit(str(fehler))
print(adresse,'erfolgreich als',dateiName,'abgespeichert.')

```

24_absaetze_extrahieren.py

```

#!/usr/bin/env python3
# 24_absaetze_extrahieren.py

from bs4 import BeautifulSoup

with open("HTML_Download_Testseite_zur_Analyse.html",
        mode='r', encoding='UTF-8') as seite:
    baum = BeautifulSoup(seite,"html.parser")
absaetze = [absatz.getText() for absatz in baum.find_all('p')]
print('\n'.join(absaetze))

```

25_txt_nach_xml.py

```

#!/usr/bin/env python3
# 25_txt_nach_xml.py
import re
import sys

'''Dies ist ein kleines Beispielprogramm, was verdeutlichen soll, wie
man ein Buch aus einer Textdatei in XML umwandeln kann. Allerdings
ist es hinsichtlich der Konvertieroptionen extrem beschränkt, da das

```


Eingabeformat u.U. rein auf Hesses Siddhartha beschränkt ist, und deshalb nicht unbedingt generell eingesetzt werden kann. Auch sollten eigentlich die Textteile & Kapitel ebenfalls in Tagklammern eingeschlossen werden, worauf wir hier verzichtet haben, um das Programm nicht zu komplex zu machen.'''

```
n_teilueberschrift = 0
n_kapitel = 0
n_absatz = 0
n_satz = 0

def satz_erzeugen(match_objekt):
    satz = match_objekt.group(1)
    global n_satz
    n_satz += 1
    return f'<satz n="{n_satz}">\n{satz}\n</satz>\n'

# Eingabedatei einlesen
try:
    with open('./hesse_siddhartha.txt', mode='r',
              encoding='utf-8') as ein:
        absaetze = re.split(r'\n{2,}', ein.read())
except OSError as fehler:
    sys.exit(str(fehler))

try:
    with open('./hesse_siddhartha.xml', mode='w',
              encoding='utf-8') as aus:
        # 'header' schreiben
        aus.write('<?xml version="1.0"?>\n')
        aus.write('<text id="siddhartha" corpus="hesse"'
                  ' lang="de">\n')
        for absatz in absaetze:
            # Zeilen in Absatz zusammenfügen
            absatz = re.sub(r'\n', ' ', absatz)
            absatz = re.sub(r' +', ' ', absatz)
            if re.search(r'^[A-ZÄÖÜ]{2,}\b', absatz):
                if re.search(r'\bTEIL\b', absatz):
                    n_teilueberschrift += 1
                aus.write(
                    f'<teil n="{n_teilueberschrift}">'
                    f'\n{absatz}\n</teil>\n')
```

```

        else:
            n_kapitel += 1
            aus.write(
                f'<kapitel n="{n_kapitel}">'
                f'\n{absatz}\n</kapitel>\n')
            continue
    else:
        n_absatz += 1
        absatz=re.sub(
            r'([\w, "::-]+?[.?!]"?) (?:\s+|$)',
            satz_erzeugen, absatz)
        aus.write(
            f'<absatz n="{n_absatz}">\n{absatz}</absatz>\n')
        # 'footer' schreiben
        aus.write('</text>')

