Kantonsschule im Lee Winterthur

Thomas Graf

thomas.graf@ksimlee.ch

Ergänzungsfach / Wahlfach Informatik

29. Oktober 2018

Kantonsschule Im Lee

Inhaltsverzeichnis

Kapitel 0

Vorwort

0.1 Feedback

Dieses Skript wird für das Ergänzungsfach / Wahlfach Informatik an der Kantonsschule im Lee verfasst. Die Unterrichtsmaterialien befinden sich noch im Entwicklungsstadium. Es ist uns ein grosses Anliegen, sie laufend zu verbessern und auszubauen.
Für Hinweise auf Fehler oder Ungenauigkeiten jeder Art sind wir sehr dankbar. Ebenfalls ausgesprochen willkommen sind Verbesserungsvorschläge, Ergänzungen, Kritik und Lob.

Allgemeines Feedback bitte an thomas.graf@ksimlee.ch mailen.
Alternativ, kann man einen anonymen Kommentar auf
`https://sayat.me/ThomasGraf` posten.

0.2 Unterrichtsmaterialien

Wir werden versuchen die aktuellen Unterrichtsmaterialien jeweils vor dem Unterricht zur Verfügung zu stellen.

Die aktuellen Unterrichtsmaterialien lassen sich sowohl auf dem *Moodle* der Kantonsschule Im Lee
https://klwmoodle.tam.ch/course/view.php?id=358
oder (alternativ) auf unserem *GitHub* Repository
https://github.com/ThomasGrafCode/EF_WF_Informatik finden.

Bitte stellt sicher, dass Ihr stets die aktuellsten Unterlagen verwendet.

Auf unserem YouTube-Kanal
www.youtube.com/channel/UCxIIMOe4oVImeJQbsphVjug/playlists
lassen sich zu vielen der Themen in diesem Kurs (hoffentlich) hilfreiche Video-Erklärungen finden.

0.3 Voraussetzungen und Niveau

Der Inhalt dieses Unterrichts setzt so gut wie keine Vorkenntnisse in der Informatik voraus. Es wird von erheblichen Unterschieden bezüglich der Vorkenntnisse unter den Schülern ausgegangen. Der Inhalt ist so gestaltet, dass alle Schüler dem Unterricht folgen können. Gleichzeit sollen auch Schüler mit mehr Erfahrung gefordert werden und viel Neues dazu lernen.
Das Niveau dieses Fachs ist hoch ($\rightarrow$ Maturitätsniveau):
Aktive Mitarbeit im Unterricht sowie zusätzliches Arbeiten zu Hause werden als selbstverständlich angesehen.
Wer bereit ist, sich emotional und zeitlich intensiv mit der Materie auseinanderzusetzen, wird deutlichen Fortschritt erzielen und auch mit entsprechendem Prüfungserfolg belohnt werden.

Kapitel 1

Einleitung

1.1 Was ist Informatik?

Es ist schwierig, eine genaue und abschliessende Definition einer wissenschaftlichen Disziplin zu geben.
In der Literatur lassen sich Versuche einer groben Definition der Informatik finden:

> Informatik ist die „Wissenschaft von der systematischen Darstellung, Speicherung, Verarbeitung und Übertragung von Informationen, besonders der automatischen Verarbeitung mithilfe von Digitalrechnern“. [1]

Wer jedoch eine konkretere und genauere Vorstellung von der Informatik gewinnen möchte, muss bereits sein, ihre historische Entwicklung zu studieren und sich intensiv mit den Inhalten der Informatik auseinanderzusetzen.
Historisch hat sich die Informatik aus der Mathematik und den Ingenieurswissenschaften entwickelt, um den praktischen Bedarf nach der schnellen und automatischen Ausführung mathematischer Operationen (Berechnungen) zu decken.
‘Informatik’ ist ein Sammelbegriff, der sämtliche Aspekte, welche in irgend einer Form mit *Information* zu tun haben, umfasst.
Das Spektrum möglicher Aufgaben der Informatik reicht von simplen Anwendungsproblemen im Büroalltag bis hin zur Beantwortung einiger der schwierigsten Fragen der letzten Jahrhunderte [1].
So hat z.B. das Gebiet der ‘theoretischen Informatik’, welches sich mit Themen wie der Berechenbarkeit von Funktionen, Komplexität, Theorie der Pro-

[1] Probleme wie z.B. das *P vs NP Problem*: http://www.claymath.org/millennium-problems/p-vs-np-problem

grammiersprachen, Algorithmik und Kryptographie beschäftigt, fast ausschliesslich mathematische / philosophische Facetten. Der Begriff 'Informatik' umfasst aber auch Gebiete wie das der praktischen Programmierung von Computern, der digitalen Kommunikation, des Internets, der Textverarbeitung, der digitalen Medien und noch viele mehr.
Methoden der Informatik kommen in praktisch jeder modernen Wissenschaft zum Einsatz. Dadurch verwischen die Grenzen zwischen der Informatik und anderen Disziplinen mehr und mehr. Diese Tatsache erhöht die Schwierigkeit, die Informatik aussagekräftig und in wenigen Sätzen beschreiben zu wollen, natürlich noch zusätzlich.

1.2 Inhalt und Ziel des Unterrichts

Wie in Abschnitt 1.1 bereits gesagt wurde, ist das Gebiet der Informatik äusserst umfangreich. Es ist nicht möglich, sich mit allen Bereichen ausführlich zu beschäftigen. Dies gilt natürlich auch für jede andere Wissenschaft. Deshalb werden wir eine Auswahl von Themen treffen, die klein genug ist, um jedes Thema in sinnvoller Tiefe behandeln zu können, aber immer noch gross genug ist, dass sie ermöglicht das Wesen und den Stellenwert der Informatik zu erkennen.
Viele Aspekte der Informatik entwickeln sich rasend schnell. Aus diesem Grund kann spezifisches Wissen über, zum Beispiel bestimmte Software-Produkte, oder detaillierte Kenntnisse einer bestimmten Programmiersprache, schon nach wenigen Jahren komplett veraltet sein. Deshalb ist es viel wichtiger, die grundlegenden Konzepte, welche hinter konkreten Implementierungen stehen, zu verstehen. Beispielsweise sind viele Aussagen der Informatik mathematisch beweisbar und werden für alle Zeiten gültig sein.
Unter Berücksichtigung der soeben erwähnten beiden Punkte, haben wir eine Auswahl von interessanten Themen getroffen, welche verschiedene Gebiete der Informatik repräsentiert und unser analytisches Denken fördern wird. Ziel ist es, solide Grundlagen in verschiedenen Disziplinen der Informatik aufzubauen, welche auch im späteren Leben und vor allem im Studium sehr nützlich sein werden. Dazu gehören auch gewisse "handwerkliche" Fähigkeiten, wie zum Beispiel die Textverarbeitung in LaTeXund Grundkenntnisse über Computerarchitekturen und Programmiersprachen.
Der Fokus ist dabei immer stark auf das Verständnis der zugrundeliegenden Konzepte gerichtet.

Kapitel 2

Einführung in LaTeX

2.1 Vorbemerkungen

In diesem Kapitel wollen wir uns mit dem Computerprogramm 'LaTeX' beschäftigen. LaTeX ist ein ausgesprochen populäres Werkzeug zur Erstellung schriftlicher Arbeiten wie z.B. Maturitätsarbeiten, Bachelor- und Masterarbeiten, Büchern, wissenschaftlicher Paper, Schulunterlagen und professionell anmutender Beamer-Präsentationen. Auch dieses Skript wird in LaTeX verfasst. Darüber hinaus ist LaTeX hervorragend geeignet zur Erzeugung von Grafiken sowie zur Gestaltung imposanter Plakate und ästhetischer Bewerbungsunterlagen.

In vielen Studiengängen wird erwartet und verlangt, dass die Studierenden fähig sind, Dokumente in LaTeX zu verfassen. Die dazu notwendigen Kenntnisse müssen typischerweise selbstständig erarbeitet werden.
Um sich später im Studium besser auf die eigentlichen Lehrinhalte fokussieren zu können, ist es nützlich, den grundlegenden Umgang mit Werkzeugen, wie LaTeX, bereits in der Kantonsschule zu üben.

LaTeX unterscheidet sich grundlegend von Textverarbeitungsprogrammen, wie z.B. Microsoft Word (siehe Unterabschnitt 2.2.4). Daher kann das Arbeiten mit LaTeX zu Beginn recht gewöhnungsbedürftig sein. Dennoch ist es nicht schwierig, den Einstieg in das Arbeiten mit dieser Software zu finden.

Zu diesem Thema lassen sich Video-Erklärungen auf unserem YouTube-Kanal
www.youtube.com/channel/UCxIIMOe4oVImeJQbsphVjug/playlists
finden.

2.2 Installation

Im Folgenden werden wir verschiedene Tools vorstellen, die es ermöglichen mit LaTeX zu arbeiten. Grundsätzlich kann man entweder mit einem online LaTeX-Tool oder einer lokalen Installation arbeiten.
Wir ziehen online Tools den lokalen Installationen ganz klar vor.

