GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Vorsemesterkurs

Quick-Start Informatik

4. Oktober 2011

Prof. Dr. Isolde Adler
Dipl. Inf. Carsten Heep
Dipl. Math. Felix Salfelder

Institut für Informatik
Goethe Universität Frankfurt

Vorsemesterkurs Quick-Start Informatik 2010

Idee und Konzeption: Isolde Adler, mit Unterstützung von Carla Cederbaum, Ronja Düffel, Carsten Heep, Sandra Kiefer, Grzegorz Lato, Roman Lossa, Jürgen Poloczek, Kai-Marian Pukall, Pavel Safre, Johannes Seitz

Vorlesung: Sandra Kiefer, Roman Lossa, Kai-Marian Pukall, Johannes Seitz

Übungsleitung: Moritz Jäger, Sandra Kiefer, Grzegorz Lato, Roman Lossa, Linda Luy, Kai-Marian Pukall, Pavel Safre, Johannes Seitz, mit Unterstützung von Ronja Düffel und Felix Salfelder

Organisation: Isolde Adler, Ronja Düffel

Schulung der Tutorinnen und Tutoren (Konzept und Durchführung): Isolde Adler, Carla Cederbaum

Skript (Praktische Informatik): Isolde Adler, Carsten Heep, Felix Salfelder

Skript (Theoretische Informatik): Isolde Adler

Übungsaufgaben: Isolde Adler, Sandra Kiefer, Roman Lossa, Linda Luy, Kai-Marian Pukall, Pavel Safre, Felix Salfelder, Johannes Seitz

Druck und Technik: Rechnerbetriebsgruppe Informatik

Liebe Teilnehmerinnen und Teilnehmer,

der Vorsemesterkurs Quick-Start Informatik findet 2010 zum ersten Mal statt. Ziel ist es, den Übergang von der Schule zum Informatikstudium zu erleichtern. Was ist Informatik? Muss ich ein Programmierfreak sein? Muss ich ein Mathe-Ass sein? Wie sieht das Studium eigentlich aus? Was sollte ich können? Wo finde ich Antwort auf meine Fragen? Wie ist das Leben an der Uni? Wer sind meine Mitstudierenden?

Um diese und weitere Fragen zu beantworten, entstand die Idee, einen Vorsemesterkurs Informatik anzubieten. Der Kurs richtet sich in erster Linie an Studienanfängerinnen und -anfänger, aber auch an interessierte Schülerinnen und Schüler und an Studierende nach dem ersten Studienjahr, die nochmal die Grundlagen wiederholen möchten. Er besteht aus einer Einführung in das Programmieren mit python und einer Einführung in grundlegende Arbeitsweisen der theoretischen Informatik. In Vorlesungen und praktischen Übungen habt Ihr die Möglichkeit, den Stoff kennenzulernen und einzuüben. Es werden keine fachspezifischen Kenntnisse vorausgesetzt. Die Aufgaben in verschiedenen Schwierigkeitsgraden ermöglichen Euch, je nach Vorwissen, Lerntempo und Ausdauer, Neues dazuzulernen. Die langen Mittagspausen und das Wochenende geben Raum, sich selbständig mit den Aufgaben zu beschäftigen. Wer über Mittag allein oder in Gruppen arbeiten möchte, kann dies gerne in den Räumlichkeiten des Lernzentrums oder in den Fischerräumen tun. Für Fragen sind auch Ansprechpartner vor Ort.