except OSError as fehler:
    sys.exit(str(fehler))

```

Register

/ 16, 30

. 18

.. 18

" 27

\ 27, 34

+ 30f.

+= 30

- 30

-= 30

* 30f.

*= 30

** 30

/= 30

// 30

% 30f., 68

== 30f.

!= 30f.

< 31, 172

<= 31

> 31, 172

>= 31

36

\n 46, 54, 59

\t 59

\r\n 60

\r 60

\n\r 60

\w 97

\s 97, 100

^ 97, 101

\b 101

\$ 101

\A 101

\Z 101

?: 102

?P<> 102

| 102, 105

?= 103

?! 103

?<= 104

?<! 104

__init__() 120

super().__init__() 155

< 172

> 172

& 172

= 172

Abkürzungen 97

Ableitung 151

actionTriggered 164

addAction() 162f.

addActions() 162f.

addLayout() 157f.

addMenu() 163

addToolBar() 163

addWidget() 157

Administratorrechte 21

Alternation 102

and 33

Anführungszeichen 27

Annotierungen 171, 178

Anweisungen 23

app.exec_() 155

append() 44f.

Argumente 23

Argument-Werte-Paare 23

Schlüsselwort-Argumente 23

as 114

Attribute 118

Auszeichnungssprache 171

backslash 27

Baum 176

Beautiful Soup 175f.

Benutzerdefinierte Funktionen 113

- Bits 27
- Block 48
- bool 26, 29
- bs4 → Beautiful Soup
- Bytes 27
-
- capitalize() 67
- casefold() 67
- case-sensitive 28
- cd 17
- chmod 34
- chr() 28
- class 118
- clear() 45, 112
- clicked 161
- close() 76
- cmd 14
- connect() 160
- count 94
- count() 62
- cwd() 84
-
- Dateimodi
 - a 76
 - r 76
 - t 76
 - w 76
- Dateisysteme 16
- debugging 19
- def 113
- dict 26, 43, 111
- dict() 111
- Dictionaries 111
- dir 16
- divide-and-conquer 47
- Docstrings 36
- DOTALL → re.S
-
- Eingabeaufforderung 14
- Elemente 172
- elif 48
- else 48
-
- Elternteil 154
- encoding 77
- end 84
- end() 95
- endswith() 59
- Entities 172
- enumerate() 80
- Ereignisschleife 149
- Ereignisse 149, 160
- except 78
- exec_() 161
- exists() 84
- exit 16
- extend() 44f.
- eXtensible Markup Language → XML
-
- False 29, 32
- file 75
- finally 78
- find_all() 176
- find() 59
- findall() 93
- finditer() 94
- float 26, 29
- for 51
- format() 68
- for-Schleifen 51
- from 114
- f-strings 70
-
- get_text() 176
- getExistingDirectory() 165
- getOpenFileName 164
- getOpenFileNames 164
- getSaveFileName 164
- gierig 99
- global 179
- Grafische Benutzeroberflächen 149
- greedy → gierig
- group() 95
- groupdict() 102
- groups() 102

Gruppen 102
 GUIs → Grafische Benutzeroberflächen

Hauptfensterklasse 151
 home() 84
 HTML 171
 html.parser 176
 HyperText Markup Language → HTML

IDE 19
 if 48
 IGNORECASE → re.I
 import-Anweisung 45
 in 63, 112
 IndentationError 79
 index() 65
 IndexError 79
 Indexpositionen 43
 Initialisierungsmethode 119
 initUI 151
 input() 46
 int 26, 29
 int() 29
 is_dir() 83f.
 is_file() 83f.
 isdigit() 59
 islower() 59
 isupper() 59
 items() 112
 iterieren 49

join() 66

key 134
 keys() 112
 Klammerung 102
 Klassen 118
 Elternklasse 151
 Klassenschema 120
 Kodierung
 legacy encodings 28
 Kommandozeile 14

Kompilierungsflags 104
 Konstruktor 119

lambda 137
 Lambda-Ausdrücke → Lambda-Funktionen
 Lambda-Funktionen 137

Laufwerke
 : 17
 Laufwerksbuchstaben 16
 mount points 16
 Partition 16