2.2.1 Online LaTeX-Dienste

Im Internet werden verschiedene Webdienste angeboten, die es erlauben, mit LaTeX online im Browser zu arbeiten. Es wird keine lokale Installation auf dem Computer benötigt, und man kann deshalb an jedem Computer mit Internetzugang sofort zu arbeiten beginnen. Beispiele solcher online Dienste sind Papeeria, Overleaf, ShareLaTeX, Datazar, und LaTeX base.
Wir können den online LaTeX-Dienst ***Overleaf*** sehr empfehlen. Der Service ist kostenlos und man muss sich lediglich mit einer E-Mail-Adresse auf der Overleaf-Webseite:

`https://www.overleaf.com/`

registrieren. Wir benutzen Overleaf selber (z.B. um dieses Dokument zu schreiben), weil wir dieses Tool enorm praktisch finden.
Auf der Webseite
`https://www.latex-project.org/get/`
sind in der Sparte *Online* nebst dem Link zur Overleaf-Webseite auch die Links zu allen anderen oben genannten online Tools angegeben.

Um die LaTeX-Beispiele in diesem Skript und in den Übungen fehlerfrei ausführen zu können, werden wir diverse LaTeX-Zusatzpakete (siehe 2.4) benötigen. **Da dieses gesamte Skript und alle Übungen in Overleaf werden, können wir garantieren, dass sämtliche in diesem Kurs benötigten Zusatzpakete in Overleaf zur Verfügung stehen**.

Ihr seid alle dazu eingeladen (anstelle von Overleaf), ein anderes online Tool Eurer Wahl zu benutzen oder eine lokale Installation (siehe 2.2.2) zu verwenden.
Dabei gilt es allerdings zu beachten, dass Ihr eventuell fehlende Zusatzpakete manuell installieren müsst. Je nach Wahl Eures Setups kann dies sehr einfach oder recht schwierig sein. Bei mehreren dutzend, in das Ergänzungsfach / Wahlfach Informatik eingeschriebenen Personen, würde es einen enormen Aufwand bedeuten, allen bei der individuellen Problembehebung zu helfen.
Daher empfehlen wir nur erfahrenen Usern sich für einen anderen Setup als Overleaf zu entscheiden.

2.2.2 Lokale Installation

Natürlich kann man LaTeX auch lokal auf dem eigenen Rechner installieren. Um effizient arbeiten zu können, müssen dazu zwei Sachen installiert werden:

- eine TeX-Distribution
- ein LaTeX-Editor.

TeX-Distribution

TeX-Distributionen können z.B. von dieser Webseite heruntergeladen werden (für GNU/Linux, Microsoft Windows und macOS):
https://www.latex-project.org/get/.
Dort lassen sich auch ausführliche Installationsanleitungen finden, mit deren Hilfe die Installation reibungslos ablaufen dürfte.

LaTeX-Editor

Die LaTeX-Editoren **TEX**MAKER (http://www.xm1math.net/texmaker/) und TeXstudio (https://www.texstudio.org/) sind sehr gut.
Eventuell müssen zur fehlerfreien Ausführung der LaTeX-Beispiele in diesem Skript und den Übungen noch Zusatzpakete installiert werden.

2.2.3 Aussprache des Wortes LaTeX

Der Buchstabe 'X' in dem Wört LaTeX steht nicht für den lateinischen Buchstaben 'X' so wie beispielsweise in dem Wort '**X**ylophon', sondern für den grossgeschriebenen griechischen Buchstaben 'Chi' (χ), welcher wie das harte

'ch' im deutschen Wort 'ach' ausgesprochen wird. Damit lautet die Aussprache nicht 'Latex' sondern 'Latech'.

2.2.4 WYSIWYG vs. WYSIWYM

Textverarbeitungsprogramme wie z.B. Microsoft Word oder LibreOffice Writer bezeichnet man als WYSIWYG "**W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **G**et" - man sieht den formatierten Text und die endgültige Form des Dokuments stets auf dem Bildschirm.
Im Gegensatz dazu stehen WYSIWYM "**W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **M**ean" Textverarbeitungsprogramme wie z.B. LaTeX. In LaTeX wird mit Computerbefehlen gearbeitet, welche die Struktur und Optik des zu erstellenden Dokumentes beschreiben. Die im LaTeX-File gespeicherten Befehle (Commands) müssen zuerst kompiliert werden [1,2], um das finale Produkt (typischerweise ein PDF-Dokument) zu erhalten (siehe 2.4).
Das endgültige Resultat eines LaTeX-Dokuments ist also nicht das LaTeX-Dokument selber, sondern vielmehr das bei seiner Kompilation entstehende Output-File.

Das folgende Beispiel soll den Unterschied zwischen WYSIWYG und WYSIWYM Programmen illustrieren:
Nehmen wir an, wir möchten eine linguistische Arbeit verfassen und dabei den griechischen kleingeschriebenen Buchstaben α (alpha) erwähnen. Um dies in Microsoft Word zu erreichen, könnte man die in Word integrierte Sammlung von Symbolen öffnen und das Symbol für α an der entsprechenden Stelle im Text einfügen. In LaTeX würde man anstatt ein α-Symbol einzufügen, lediglich den Command `$\alpha$` hinschreiben. Dieser Befehl hat optisch gesprochen natürlich mit dem gewünschten α-Symbol überhaupt nichts zu tun. Das tatsächliche α-Symbol erscheint nicht im LaTeX-Dokument, sondern erst im PDF-Dokument, welches bei der Ausführung der Befehle im LaTeX-Dokument erstellt wird.

---

1 Der Prozess des Kompilierens umfasst das Übersetzen des LaTeX-Files (Files mit der Dateiendung `.tex`) zu verschiedenen Output Formaten (über verschiedene Zwischenstufen).

2 Die Details der Kompilation gehen deutlich über den Rahmen dieses Kapitel hinaus. Interessierte finden auf der Webseite
https://www.sharelatex.com/learn/Choosing_a_LaTeX_Compiler
gibt einen guten ersten Überblick über den Ablauf der Kompilierung.

WSIWYYG-like Tools

Der Vollständigkeit halber soll noch erwähnt werden, dass es sogenannte *WSIWYYG-like* (WSIWYYG ähnliche) Tools für LaTeX gibt, welche z.B. Symbole wie das oben erwähnte α direkt in als eigentliche Symbole in dem LaTeX-Dokument und nicht erst im Output-File (PDF-Dokument) erscheinen lassen.
Beispiele solcher Tools sind
GNU TeXmacs: http://www.texmacs.org/tmweb/home/welcome.en.html
(nur für lokale Installationen),
BaKoMa TeX: http://www.bakoma-tex.com/
(nur für lokale Installationen)
oder der *Rich Text* Modus von Overleaf. Wir verwenden keines dieser WSIWYYG-like Tools und haben von den soeben erwähnten lediglich das Letzte (Overleaf Rich Text-Modus) etwas eingehender betrachtet. Wir finden, dass dieses sehr schön gemacht ist und sicherlich bei vielen Nutzern (auch bei erfahrenen Nutzern) Anklang finden wird. Andere Online Tools verfügen über recht ähnliche Features ($\rightarrow$ Ausprobieren lohnt sich).

2.3 Historische Entwicklung

TeX

TeX ist ein im Jahre 1978 veröffentlichtes Computerprogramm zur Erstellung von digitalen Texten, welches hauptsächlich von dem berühmten US-amerikanischen Informatiker Donald E. Knuth geschrieben wurde. Sein Hauptziel war es, mit TeX ein Werkzeug zu schaffen, mit dessen Hilfe hochqualitative Bücher mit wenig Aufwand verfasst werden können. Des weiteren sollte TeX auf jedem Computer und zu jeder Zeit identische Resultate (Texte) produzieren [3].
TeX ist insbesondere eine Programmiersprache. Diese ermöglicht die Definition diverser Befehle, welche den Computer z.B. beauftragen, die Schriftgrösse anzupassen, Abstände im Text einzubauen oder die Schriftart zu ändern. Eine TeX-Distribution enthält viele solcher bereits definierten primitive Befehle. Ein Beispiel eines solchen Befehls ist `\baselineskip = 24pt`, welcher bewirkt, dass zwischen den einzelnen Zeilen im Text ein Abstand von 24 Einheitspunkten gesetzt wird (in TeX beginnen Befehle mit einem Backslash '\').

[3] Details zur Geschichte von TeX findet man hier: https://www.tug.org/whatis.html

Nehmen wir an, wir möchten nun ein Textdokument wie dieses hier schreiben.
Konzentrieren wir uns einmal auf die Überschrift "Historische Entwicklung" dieses Abschnitts (Section). Wie ist diese wohl entstanden?
Um eine solche Überschrift in TEX zu erstellen, müssten dem Computer diverse Instruktionen (Befehle) übergeben werden:

- Befehle zur Anpassung der Schriftgrösse und Dicke der Schrift
- diverse Befehle zum Einfügen bestimmter Abstände vor und nach der Überschrift
- Anweisungen zur Nummerierung der verschiedenen Abschnitte
- usw.