Die Aufteilung in Vorlesung, Übungen und selbständiges Nach- und Vorarbeiten ist typisch für das Informatikstudium. Im Gegensatz zum Studium wird der Kurs aber ausschließlich von Studierenden höherer Semester geleitet. Zudem seid Ihr freiwillig hier, und es gibt keinerlei Leistungsdruck. Hier habt Ihr die Möglichkeit, einfach mal auszuprobieren, und gemäß Eurem eigenen Kenntnisstand dazuzulernen. Der Inhalt des Skripts wird nicht zu Studienbeginn vorausgesetzt. Die Themen werden Euch aber im ersten Jahr wieder begegnen. Dass Ihr ganz unterschiedliche Vorkenntnisse habt, liegt an Euren unterschiedlichen Werdegängen und ist völlig normal. Man kann das Informatikstudium auch mit wenig Vorkenntnissen meistern. Allerdings muss dann viel Zeit in sorgfältige Vor- und Nacharbeitung der Vorlesungen investieren. Wichtig ist, dass Ihr lernt, Fragen zu stellen. Wer konkrete Fragen formulieren kann, ist schon weit gekommen. Für inhaltliche Fragen zu den Themen des Vorsemesterkurses stellt die Fachschaft Informatik das Online-Forum `http://fsinf-forum.de/` zur Verfügung. Zudem gibt es eine Mailing-Liste, an die Ihr Eure Fragen schicken könnt. Auf der Mailingliste stehen Eure Tutorinnen und Tutoren, die dann Eure Fragen beantworten.
Hilfreich ist es auch, in Gruppen zu arbeiten. Gemeinsam weiß und sieht man mehr als allein. Und wenn man etwas erklären kann, dann hat man es verstanden. Für Gruppenarbeit steht Euch auch während des Studiums das Lernzentrum zur Verfügung. Bei Fragen zu Vorlesungsinhalten könnt Ihr Euch dort an Frau Dipl.-Inf. Ronja Düffel wenden.

Last but not least habt Ihr die Möglichkeit, in den Fachschaftsräumen Mitstudierende

kennenzulernen, alles über das Studium zu erfahren, selber Eure Ideen aktiv einzubringen, Spiele zu spielen, Kaffee und Tee zu trinken, oder einfach eine Pause zu machen und zu entspannen.

Wir wünschen Euch einen schönen Vorsemesterkurs und einen erfolgreichen Start ins Studium!

Euer Orga-Team

P.S.: Wenn es Euch gefallen hat oder wenn Ihr Eure eigenen Ideen mit einbringen möchtet, seid Ihr herzlich eingeladen, Euch in Eurem Studium zukünftig auch bei der Organisation und Durchführung des Vorkurses zu engagieren!

Dank

An der Konzeption und Umsetzung des Vorsemesterkurses Quick-Start Informatik waren das Institut für Informatik (insbesondere das Ingo-Wegener-Lernzentrum), das Institut für Didaktik der Mathematik und der Informatik und die Fachschaft Informatik, sowie die Trainerin Carla Cederbaum beteiligt. Ich möchte allen für ihr herausragendes Engagement, die zahlreichen Ideen, Anregungen und Beiträge danken – für einen Einsatz, der weit über dem gewöhnlichen Maß lag: Ronja Düffel hat den Kurs mit konzipiert, koordiniert und umgesetzt – von der Vorbereitung bis zur Nachbereitung liefen und laufen bei ihr die Fäden zusammen. Carla Cederbaum hat mit zahlreichen Anregungen und ihrer Erfahrung zum Konzept des Vorsemesterkurses beigetragen und die Tutorenschulung mit entworfen und geleitet. Carsten Heep hat den Praxisteil des Skripts entworfen, Felix Salfelder hat geholfen, ihn auszuarbeiten und ihn ergänzt. Sandra Kiefer, Grzegorz Lato, Roman Lossa, Linda Luy, Kai-Marian Pukall, Pavel Safre und Johannes Seitz haben zusätzlich zu ihrer Tutorentätigkeit im Rahmen des Vorkurses an der Themenauswahl, der Aufgabenerstellung und Vorbereitung des Kurses mitgewirkt. Zudem haben sie viele wichtige Hinweise zu Wünschen und Schwierigkeiten von Studienanfängerinnen und -anfängern gegeben, zusammen mit Ideen, wie man diesen am besten begegnet. Jürgen Poloczek hat uns mit Ratschlägen und Ideen, insbesondere zur didaktischen Umsetzung, und mit seinem Kontakt zu den Schulen stets beiseite gestanden. Manfred Schmidt-Schauß und Georg Schnitger haben uns als Studiendekane den Weg geebnet und mit ihrer Unterstützung und ihrem Rückhalt den Kurs in dieser Form erst möglich gemacht. Sandra Kiefer und Philipp Krause haben zahlreiche wertvolle Tipps und Korrekturen zum Skript beigetragen. Philipp Krause hat die Geschichte vom „kleinen Gauß“ recherchiert und uns technisch unterstützt. Das RBI hat uns technisch unterstützt und das Kursmaterial gedruckt. und Die Fachschaft Mathematik der Albert-Ludwigs-Universität Freiburg hat freundlicherweise den Aufgabenfundus der Grundlagenübung zur Verfügung gestellt.