len() 45, 62
 limit 176
 list 26, 43
 list() 44, 59
 list comprehension 49, 177
 listdir() 82
 Listen 44
 lower() 51, 59, 67
 ls 17
 lstrip() 61

markup language → Auszeichnungssprache
 maskieren 99
 Maskierung 99f.
 match 179
 match() 93
 Meta-Informationen 172
 Methoden 25
 mkdir 17
 mkdir() 84
 mode 75
 Modularisierung 111f.
 Module 113
 mousePressEvent 160
 move() 154
 msg 100
 MULTILINE → re.M

name 83
 NameError 79
 NavigableString 176

- N-Grammlisten 133
- None 26
- not 33, 63
- ntegrated Development Environment 19
- Objekte 118
- open() 76
- or 33
- ord() 28
- os 82
- os.path.join() 83
- os.path.split() 83
- os.sep 82
- OSError 78f.
- Pakete 114
- pass 49
- path 83
- Path 84
- pathlib 84
- pattern 100
- Pfade 17, 82, 84
- Pfaden 82
- pip 117
- pip3 117
- pos 100
- print() 23
- Prompt → Eingabeaufforderung
- prompt-Argument 46
- PyQt5.QtCore 150
- PyQt5.QtGui 150
- PyQt5.QtWidgets 150
- Python package installer 114
- Python-Shell 24
- QAction 162
- QActions 151
- QApplication 153
- QCheckBox 152
- QComboBox 152
- QDialog 153
- QFileDialog 152, 164
- QFont() 156
- QFrame 152
- QGridLayout 157
- QHBoxLayout 157
- QLabel 152
- QLineEdit 152, 159
- QListWidget 152, 165
- QMainWindow 155
- QMenuBar 152
- QMessageBox 156, 161
- QPushButton 152, 159
- QRadioButton 152
- QSpinBox 152
- QStatusBar 152
- QTextEdit 152
- QToolBar 152
- Quantifikatorsymbole 98
 - * 98
 - ? 99
 - + 99
- Quantifizierung 98
- Quellcode 34
- QVBoxLayout 157
- QWidget 152
- r 27
- raise 138
- re 93
- re.error 100
- re.I 104
- re.M 105
- re.S 105
- re.X 105
- read() 77
- readline() 77
- readlines() 78
- Reguläre Ausdrücke 93
 - Regex 93
 - regular expression 93
- replace() 61
- return 113
- reverse 134

- rfind() 59
- rglob() 85
- roh 27
- round() 29
- rstrip() 61
- Rückgabewert 31
- scandir() 83
- Schlüssel 111
- Schrittweite 63
- search() 93
- Segmentierung 134
- Seitenkopf 172
- Seitentext 172
- self 119
- Separator 102
- Sequenzen 43, 62
- set 26, 43
- set() 134
- setCentralWidget() 158
- setDefault() 112, 136
- setFont() 156
- setNativeMenuBar() 163
- setText() 154
- setWindowTitle() 154
- Shebang-Zeile 33
- shorthands → Abkürzungen
- show() 154
- ShowDirsOnly 165
- showMessage() 164
- Signal 160
- Signale 150
- Slices 59
- Slicing 61
 - Slices 63
- slots 150
- sort() 44, 134
- sorted() 134
- Sortierschlüssel 134
- span() 95
- split() 51, 59, 94
- splitlines() 80
- start() 95
- startswith() 59
- statusBar() 164
- Steuerelemente 149f.
- str 26
- str.lower 134
- str.upper 134
- str() 29
- string 176
- strip() 61
- Stylesheets 173
- sub 179
- sub() 94
- swapcase() 67
- SyntaxError 79
- sys.argv 45
- sys.exit() 79
- sys Modul 45
- Tabulatorzeichen 59
- Tags 172
 - End-Tag 172
 - leere Elemente 172
 - Start-Tag 172
 - Tagklammern 172
- title() 67
- Tokenisierung 134
- Tokens 133
- Treffer-Objekt 95
- Trenner 44
- triggered 162
- True 29, 32
- try 78
- Tupel 64
- tuple 26, 43
- tuple() 64
- Type 133
- TypeError 126
- Umherschau 103
 - Rückwärtsschau 104