Dies bedeutet, dass wir zahlreiche TEX-Befehle geschickt kombinieren müssten, um einen neuen Abschnitt deklarieren zu können. Dies erscheint recht unpraktisch zu sein. Was wäre jedoch, wenn jemand sich die Mühe machen würde, so einen komplexen Befehl für das Erstellen eines neuen Textabschnittes aus primitiven TEX-Befehlen zusammenzusetzen? Diese tatkräftige Person könnte der Kombination aller dafür benötigten primitiven Befehle einen neuen Namen, wie z.B. `\section{}` geben. Dieser Befehl steht dann als Stellvertreter für die Kombination aller Befehle, welche zur Erstellung eines neuen Abschnitts notwendig sind. Man bezeichnet den dadurch entstandenen neuen Befehl als *Makro* (griechisch $\mu\alpha\kappa\rho\acute{o}\sigma$ makros = 'gross', 'weit', 'lang').
Die Idee hinter Makros ist simpel und effektiv: man fasst alle kleinen (mikro) Befehle, welche benötigt werden, um z.B. einen neuen Abschnitt (Section) zu erstellen, zu einem neuen, grossen (makro) Befehl zusammen und gibt diesem einen Namen, wie beispielsweise `\section{}`. Dass dieses Vorgehen möglich ist, dürfte nicht erstaunen, da TEX wie bereits erwähnt, eine vollwertige Programmiersprache ist, mit welcher natürlich solche neuen Befehle definiert werden können.

LaTeX

Der beflissene US-amerikanische Informatiker Leslie B. Lamport baute am Anfang der 1980er Jahre zahlreiche höchst nützliche Makro-Befehle aus primitiven TEX-Befehlen zusammen. Das daraus resultierende Softwarepaket, welches die Benutzung von TEX mit Hilfe von Makros stark vereinfacht, nennt sich LaTeX (kurz für **La**mport **TeX**). LaTeX ist also nichts anderes als eine auf TEX aufbauende (und in TEX geschriebene) Erweiterung von TEX. Mittlerweile gibt es tausende von TEX-Packeten, welche von diversen Leuten

verfasst wurden und eine ungeheure Vielfalt an nützlicher Funktionalität zur TEX / LATEX Basisdistribution hinzufügen [4]. LATEX ist eine sogenannte *freie Software* und unterliegt den Regelungen der LaTeX Project Public License (LPPL) [5].

2.4 LATEX-Grundlagen

In diesem Abschnitt werden wir lediglich eine handvoll LATEX Befehle und Techniken kennen lernen. Ihr werdet aber schnell sehen, dass diese limitierten Kenntnisse bereits ausreichen um direkt in LATEX loslegen zu können:
Sobald man diese essentiellen Grundlagen kennt, kann alles Weitere bei Bedarf im Internet gefunden werden.
Diese essentiellen Grundlagen wollen wir hier kennenlernen.

Erstes LATEX-Dokument

Das einfachste LATEXDokument sieht so aus:

```
1 \documentclass{article}
2 \begin{document}
3 Dies ist mein erstes \LaTeX-Dokument!
4 \end{document}
```

Wenn man dieses kompiliert (wird bei Overleaf automatisch nach jeder Änderung des LATEX-Files gemacht), wird ein PDF-File mit folgendem Inhalt generiert:

Dies ist mein erstes LATEX-Dokument!

Unserem ersten LATEX-Dokument entnehmen wir, dass Commands in LATEX mit einem Backslash '\' beginnen (siehe auch Abschnitt 2.3). Der Command `\documentclass` legt fest, welche Art von Dokument wir schreiben möchten. Damit der Command wissen kann, welche Art von Dokument wir uns wünschen, müssen wir ihm die Zusatzoption `article` übergeben. Anstelle von `article` könnte man z.B. auch `book` oder `letter` angeben. Die allgemeine Form eines Commands ohne weitere Optionen oder mit einer Option ist also

[4] umfangreiche Sammlung von TEX-Paketen: https://ctan.org/?lang=en
[5] Details zur LPPL-Lizenz: https://www.latex-project.org/lppl/

```
\dosomething % ohne Zusatzoptionen
\dosomething{} % mit einer Zusatzoption.
```

Das Prozent Symbol % markiert den Anfang eines Kommentars (Comment). Alles, was nach diesem Symbol steht, wird ist ein Kommentar und wird bei der Kompilierung vom Computer ignoriert. Mit Kommentaren, lässt sich in Worten beschreiben, was im Code gemacht wird. Kommentierter Code ist meist (besonders für andere Leute) viel leserlicher als Code ohne Kommentar. Im obigen Beispiel ist `\LaTeX` ein Command ohne Zusatzoption. Der Befehl macht nichts anderes als das Wort 'LaTeX' im Design des LaTeXLogos zu schreiben.
Das Konstrukt

```
\begin{document}
...
\end{document}
```

definiert eine Umgebung (Environment), welches den Beginn und das Ende des Dokuments markiert. Dazwischen wird der eigentliche Inhalt des Dokuments geschrieben. Die allgemeine Form eines Environments ist:

```
\begin{something}
Hier soll etwas Interessantes geschehen!
\end{something}
```

Zusatzpakete importieren

Unser erstes LaTeX-Dokument 2.4 enthielt nur sehr elementare Befehle. Häufig möchte man dem Dokument aber weitere Funktionalität hinzufügen. Diese Funktionalität wird durch das Einfügen von Zusatzpaketen erreicht. Dazu muss das Zusatzpaket allerdings am Richtigen Ort auf dem Rechner installiert und gespeichert sein (bei lokalen Installation) oder von dem online Tool zur Verfügung gestellt werden (ansonsten muss man das Paket manuell installieren bzw. zur online Umgebung hinzufügen). Falls das Paket vorhanden ist, kann es durch den Befehl

```
\usepackage{NamesDesPakets}
```

in das LaTeX-File eingefügt werden. Wichtig ist, dass der Befehl vor dem

```
\begin{document}
...
\end{document}
```

Environment steht.

Dazu ein Beispiel mit einem Zusatzpaket:

```
\usepackage{xcolor} % stellt diverse Farben /
 ↪ Schattierungen zur Verfügung

\begin{document}
% verwende nun das Paket 'xcolor' um in Farbe zu
 ↪ schreiben:
\textcolor{red}{Dieser Text soll in roter Farbe geschrieben
 ↪ werden.}
\end{document}
```

ergibt:

Dieser Text soll in roter Farbe geschrieben werden.

Wenn man viele Pakete importiert, ist es stilistisch am besten, wenn man alle `\usepackage{...}`-Commands in einem separaten .tex-File (nennen wir es `preamble.tex`) speichert.
Der Vorteil liegt darin, dass man sein Hauptdokument nicht mit etlichen `\usepackage{...}`-Commands beginnen muss, sondern diese Commands mit dem Befehl `\input{preamble}` (muss vor `\begin{document}` stehen) einfügen kann.

Das Präambel-File der Autoren (Name: `preamble.tex`) dieses Skripts wird auf Moodle oder GitHub zur Verfügung gestellt. **Es muss in demselben Verzeichnis liegen wie das Haupt-LaTeX-File**. Dadurch kann garantiert werden, dass Ihr alle Beispiele in diesem Skript und den Übungen ohne Fehlermeldungen kompilieren könnt.

Sections und subsections

Die Einteilung des Dokuments in Abschnitte und Unterabschnitte (oder auch Kapitel, Unterkapitel etc.) trägt stark zur Übersichtlichkeit und Struktur des Dokuments bei (siehe Beispiel 2.4).
Der Befehl `\\` erzwingt einen Zeilenumbruch. Die Nummerierung kann durch das Hinzufügen des Symbols '`*`' unterdrückt werden:
`\section*{...}`, `\subsection*{...}`.

```
\section{Abschnitt}
Habe nun, ach!  Philosophie,\\
Juristerei und Medizin,\\
Und leider auch Theologie\\
Durchaus studiert, mit heißem Bemühn.
\subsection{Unterabschnitt}
Da steh ich nun, ich armer Tor!
\subsubsection{Unterunterabschnitt}
Und bin so klug als wie zuvor;
\subsection{noch ein Unterabschnitt}
Heiße Magister, heiße Doktor gar\\
Und ziehe schon an die zehen Jahr\\
Herauf, herab und quer und krumm\\
Meine Schüler an der Nase herum-\\
Und sehe, daß wir nichts wissen können!
```

kompiliert zu →

1 Abschnitt

Habe nun, ach! Philosophie,
Juristerei und Medizin,
Und leider auch Theologie
Durchaus studiert, mit heißem Bemühn.

1.1 Unterabschnitt

Da steh ich nun, ich armer Tor!