Ich denke, dass wir hier ein solides Konzept entwickelt und ein gutes Fundament für die kommenden Jahre gelegt haben. Allen Beteiligten gilt mein allerherzlichster Dank für die konstruktive und bereichernde Zusammenarbeit!

Frankfurt am Main, den 22.9.2010,

Isolde Adler

Inhaltsverzeichnis

Teil I.

Praktische Informatik

1. Einleitung

1.1. Unix

Unix ist ein Mehrbenutzer-Betriebssystem, das ursprünglich 1969 in den *Bell Labs* entwickelt wurde. Der Stammbaum der Unix-Derivate ist riesig und Unices bilden den Kern der meisten modernen Betriebssysteme (`www.levenez.com/unix`). Linux ist ein bekanntes Beispiel. Gegenstand dieses Kapitels ist jedoch nicht eine geschichtliche Abhandlung, vielmehr geht es darum, elementare Unix-Befehle vorzustellen und erste Schritte im Umgang mit Unix zu vermitteln. Im Folgenden wird stets auf das System der **R**echner**b**etriebsgruppe[1] des Instituts für **I**nformatik der Goethe-Universität Frankfurt (RBI) Bezug genommen.

1.1.1. Terminals

Unix stellt seinen Benutzern sogenannte *Terminals* bereit, an denen gearbeitet werden kann. Ein Terminal ist die Schnittstelle zwischen Mensch und Maschine. Es gibt zwei grundlegend verschiedene Arten von Terminals: Die *textbasierten* Terminals präsentieren dem Benutzer einen Bildschirm[2], der im Wesentlichen Buchstaben und andere Zeichen in einem festen Raster anzeigt. Über diese Terminals ist es möglich, mit Programmen, die über ein CLI (*command line interface*) verfügen, zu interagieren. Dies sind oft Programme, die Eingaben in Textform entgegennehmen.

Neben dem CLI gibt es *graphische* Terminals, die dem Benutzer, von technischen Einschränkungen einmal abgesehen, beliebige bunte, dreidimensionale und animierte Steuerelemente und Interaktionsobjekte präsentieren. Die Schnittstelle eines Programms, die auf derartige Technik zurückgreift, wird mit GUI (*graphical user interface*) bezeichnet. Graphische Terminals werden oft zur Anzeige von graphischen Benutzeroberflächen verwendet, deren Optik mitunter an Software aus dem Hause Apple oder Microsoft erinnert. Auf den Rechnern des RBI findet man unter anderem „GNOME“ und „KDE“.

Ein einzelner Rechner stellt sieben voneinander unabhängige Terminals zur Verfügung. Mit den Tastenkombinationen Ctrl + Alt + F1, ..., Ctrl + Alt + F2 bis Ctrl + Alt + F7 lässt sich bestimmen, welches davon auf dem Bildschirm angezeigt werden soll. An das angezeigte Terminal werden dann auch Tastatureingaben weitergeleitet.

[1] `http://www-intern.rbi.informatik.uni-frankfurt.de/rbi/linux-tutorials/`
[2] Anfangs wurden stattdessen Drucker mit Endlospapier verwendet.