 - Vorausschau 103

- Unicode 28
- upper() 59, 67
- urllib 171
- utf-8 77

- values() 112
- Variablen 24
 - camel case 25
 - Deklaration 24
 - global 113
 - Initialisierung 24
 - Instanzvariablen 118
 - Klassenvariablen 118
 - lokal 113
 - Unterstrichform 25
- VERBOSE → re.X

- while 50
- while-Schleifen 50
- Widgets → Steuerelemente
- with ... as 76
- Wortgrenze 101
- Wortlisten 133
- write() 81
- writelines() 81
- Wurzel 16

- XML 171
 - Validität 178
 - Wohlgeformtheit 178
- xml.etree 186
- XML-Deklaration 178

- Zeichenketten 59
 - roh 59
 - Zeichenkette 23
- Zeichenklassen 96
- Zeichensätze 27
 - ASCII 27
 - ISO 8859-1 27
 - Latin1 27
 - UTF-16 28
 - UTF-32 28
 - UTF-8 28
- Zeilenanfang 101
- Zeilenende 101
- Zeilenumbruch 59
- zentrales Steuerelement 150
- ZeroDivisionError 79
- zip() 77
- Zirkumflex 97
- Zuweisung 24

Abbildungsverzeichnis

Abb. 1:	Python Installation unter Windows	13
Abb. 2:	Eingabeaufforderung als Administrator	14
Abb. 3:	Terminal unter MacOS	15
Abb. 4:	Beispiel für die <code>format</code> -Methode	69
Abb. 5:	Layout für GUI-Inversion	166

Tabellenverzeichnis

Tabelle 1:	Für linguistische Zwecke wichtigste Datentypen	26
Tabelle 2:	Nützliche Zeichenkettenmethoden	26
Tabelle 3:	Latin1-Kodepositionen für Zeichen	27
Tabelle 4:	Wichtige Funktionen für Zahlen	29
Tabelle 5:	Mathematische Operatoren	30
Tabelle 6:	Zeichenkettenoperatoren	31
Tabelle 7:	Logische Operatoren	33
Tabelle 8:	Typen zusammengesetzter Datentypen	43
Tabelle 9:	Nützliche Listenmethoden	44
Tabelle 10:	Nützliche Zeichenkettenmethoden	59
Tabelle 11:	Wichtigste Fehlertypen	79
Tabelle 12:	Wichtigste Methoden der <code>re</code> -Objekts	93
Tabelle 13:	Methoden des Trefferobjekts	95
Tabelle 14:	Nützliche Attribute des <code>re.error</code> -Objekts	100
Tabelle 15:	Wichtigste Dictionary-Methoden	112
Tabelle 16:	PyQt5-Submodule	150
Tabelle 17:	Auswahl nützlicher Steuerelemente	152

Dieses Buch stellt die erste deutschsprachige Einführung in die Python-Programmierung für Germanist:innen sowie sprachorientierte Studierende oder Forschende in den Digital Humanities dar. Alle Beispiele sind konsequent der deutschen Sprache entnommen und verdeutlichen, wie diese auf verschiedene sprachliche Phänomene hin in geeigneter Weise quantitativ und qualitativ untersucht oder modelliert werden kann. Die behandelten Programmierkonzepte umfassen Grundbegriffe der Programmierung wie Datentypen und Kontrollstrukturen, die für Sprache essenzielle Handhabung von Zeichenketten und Mustererkennung, Modularisierung und Objektorientierung, die Erstellung von Frequenzlisten und grafischer Benutzeroberflächen sowie den Umgang mit Web-Daten und linguistischen Annotationen. Der Band setzt keinerlei Vorkenntnisse im Programmieren voraus und führt auch Anfänger:innen Schritt für Schritt fachgerecht in Python ein. Zahlreiche Übungen sowie Hinweise auf Fallstricke helfen beim Einstieg in die erfolgreiche Arbeit mit Python.

narr STUDIENBÜCHER

ISBN 978-3-8233-8456-4



narr
francke
attempto