1.1.1 Unterunterabschnitt

Und bin so klug als wie zuvor;

1.2 noch ein Unterabschnitt

Heiße Magister, heiße Doktor gar
Und ziehe schon an die zehen Jahr
Herauf, herab und quer und krumm
Meine Schüler an der Nase herum-
Und sehe, daß wir nichts wissen können!

Listen, Aufzählungen

Listen können ganz einfach wie folgt erstellt werden

```
\begin{itemize}
\item erstens
\item zweitens
\item ...
\end{itemize}
```

erzeugt folgende Liste

- erstens
- zweitens
- ...

Aufzählungen lassen sich sehr ähnlich erstellen:

```
\begin{enumerate}
\item erstens
\item zweitens
\item ...
\end{enumerate}
```

erzeugt die Aufzählung

1. erstens
2. zweitens
3. ...

Mit dem Parameter z.B. `\begin{enumerate}[label=\roman*.]` erfolgt die Aufzählung mit römischen Zahlen.

Formeln, Gleichungen, mathematische Symbole

LaTeX brilliert ganz besonders beim Verfassen mathematischer Texte und Aufschreiben von Gleichungen.
Schauen wir uns folgendes, umfangreiches Beispiel an:

```
Betrachte die quadratische Gleichung
\begin{equation}
\label{quadratische_gleichung}
ax^2+bx+c=0,
\end{equation}
mit $a,\,b,\,c\in\mathbb{R}$ und $a\neq 0$.\\

\noindent
Definiere die Diskriminante $\Delta = b^2-4ac$.\\
Dann hat die quadratische Gleichung
 ↪ \ref{quadratische_gleichung} folgende Lösungen in den
 ↪ reellen Zahlen:
\begin{enumerate}[label=\roman*.]
\item die zwei verschiedene reellen Lösungen
\begin{equation}
x_{1,2}=\frac{-b\pm\sqrt{\Delta}}{2a},
\end{equation}
falls $\Delta > 0$,
\item genau eine reelle Lösung
\begin{equation}
x=-\frac{b}{2a},
\end{equation}
falls $\Delta = 0$,
\item keine reellen Lösungen, falls $\Delta < 0$.
```

Dieser LaTeX-Code wird uns folgenden Output liefern:

Betrachte die quadratische Gleichung

$$ax^2 + bx + c = 0, \tag{2.1}$$

mit a, b, $c \in \mathbb{R}$ und $a \neq 0$.

Definiere die Diskriminante $\Delta = b^2 - 4ac$.
Dann hat die quadratische Gleichung 2.1 folgende Lösungen in den reellen Zahlen:

i. die zwei verschiedene reellen Lösungen

$$x_{1,2} = \frac{-b \pm \sqrt{\Delta}}{2a}, \tag{2.2}$$

falls $\Delta > 0$,

ii. genau eine reelle Lösung

$$x = -\frac{b}{2a}, \tag{2.3}$$

falls $\Delta = 0$,

iii. keine reellen Lösungen, falls $\Delta < 0$.

Hier haben wir recht viele neue Konzepte benutzt. Schauen wir uns das Beispiel mit der quadratischen Gleichung etwas genauer an. Wir verwenden hier das `equation`-Environment, welches Anfang und Ende einer neuen Gleichung (oder eines allgemeinen mathematischen Ausdrucks) kennzeichnet. Wir geben dieser Gleichung das Label `\label{quadratische_gleichung}` um später im Text auf diese Gleichung mit `\ref{quadratische_gleichung}` Referenz nehmen zu können. Referenzen sind ungeheuer wichtig in wissenschaftlichen Arbeiten. Man beachte auch, dass die Gleichung durch das `equation`-Environment eine Nummer erhalten hat. Damit hat die Gleichung eine 'Namen' bekommen.
Durch die Dollar-Symbole kann innerhalb des Textflusses in die Mathematik-Umgebung (*math environment*) gewechselt werden.
Der Command `$\frac{a}{b}$` ergibt den Bruch $\frac{a}{b}$ und `$\sqrt{a}$` die Quadratwurzel (*square root*) $\sqrt{a}$.
Das `enumerate`-Environment haben wir bereits in 2.4 kennengelernt. Die Bedeutung der anderen Commands sollte aus dem Kontext klar sein.

Makros

Zum Schluss wollen wir uns noch der Erstellung von Markos widmen, welche bereits in 2.3 kurz angesprochen wurden.
Die nächsten zwei Beispiele dienen der Illustration der Funktionsweise von Makros in LaTeX.

Beispiel 1 (ohne Argumente)
Angenommen du musst in einer Arbeit in der Chemie die Zusammensetzung des Sprengstoffs *TNT* untersuchen. In deinem Laborbericht möchtest du ab und zu den vollständigen Namen von TNT, nämlich **2-Methyl-1,3,5-trinitrobenzen** erwähnen (in Fettschrift). Dazu müsste man an den entsprechenden Stellen jeweils den Command
`\textbf{2-Methyl-1,3,5-trinitrobenzen}` hinschreiben. Wenn man dies oft tun muss, ist es einfach, diesen Command durch ein Makro zu ersetzen, welchem wir den Namen `\TNT` geben möchten:

```
\newcommand{\TNT}{\textbf{2-Methyl-1,3,5-trinitrobenzen}}
```

Von nun an, kann in LaTeX einfach `\TNT` geschrieben werden. Bei der Kompilierung wird jedes `\TNT` durch **2-Methyl-1,3,5-trinitrobenzen** ersetzt werden. Dies macht das Schreiben einfacher und das LaTeX-File bleibt übersichtlicher.

Beispiel 2 (mit Argumenten)
In einer Arbeit über Geometrie sind wir es leid, für viele verschiedene Zahlen a und b den Ausdruck $\sqrt{a^2+b^2}$ aus dem Satz des Pythagoras aufzuschreiben. Wir schreiben lieber ein Makro mit dem Namen `\pythagoras` schreiben:

```
\newcommand{\pythagoras}[2]{\sqrt{#1^2+#2^2}}
```

Von nun an können wir anstelle von `$\sqrt{3^2+4^2}$` einfach `$\pythagoras$` schreiben um $\sqrt{3^2+4^2}$ zu erhalten. Man beachte, dass wir dem Makro eine Anzahl von zwei Makros (`$[2]$`) als Input übergeben.
Die allgemeine Struktur eines solchen Makros ist also:

```
\newcommand{\NameMakro}[Anzahl Argumente]{Definition}
```

2.5 Nützliche Ressourcen

Templates (Vorlagen)

Wir empfehlen, neue LaTeX-Projekte mit einer bereits bestehenden Vorlage (sogenannten *Templates*) zu beginnen.
Gute Quellen für LaTeX-Templates sind:

- https://www.overleaf.com/latex/templates/
- https://www.latextemplates.com/

Nachschlagewerke

- **Internet-Suchmaschinen**
- ausführliches Tutorial:
  http://web.mit.edu/jgross/Public/latex/lshort.pdf
- kurzes Tutorial:
  http://www.ipcms.unistra.fr/wp-content/uploads/2018/04/latex.pdf
- Symbol-Erkennungstool:
  http://detexify.kirelabs.org/classify.html
- bei Fragen und Problemen:
  https://tex.stackexchange.com/
- kurze Übersicht mathematischer Symbole:
  https://oeis.org/wiki/List_of_LaTeX_mathematical_symbols
- riesige Sammlung von LaTeX-Symbolen: http://tug.ctan.org/info/symbols/comprehensive/symbols-a4.pdf
- Wikibook
  https://en.wikibooks.org/wiki/LaTeX

Übungsaufgaben

- https://guides.nyu.edu/c.php?g=601858&p=4168140

Kapitel 3

Programmieren in Python I

3.1 Wahl der Programmiersprache

Es existieren dutzende bekannte Programmiersprachen [1]. Da die meisten Programmiersprachen jedoch viele Konzepte teilen, und auch oft eine starke Verwandtschaft aufweisen, wird dem Kenner von mindestens einer Sprache, das Hinzulernen einer weiteren Sprache deutlich leichter fallen. Deshalb genügt es meist, nur eine oder wenige Sprachen gut zu kennen. Mit diesen Vorkenntnissen können dann weitere Sprachen, bei Bedarf, recht schnell dazugelernt werden.
Wir haben uns entschieden, in diesem Kurs mit der Programmiersprache 'Python' zu arbeiten. Die Wahl fiel uns nicht schwer, da viele Gründe für sie sprechen: Python ist modern, enorm stark verbreitet, universell einsetzbar und höchst einsteigerfreundlich.

Zu diesem Thema lassen sich Video-Erklärungen auf unserem YouTube-Kanal
www.youtube.com/channel/UCxIIMOe4oVImeJQbsphVjug/playlists
finden.
Viele der Beispiele, die wir in diesem Kapitel besprechen werden, sind stark motiviert durch die Beispiele, welche im exzellenten Python-Tutorial der *Python Software Foundation*:
https://docs.python.org/3/tutorial/
gefunden werden können.

[1] Liste von verbreiteten Programmiersprachen:
https://en.wikipedia.org/wiki/List_of_programming_languages

3.2 Python Befehle im Terminal und erstes Programm

An den Computern der Schule werden wir mit einer bereits vorinstallierten Python-Distribution arbeiten. Für das Arbeiten in Python an privaten Rechnern, empfehlen wir die Python-Distribution *Anaconda* zu installieren. Genaue Anleitungen und Informationen zu Anaconda findet Ihr im Anhang A oder auf unserem YouTube-Kanal.
Alle folgenden Betrachtungen sind allgemeingültig und unabhängig vom persönlichen Python-Setup.

3.2.1 Python Befehle im Terminal

Um Python-Code ausführen zu können, müssen wir das Terminal unter macOS starten. Dazu klickt man oben rechts auf *Spotlight* (Lupen-Symbol) und sucht im Suchfeld nach 'terminal', um das Terminal zu öffnen. Im Terminal gibt man das Wort `python` ein und drückt anschliessend die *ENTER*-Taste. Dadurch gelangt man in das Python-Environment. Als kleinen Test kann man den Befehl `import this` eingeben und sollte dann folgenden Output im Terminal erhalten:

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Listing 1: The Zen of Python

Die Auflistung 1 ist eine Sammlung von Software-Prinzipien, welche das Design der Python-Programmiersprache beeinflussen. Beim Command `import this` handelt es sich um ein sogenanntes *easter egg*[2].
Im Folgenden wollen wir viele weiter Python-Commands kennenlernen.
Als erstes wollen wir sehen, wie uns das Python-Terminal als gewöhnlicher Taschenrechner dienen kann:

```
>>> 4+7*3 # Punkt vor Strich
25
>>> 8/5 # gewöhnliche Division
1.6
>>> 8//5 # ganzzahlige Division (floor division)
1
>>> 12%7 # modulo Operation (Rest der ganzzahligen Division)
5
>>> (80-4*5)/2 # Klammersetzung
30.0
>>> 5**2 # 5 hoch 2
25
```

Kommentare haben wir bereits in LaTeX in Kapitel 2 kennengelernt. In LaTeX beginnen Kommentare mit dem Prozent-Symbol (%), in Python hingegen werde Kommentare mit dem Hashtag-Symbol (#) eingeleitet.

3.2.2 Erstes Python-Programm

Das Python-Terminal ist nützlich um einzelne Befehle auszuführen. Ganze Python-Programme lassen sich aber komfortabler in einem Texteditor schreiben. Dazu öffnen wir, in einem beliebigen Texteditor, ein neues Dokument und schreiben Folgendes in das (noch) leere Dokument:

```
print("Hello, World!")
```

Nun geben wir dem Dokument den Namen `hello_world.py` und speichern es auf dem `Desktop` ab. Dies ist unser erstes Python-Programm. Um das Programm auszuführen, öffnen wir ein neues Terminal (da wir dafür nicht im Python-Environment sein wollen) und führen den Befehl:

```
cd Desktop/
```

[2] Gute Erklärung des Begriffs "easter egg": https://en.wikipedia.org/wiki/Easter_egg_(media)

aus.
Der Befehl `cd` steht für *change directory* und lässt uns in das Verzeichnis *Desktop* wechseln, wo unser `hello_world.py` File gespeichert ist. Das Programm `hello_world.py` kann nun durch folgenden Befehl (im Terminal) ausgeführt werden

```
python hello_world.py
```

und sollte den Output

```
Hello, World!
```

liefern.
Typischerweise, schreibt man (mehrzeilige) Programme in einem Texteditor. Das Python-Terminal ist aber zum Testen einzelner Commands hervorragend geeignet.

3.3 Grundoperationen

3.3.1 Variablen deklarieren, Typ einer Variablen

Eine der häufigsten Tätigkeiten beim Programmieren in Python,- ist das Deklarieren von Variablen. Das Gleichheitszeichen (=) wird verwendet um Variablen einen Wert zuzuweisen (assignment operator):

```
>>> a = 15 # a soll 15 sein
>>> b = 5 # b soll 5 sein
>>> c = a + b # c soll die Summe von a und b sein
>>> print(c) # gib die Variable an den Output aus
20
>>> x = "wundervoll" # x ist die Zeichenfolge (string) wundervoll
>>> print("Python ist " + x) # füge zwei Strings zusammen
Python ist wundervoll
>>> word = 'Python'
>>> word[0] # 1. Zeichen des Wortes
'P'
# (in Python hat das erste Element die Postion 0 und nicht 1)
>>> word[1] # 2. Zeichen des Wortes
'y'
>>> word[-1] # letztes Zeichen des Wortes
'n'
# Zeichen von Position 0 (enthalten) bis 2 (nicht enthalten):
```

```
>>> word[0:2]
'Py'
# Zeichen von Position 3 (enthalten) bis zum Ende
>>> word[3:]
'hon'
>>> x = 1 # x ist eine ganze Zahl (integer)
print(type(x)) # der Typ von x ist integer (int)
<class 'int'>
>>> y = 1.0 # y ist eine Gleitkommazahl (floating point number)
>>> z = 1.1 # z ist auch eine floating point number (float)
>>> print(type(y)) # der Typ von y ist float
<class 'float'>
>>> print(type(z))
<class 'float'>
>>> print(type(word)) # word ist ein String (str)
<class 'str'>
# die Funktion len() sagt uns die Länge eines Strings
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

3.3.2 Input, Typecasting

Öffne ein neues Textfile und speichere dieses als `my_python.py` auf dem Desktop. Schreibe folgenden Code in das File und speichere erneut.

```
print("Enter your name:")
x = input()
print("Hello, " + x)
```

Wie bereits bei unserem `Hello, World` Programm (siehe Unterabschnitt 3.2.2), kann das Programm mit dem Command `python my_python.py` im Terminal ausgeführt werden.
Der Command `x = input()` liest den Input aus dem standard-Inputstream (üblicherweise der Input von der Tastatur) und speichert ihn als String in der Variable `x`.
Was ist aber, wenn wir möchten, dass der User eine (ganze) Zahl und nicht ein String eingeben kann? Würde man nämlich im obigen Programm anstelle eines Namens, eine Zahl mit der Tastatur eingeben (z.B. `42`), dann würde diese Folge von Zahlen nicht als der Integer (Ganzzahl) `42` gespeichert werden, sondern als der String `'42'`. Dieses Problem führt uns zum Begriff

des *Typecastings*. Typecasting ist die Umwandlung eines Datentyps in einen anderen:

```
a = 1.8 # a ist vom Typ float
# a wird explizit in einen Integer umgewandelt
# dies geschieht durch das Weglassen des dezimalen Teils
a = int(1.8) # a hat nur den ganzzahligen Wert 1
# dieses Programm ermöglicht dem User die Eingabe eines Integers
print("Gib eine ganze Zahl ein:")
# der Tastatur Input (String) wird explizit zu einem
# Integer konvertiert (Typecasting)
x = int(input()) # int(...) = explizites Casting nach int
print("Du hast folgende Zahl eingegeben:", x)
# implizites Typecasting findet hier statt:
b = 10/5.0 # b wird den float Wert 2.0 (und nicht 2) haben
# der Integer 10 wurde implizit zu einem float konvertiert
```

3.4 Control Flow

Alle unsere bisherigen Python-Programme wurden von oben nach unten ausgeführt (linear control flow). Diese Art von Control Flow ist sehr restriktiv, da jeder Command in unserem Programm maximal einmal ausgeführt wird. Nehmen wir an, wir möchten ein Programm schreiben, welches eine Million Schritte auf irgend einem Input durchführt. Mit linearem Control Flow hätte dieses Programm mindestens eine Million Zeilen. Offensichtlich ist dies nicht wünschenswert. Wir benötigen deshalb mächtigere Tools, um mit solchen Aufgaben sinnvoll umgehen zu können.
Zu den wichtigsten Konzepten, die einen nicht linearen Control Flow ermöglichen, gehören sogenannte *control statements* wie
`for`-loops, `while`-loops, `if`-statements und `jump`-statements.
Diese wollen wir nun anhand von Beispielen kennenlernen.

3.4.1 for-loops

Loops sind, in praktisch jeder Programmiersprache, ungeheuer wichtig. Betrachten wir dazu einige Beispiele:

```
# for-loop, welcher die Zahlen 0, 1, 2, ..., 8, 9 ausgibt:
for i in range(10):
    print(i) # Output: 0, 1, 2, ..., 8, 9

```

```
for i in range(3,8):
   print(i) # Output: 3, 4, 5, 6, 7

# Berechne die Summe der Ersten
# 100'000 Zahlen (sum = 1+2+3+...+100000)
sum = 0 # initialisiere die Summe mit 0
for k in range(1,100001):
   sum = sum + k # erhöhe die Summe um den nächsten Summanden
print(sum) # Output: 5000050000

# erstelle eine Liste von Strings
# eine Liste hat die Form [Element 1, Element2, ...]
languages = ['C++', 'Python', 'Fortran', 'Go', 'Assembly', 'C']
# sortiere die Liste alphabetisch
# die Bedeutung des Punktes "." in "list.sort" wird noch
↪  besprochen
list.sort(languages)
# \n bewirkt einen Zeilenumbruch:
print('Bekannte Programmiersprachen:\n')
# der Index 'j' geht durch die Liste "languages" durch
for j in languages:
    print(j) # Output: C++, Python, Fortran, Go, Assembly, C
```

Man beachte die Einrückung (indentation) nach Beginn des for-loops:

```
for i in range(10):
    print(i)
```

Alles, was unter dem for-loop eingerückt ist, gehört zum Körper (body) des Loops. Blöcke von Code werden in Python nicht, wie in sehr vielen anderen Programmiersprachen (wie `R`, `C++`, `C`), durch geschweifte Klammern strukturiert, sondern durch Einrücken. Dadurch soll der Code leserlicher werden. Obiger Loop würde in `R` z.B. so aussehen:

```
for(i in 0:9) {
  print(i)
}
```

Dabei ist die Einrückung der `print()`-Funktion in `R` reine Kosmetik und könnte auch weggelassen werden:

```
for(i in 0:9) {
print(i)
}
```

Ebenfalls obligatorisch in Python ist der Doppelpunkt am Ende der Deklaration des Loops: `for i in range(10):`

3.4.2 while-loops

Der while-loop ist sehr ähnlich aufgebaut wie der for-loop. Es lässt sich beweisen, dass jeder while-loop als for-loop geschrieben werden kann und umgekehrt. So gesehen, sind die beiden Konstrukte äquivalent.
Das Konstrukt der while-loops wird typischerweise dann eingesetzt, wenn nicht von Anfang an (a priori) klar ist, wie viele Schritte ausgeführt werden müssen. Als Beispiel dazu, betrachten wir die Fibonacci-Folge. Die Fibonacci-Folge $f_1, f_2, f_3, ...$ ist für $n > 2$ rekursiv definiert durch

$$f_n = f_{n-1} + f_{n-2}, \tag{3.1}$$

mit den Anfangswerten $f_1 = f_2 = 1$.
Nun wollen wir alle Folgeglieder der Fibonacci-Folge ausgeben, die kleiner sind als z.B. 500. Wie viele solche Folgeglieder dies sind, wissen wir (noch) nicht:

```
# Fibonacci-Folge mit while-loop
a, b = 0, 1 # Anfangswerte: a = 0, b = 1
# führe den while-loop aus, solange a kleiner ist als 500
while a < 500:
    print(a)
    a, b = b, a+b
# Output:
# 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377
```

3.4.3 if-statements

Die sogenannten if-statements werden beim Programmieren ständig gebraucht. Ihre Funktionsweise wollen wir wieder anhand einfacher Beispiele aufzeigen:

```
a = 50
b = 312.3
if a > b:
   # kein Output, da a nicht grösser ist als b
    print('a ist groesser als b')

x = 10.0
```

```
y = 0.0
if y != 0: # != steht für "nicht gleich"
    z = x/y # die Division wäre hier erlaubt, da y nicht Null
     ↪  ist
else:
   # in diesem Fall würden wir durch Null teilen
    print('ACHTUNG! DIVISION MIT NULL!')

# gib alle nicht-negativen Zahlen < 100 aus,
# die durch 17 teilbar sind:
for i in range(100):
    # % ist der modulo-Operator
    # == ist der Vergleichsoperator
    if i%17 == 0:
        print(i)

x = -23.21329
if x > 0:
    print('x ist eine positive Zahl')
elif x == 0: # else if
    print('x ist genau Null')
else:
    print('x ist negativ')
```

3.4.4 jump-statements

Die jump-statements *continue* und *break* sind gelegentlich sehr nützlich. Ihre Bedeutung sollte aus einem Beispiel klar werden:

```
# errate die richtige Kombination eines 4-Stelligen
# Zahlenschlosses
kombination = 3110 # richtige Kombination

# der while-loop wird wiederholt, bis die richtige
# Kombination eingegeben wird
while True: # ist immer true
    print('errate die 4-Stellige Kombination')
    x = int(input())
    if len(str(x)) != len(str(kombination)):
        print('die Eingabe hat die falsche Laenge')
```

```
12          continue # der Rest des Loops kann übersprungen
             ↪  werden
13      if x == kombination:
14          print('richtig geraten!')
15          break # der while-loop kann nun abgebrochen werden
16      else:
17          print('leider falsch')
```

Nach Ausführung des `continue`-statements, wird der Rest der aktuellen Iteration (Schritt) übersprungen und mit dem nächsten Loop-Durchgang weiter gemacht. Das `break`-statement bricht den Loop ab.

3.5 Funktionen in Python

Eines der wichtigsten Konzepte in Python ist das der Funktionen. Wir haben in diesem Kapitel schon einige Funktionen angetroffen, sind jedoch nicht näher auf diese eingegangen.
Zum Beispiel sind wir der Funktion `print(...)` häufig begegnet und haben (in Unterabschnitt 3.3.1) die Funktion `len(s)` benutzt, welche uns die Länge des Strings `s` genannt hat. Man beachte die grosse Ähnlichkeit dieser Python-Funktionen zu dem Funktionsbegriff, den wir aus der Mathematik kennen:
Genau wie in der Mathematik, hat eine Python-Funktion ein Name wie z.B. `len` (in der Mathematik wird als Funktionsname häufig f gewählt) und akzeptiert eines oder mehrere Argumente `len(x)` $\leftrightarrow f(x)$. Um diese Ähnlichkeit noch besser zu verstehen, schauen wir uns an, wie in Python eine lineare Funktion

$$f(x) = ax + b, \tag{3.2}$$

mit $a, b \in \mathbb{R}$ und $a \neq 0$ (falls $a = 0$ gilt, ist die Funktion konstant und nicht linear). In Python lässt sich die lineare Funktion 3.2 wie folgt implementieren:

```
1  a = 2
2  b = 3
3  def lineare_funktion(x):
4      y = a*x + b
5      return(y)
6
7  # Aufruf der linearen Funktion mit dem Argument x == 5
8  t = lineare_funktion(5)
9  # t hat den Wert 2*5 + 3 = 13
10 print(t) # Output: 13
```

Eine Python-Funktion beginnt immer mit dem Keyboard **def**. Danach schreibt man, wie die Funktion heissen soll und (in Klammern) welche Funktionsargumente sie benötigt.
Was ist nun aber, wenn wir eine allgemeine lineare Funktion, mit beliebigen Werten der Parameter a (Steigung) und b (y-Achsenabschnitt) haben möchten? Ganz einfach, wir übergeben a und b ebenfalls als Funktionsargumente:

```
def lineare_funktion(x,a,b):
    y = a*x + b
    return(y) # gibt den Wert y zurück (return value)

# Aufruf der linearen Funktion mit dem Argument x == 5, a ==
 ↪  2 und b == 3
t1 = lineare_funktion(5,2,3)
# t1 hat den Wert 2*5 + 3 = 13
# um sicher zu sein, dass wir die Funktionsargumente nicht
 ↪  vertauschen
# können wir sie auch explizit beim Funktionsaufruf angeben
t2 = lineare_funktion(b=3,x=5,a=2) # t2 == t1
```

Um die Nützlichkeit von Python-Funktionen noch mehr zu betonen, schauen wir uns nochmals das Beispiel der Fibonacci-Folge aus Unterabschnitt 3.4.2 an. In diesem Beispiel wollten wir alle Glieder der Fibonacci-Folge die kleiner als 500, sind ausgeben. Wenn wir unsere Fibonacci-Folge als Python-Funktion $fibonacci(n)$ umformulieren, dann wird diese in der Lage sein, alle Fibonacci-Zahlen, die kleiner als n sind auszugeben. Dies ist natürlich viel nützlicher und universeller, als sich auf eine fixe Zahl wie z.B. $n = 500$ zu beschränken:

```
# Fibonacci-Folge als Funktion
def fibonacci(n):
    # gibt alle Glieder der Fibonacci-Folge < n aus
    a, b = 0, 1 # Anfangswerte der Folge
    while a < n:
        print(a)
        a, b = b, a+b

fibonacci(4000) # Aufruf der Funktion für n == 4000
```

Man beachte, dass die Funktion `fibonacci(n)`, im Gegensatz zu der linearen Funktion, keinen Rückgabe-Wert (return value) hat.

Kapitel 4

Programmieren in Python II

In diesem Kapitel wollen wir weitere grundlegende Konzepte des Programmierens kennenlernen.

4.1 Listen

Bislang haben wir vor allem Programme geschrieben, die Operationen auf Strings und Zahlen durchführen und ihr Resultat an das Terminal ausgeben. Was wir bislang noch nicht getan haben, ist mit Daten zu arbeiten. Doch gerade Probleme, die irgendwelche Operationen auf Daten ausführen sind äusserst bedeutend.
Betrachten wir dazu ein einfaches Beispiel:
An der Kantonsschule Im Lee ist die E-Mail-Adresse der Schülerinnen und Schüler von folgender Form
Vorname.Nachname@stud.ksimlee.ch.
Wenn zwei Schülerinnen oder Schüler denselben Namen haben (Vorname und Nachname), ist diese Form jedoch nicht eindeutig. Wie könnte man eine Python-Funktion schreiben, welche überprüft, ob so ein Fall vorkommt (bei den gegeben Schulklassen)? Dazu müsste man der Funktion die Namen aller Schülerinnen und Schüler übergeben. Dies könnte man z.B. wie bisher mit Variablen tun:

```
student_1 = "Russell Crowe"
student_2 = "Ralph Fiennes"
student_3 = "Natalie Portman"
student_4 = "Anne Hathaway"
student_5 = "Joaquin Phoenix"
```

...
...

Nun hat es an der Schule aber mehrere hundert Schülerinnen und Schüler. Das heisst, die Funktion hätte mehrere hundert Argumente. Weitere Probleme gibt es dann bei der Prüfung der Name auf Duplikate. Wie sollte man z.B. alle diese Variablen gegeneinander vergleichen?
Wir stellen fest, dass es hier keinen Sinn macht mit einfachen Variablen und Strings zu arbeiten. Stattdessen, benötigen wir Daten-Container, welche es erlauben, viele Objekte (z.B. Namen von Personen) auf einmal zu speichern. Zu diesen Containern gehören beispielsweise *Listen*, denen wir in Unterabschnitt 3.4.1 bereits einmal rasch begegnet sind.
Listen sind ein sehr wichtiges Grundwerkzeug beim Programmieren. Sie sind in der Lage, viele Werte von unterschiedlichen Typen zu speichern. Man kann sich eine Liste Vorstellen wie ein Container für Daten.
Im Folgenden möchten wir mit dem Konzept der Liste vertraut werden und sehen, wie sie in Python realisiert werden. Sehr ähnliche Konzepte gibt es in praktisch jeder Programmiersprache.

4.1.1 Umgang mit Listen in Python

Besuche die Seite

`https://www.w3schools.com/python/python_lists.asp`

und arbeite Dich durch die dort gegebenen einfachen Beispiele durch.
Versuche danach die Übungsaufgaben zu lösen.

Kapitel 5

Algorithmen

In diesem Kapitel möchten wir uns mit Algorithmen beschäftigen und verstehen, was Algorithmen sind und wozu sie nützlich sind.
Ein Algorithmus ist eine eindeutige Handlungsvorschrift zur Lösung eines Problems oder einer Klasse von Problemen. Algorithmen bestehen aus endlich vielen, wohldefinierten Einzelschritten[1].
Wenn wir Programme als Algorithmen betrachten, fordern wir jedoch, daass ein solches Programm für jede zulässige Eingabe hält und eine Ausgabe liefert. Es ist also für Algorithmen unzulässig, unendlich lange (in einer Endlosschleife) zu laufen.

Dies erinnert an ein Kochrezept. Ein Rezept ist ebenfalls eine klare Handlungsvorschrift, die aus endlich vielen, wohldefinierten Einzelschritten besteht (Menge der Zutaten, Ofentemperatur, Kochzeit, Zubereitungsart etc.). Das Rezept löst dabei das Problem der Zubereitung eines bestimmten Gerichts.
Es stellt sich nun die Frage, in welchem Sinne eine Kochrezept ein Algorithmus ist und in welcher Hinsicht nicht.
Um diese Frage besser beantworte zu können, betrachten wir ein Beispiel eines Rezeptes für das Backen von Brot:

[1] aus Wikipedia: https://de.wikipedia.org/wiki/Algorithmus

Rezept für das Backen von *Bürli*

Zutaten (für 6 Bürli):

- Mehl
  - 90 g Vollkornmehl
  - 312 g Halbweissmehl
  - 198 g Weissmehl
- 13 g Salz
- 1 g Trockenhefe
- 492 g warmes Wasser

Vorgehensweise:

1. Wasser in eine Schüssel giessen
2. Trockenhefe ins Wasser geben, wenig umrühren, etwas warten
3. Salz dazu geben und auflösen lassen
4. Mehle in der Schüssel mischen. Mehl ins Wasser geben
5. Mischen, bis die Masse homogen ist
6. 30 min Teigruhe (zugedeckt), danach: erste Faltung
7. 30 min Teigruhe (zugedeckt), danach: zweite Faltung
8. 3 h Teigruhe (zugedeckt) bei Raumtemperatur
9. Zusätzliches Mehl grosszügig auf einer Fläche ausstreuen
10. Teig auf der Fläche ausbreiten, danach: dritte Faltung
11. 15 min (zugedeckt) Teigruhe, danach: Teig umdrehen
12. 15 min (zugedeckt)
13. Teigmasse in 6 gleichgrosse Stücke schneiden
14. Stücke zu runden Brötchen formen
15. Backofen und Brotstein auf 240° Celsius vorwärmen

16. geformte Brötchen (zugedeckt) für 20 min ruhen lassen (während der Ofen aufwärmt)

17. kurz vor dem Bachvorgang: Wasserbad mit kochendem Wasser in den Ofen geben

18. Brötchen auf den heissen Brotstein in dem vorgewärmten Ofen legen

19. insgesamt 26 min bei 240° Celsius backen (nach den ersten 15 min: Brötchen neu arrangieren)

20. Auf einem Metallgitter ausfkühlen lassen

Der Output dieses Rezeptes sollte ungefähr so aussehen wie in Bild 5.1.

Abbildung 5.1: Teil des Outputs des Brotback-“Algorithmus” 5

Erfahrungsgemäss hängt die Qualität des Essens jedoch nicht nur vom Rezept ab, sondern auch von dem Können und der Erfahrung der Person, welche es zubereitet. Dies bedeutet aber, dass Rezepte anscheinend nicht immer zu dem gleichen Ergebnis führen. Dies motiviert uns zur Formulierung einer ersten zusätzlichen Bedingung an einen Algorithmus:

Jede Ausführung eines Algorithmus sollte zu dem gleichen Resultat führen, unabhängig davon von wem oder von was er ausgeführt wird.

Der Leserin / dem Leser wird aufgefallen sein, dass wir uns bei der Definition dieses wichtigen Begriffs ungewöhnlich wage ausdrücken.

Die mangelnde mathematische Genauigkeit des Begriffs Algorithmus störte viele LogikerInnen und MathematikerInnen des 19. und 20. Jahrhunderts, weswegen in der ersten Hälfte des 20. Jahrhunderts eine ganze Reihe von Ansätzen entwickelt wurde, die zu einer genauen Definition führen sollten. Diese Entwicklungen gipfelten in der Definition des Begriffs *Algorithmus* durch das Modell der *Turingmaschine*. Mit diesem werden wir uns in einem späteren Kapitel über *Theoretische Informatik* beschäftigen.

Wir werden nun zwei konkrete Algorithmen konstruieren. Anhand dieser Beispiele, werden wir zwei nützliche Techniken im Umgang mit Algorithmen illustrieren. Dabei handelt es sich um *den Beweis der Korrektheit eines Algorithmus unter Verwendung einer loop-Invariante* und um den Begriff der *Laufzeitanalyse*.

In diesem Kapitel möchten wir uns nicht um die genauen Formalitäten von Programmiersprachen kümmern. Deshalb werden wir Algorithmen in sogenanntem Pseudocode angeben. Pseudocode wird ähnlich aussehen wie Code in einer echten Programmiersprache (wie z.B. Python). Pseudocode muss nicht den genauen Regeln einer konkreten Programmiersprache gehorchen muss. Deshalb sind wir bei Schreiben von Pseudocode frei, Ausdrücke zu wählen, welche uns erlauben, den Algorithmus so einfach und deutlich wie möglich zu beschreiben.

5.1 FIND_MAX

Angenommen, wir haben eine Liste A mit n Zahlen gegeben (Liste der Länge n). Wir möchten nun einen Algorithmus konstruieren, welcher das maximale Element in A findet. Ein mögliches Vorgehen wäre, das erste Element in der Liste $A[1]$ zu betrachten. Da wir bislang nur dieses eine Element gesehen haben, ist dies auch (trivialerweise) das maximale Element in A, welches wir bislang gesehen haben. Deshalb betrachten wir $A[1]$ als unser aktuelles Maximum. Nun gehen wir in einer Schleife durch die weiteren Elemente $A[2] ... A[n]$ durch. Immer, wenn wir ein Element finden, welches grösser ist, als unser aktuelles Maximum, dann werden wir dieses Element als unser aktuelles Maximum setzen. Der Wert des aktuellen Maximums, welcher am Ende der Schleife angenommen wurde, ist das Maximum der gesamten Liste. Die soeben gemachten Überlegungen lassen sich in Pseudocode ganz einfach aufschreiben:

FIND_MAX(A):

```
1 max = A[1]
2 for j = 2 to A.length:
3     if A[j] > max
4         max = A[j]
```

Beweis der Korrektheit

Wir werden nun mit Hilfer einer loop-Invariante beweisen, dass dieser Algorithmus korrekt ist.
Die loop-Invariante lautet:

> Zu Beginn jeder Iteration des for-loops (Zeilen 2-4) ist `max` das maximale Element in der Teilliste `A[1...j-1]`.

Wir müssen nun drei Dinge bezüglich der loop-Invariante zeigen:
Initialization
Es stimmt vor der ersten Iteration des loops.
Dies stimmt, denn vor der ersten Iteration besteht die Teilliste `A[1...j-1]` lediglich aus dem einzigen Element `A[1]`. Das einzige Element in einer Liste ist trivialerweise das Maximum dieser Liste.
Maintenance
Wenn es vor der Iteration stimmt, dann stimmt es auch nach der Iteration.
Dies trifft hier zu. Zu Beginn der Iteration ist `max` das grösste Element in der Teilliste `A[1...j-1]`. Falls nun `A[j]` grösser ist als `max`, dann wird dieser Wert als das neue Maximum gesetzt.
Termination
Wenn der loop endet, dann gibt uns die loop-Invariante eine nützliche Eigenschaft, welche uns erlaubt die Korrektheit des Algorithmus zu beweisen.
Die Bedingung, dass der loop endet ist, dass `j > n`. Damit haben wir aber alle Elemente angeschaut und am Schluss muss `max` gerade der Wert des grössten Elements in der Liste sein.

Laufzeitanalyse

best-case
Wir unterscheiden zwischen der *best-time* Laufzeit und der *worst-time* Laufzeit. Die *best-time* Laufzeit ist die Zeit, welcher der Algorithmus im besten / günstigsten Fall benötigt. Für unser konkreten Problems, ist der günstigste Fall gerade dann, wenn das erste Element in der Liste das Maximum der

ganzen Liste ist. Dadurch wird die Bedingung in Zeile 3 nie erfüllt sein und entsprechend muss die Zeile 4 nie ausgeführt werden.
Nehmen wir an, dass die Kosten der i-ten Zeile c_i betragen, $i = 1, 2, 3, 4$. Dann wären die Gesamtkosten im besten Fall $c_1 + nc_2 + (n-1)c_3$.
worst-case
Die *worst-time* Laufzeit ist die Zeit, welcher der Algorithmus im schlimmsten / ungünstigsten Fall benötigt. Im unserem konkreten Beispiel wäre dies der Fall, wenn die Zeile 4 in jedem loop-Durchgang ausgeführt werden muss. Dies ist genau dann der Fall, wenn die Eingabe (Liste) strikt absteigend sortiert ist. Dann ergäbe sich eine Laufzeit von $c_1 + nc_2 + (n-1)c_3 + (n-1)c_4$.

5.2 FIND_MAX

Für eine Untersuchen des Sortier-Algorithmus *Insertion Sort*:

INSERTION_SORT:

```
1  for j = 2 to A.length
2      key = A[j]
3      # fuege A[j] in die sortierte Liste A[1...j-1] ein
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i+1] = A[i]
7          i = i - 1
8      A[i+1] = key
```

verweisen wir auf das Kapitel 2.1 im Buch *Introduction to Algorithms 3rd Edition*:
https://tinyurl.com/yc6qo9om.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] Volker Claus und Andreas Schwill. *Duden Informatik A-Z: Fachlexikon für Studium, Ausbildung und Beruf*, volume 4. Bibliographisches Institut & F. A. Brockhaus, 2006. 3

Anhang A

Installation von Python

Selbstverständlich ist jeder Schüler in der Wahl seiner Python-Distribution und des Texteditors völlig frei.
Wir empfehlen die *Anaconda-Distribution*[1] zu verwenden. Diese lässt sich auf *Microsoft Windows*, *macOS* sowie auf den meisten gängigen *GNU/Linux Distributionen* auf einfache Weise installieren.
Anaconda ist eine äusserst moderne und umfangreiche Python-Distribution, die alle Werkzeuge zur Verfügung stellt, welche wir benötigen werden (und noch sehr viele mehr). Im Umfang der Anaconda-Distribution sind unter anderem zahlreiche nützliche Software-Pakete (ein paar davon werden wir benötigen), die sehr übersichtliche integrierte Entwicklungsumgebung (integrated development environment, Abkürzung: IDE) *spyder*, sowie das *Jupyter Notebook* enthalten.
Im Folgenden wird beschrieben, wie sich Anaconda auf den entsprechenden Betriebssystemen installieren und starten lässt.
Sollte man sich für eine andere Distribution als Anaconda entschieden haben, ist man für deren Installation und Verwaltung selbst verantwortlich. Im Internet lassen sich häufig nützliche Installationsanleitungen finden.

Installation von Anaconda für Windows OS

Öffne `https://www.anaconda.com/download/#windows` im Browser und lade die neueste Version von Python herunter (z.B. Python 3.6 oder höher). Die heruntergeladene Datei wird vermutlich von der Form `Anaconda3-VERSIONSNUMMER-Windows-x86_64.exe` sein. Dabei steht 'VERSIONSNUMMER' stellvertretend für diejenige Versionsnummer, welche zum Zeitpunkt des Downloads aktuell ist. Diese .exe-Datei kann durch einen

[1] offizielle Webseite: `https://www.anaconda.com/`

Doppelklick aufgerufen werden. Die Installation ist nun selbsterklärend (wähle überall die empfohlenen Einstellungen). Nach abgeschlossener Installation kann man über die Windows-Suchfunktion (auf dem Desktop unten links über 'Start' z.B.) nach 'Anaconda-Navigator' suchen und diesen aufrufen. Falls der Anaconda-Navigator erfolgreich gestartet ist, kann man von dort aus beispielsweise *spyder* öffnen.
Alternativ kann man auch den Instruktionen unter
https://docs.anaconda.com/anaconda/install/windows
folgen oder sich das YouTube-Video
https://www.youtube.com/watch?v=YB2zOi3Hd0I
anschauen.

Installation von Anaconda für macOS

Öffne https://www.anaconda.com/download/#macos im Browser und lade die neueste Version von Python herunter (z.B. Python 3.6 oder höher). Die heruntergeladene Datei wird vermutlich von der Form
`Anaconda3-VERSIONSNUMMER-MacOSX-x86_64.bin` sein. Dabei steht 'VERSIONSNUMMER' stellvertretend für diejenige Versionsnummer, welche zum Zeitpunkt des Downloads aktuell ist. Diese .bin-Datei kann durch einen Doppelklick aufgerufen werden. Die Installation ist nun selbsterklärend (wähle überall die empfohlenen Einstellungen). Nach abgeschlossener Installation kann man über die macOS-Suchfunktion (*Spotlight*) nach 'Anaconda-Navigator' suchen und diesen aufrufen. Falls der Anaconda-Navigator erfolgreich gestartet ist, kann man von dort aus beispielsweise *spyder* öffnen.
Alternativ kann man auch den Instruktionen unter
https://docs.anaconda.com/anaconda/install/mac-os
folgen oder sich das YouTube-Video
https://www.youtube.com/watch?v=EFksZiYva5k
anschauen.

Installation von Anaconda für GNU/Linux

Öffne https://www.anaconda.com/download/#linux im Browser und lade die neueste Version von Python herunter (z.B. Python 3.6 oder höher). Die heruntergeladene Datei wird vermutlich von der Form
`Anaconda3-VERSIONSNUMMER-Linux-x86_64.sh` sein. Dabei steht 'VERSIONSNUMMER' stellvertretend für diejenige Versionsnummer, welche zum Zeitpunkt des Downloads aktuell ist. Navigiere nun im Terminal (Shell), mit

Hilfe des Befehls `cd` (change directory), in den Ordner, in welchem sich die heruntergeladene Datei befindet. Führe dort im Terminal den Befehl

```
bash Anaconda3-VERSIONSNUMMER-Linux-x86_64.sh
```

aus, wobei natürlich 'VERSIONSNUMMER' durch die tatsächliche Versionsnummer der Datei ersetzt werden muss. Der komplette Befehl könnte also z.B. so lauten: `bash Anaconda3-5.1.0-Linux-x86_64.sh`. Dies startet den Installationsvorgang. Die Installation selbst ist nun selbsterklärend (drücke die `ENTER`-Taste um nach unten zu scrollen, wähle überall die empfohlenen Einstellungen und tippe, falls erforderlich, stets 'yes').
Führe nun den Befehl

```
source /.bashrc
```

im Terminal aus. Von jetzt an kann der Anaconda-Navigator stets ganz bequem über den Terminal-Befehl

```
anaconda-navigator
```

gestartet werden (die Vorherigen Schritte müssen natürlich nicht nochmals wiederholt werden). Falls der Anaconda-Navigator erfolgreich gestartet ist, kann man von dort aus beispielsweise *spyder* öffnen. Alternativ kann man auch den Instruktionen unter
https://docs.anaconda.com/anaconda/install/linux
folgen oder sich das YouTube-Video
https://www.youtube.com/watch?v=mjtkTbtakkM
anschauen.