1.1.2. Login und Shell

Bevor die Arbeit beginnen kann, muss sich ein Benutzer an einem Terminal anmelden. Dies erfolgt durch Eingabe eines Benutzernamens und des zugehörigen Passwortes. Dieser Vorgang nennt sich *Einloggen* (engl.: *login*). Findet das Einloggen an einem textbasierten Terminal statt, so übernimmt eine (Unix-)*Shell* die Kontrolle über das Terminal. Diese macht sich bemerkbar durch eine *Eingabeaufforderung* (engl.: *prompt*), eine Zeile, die Statusinformationen enthält und mit einem blinkenden *Cursor* endet: „`[benutzername]@[rechnername] >`“. Hat sich der Benutzer an einem graphischen Terminal angemeldet, wird er zuerst ein virtuelles textbasiertes Terminal starten müssen, um eine Shell zu Gesicht zu bekommen. Ein virtuelles Terminal kann über das Kontextmenü gestartet werden[3].

Das Prompt signalisiert die Bereitschaft der Shell, (Unix-)Befehle entgegenzunehmen. Befehle sind nach bestimmten Regeln zusammengesetzte Wort- und Zeichenkombinationen. Diese können einfach eingetippt werden. Nach Betätigen der *Eingabetaste* (engl.: *return*, ) wird der Befehl verarbeitet. Im gedruckten Text notieren wir die Eingabe der drei Zeichen `a`, `b` und `c` mit darauffolgender Betätigung der Eingabetaste kurz als **`abc`**. Sonstigen Text, der auf dem Bildschirm steht oder stehen kann, notieren wir in `Schreibmaschinenschrift`.

Ohne das Regelwerk weiter zu studieren, werden nachstehend einige Befehle vorgestellt, die einen Einblick in die Funktionsweise der Shell geben. Um Tipparbeit und Tastaturverschleiß zu vermeiden, fallen die Namen der Befehle oft sehr kurz aus. Meist verbirgt sich jedoch eine Abkürzung oder raffinierte Mnemotechnik dahinter.

- **`echo Hallo Welt`** veranlasst die Shell ‚Hallo Welt‘ auszugeben. Wir sagen „**`echo Hallo Welt`** gibt den Text ‚Hallo Welt‘ aus.“
- **`whoami`** und **`hostname`** sagt uns, wer bzw. wo wir sind.
- **`pwd`** (*print working directory*): gibt den Pfad des Verzeichnisses aus, in dem sich der Benutzer gerade befinden. Dies nennt man oft auch das *aktuelle Verzeichnis*. Unmittelbar nach dem Login ist dies das *Homeverzeichnis* des entsprechenden Benutzers.
- **`mkdir dinge`** (*make directory*): legt ein Verzeichnis mit dem Namen `dinge` an, falls ein solches noch nicht existiert.
- **`cd dinge`**: wechselt in das eben erstellte Verzeichnis.
- **`cd ..`** bringt uns wieder zurueck.
- **`ls`** (*list*) zeigt eine Liste der Namen der im aktuellen Verzeichnis befindlichen Dateien und Verzeichnisse an, die nicht mit einem Punkt (`.`) anfangen.

  `ls -a` lässt Namen, die mit `.` anfangen, nicht aus.

  `ls -a dinge` zeigt den Inhalt des eben erstellten Verzeichnisses an.

[3] Mit der rechten Maustaste auf den Desktop klicken und den ersten Eintrag (Konsole) auswählen.

- **doof** beschert uns eine ausführliche Fehlermeldung.

Einem Befehl folgen im Allgemeinen weitere, die Ausführung des Programms beeinflussende, Zeichen. Diese bilden die *Argumentliste*. Befindet sich beispielsweise in der Argumentliste von **ls** ein `-a` so werden auch Einträge, die mit `.` beginnen, ausgegeben. Wer mehr über **ls** und seine Argumente lernen will, dem sei der Befehl **man ls** nahegelegt (mit q beenden). Die Abkürzung `man` steht für „Handbuch“ (engl.: manual). Sie steht für alle wichtigen Befehle zur Verfügung, so auch für `man` selbst (**man man**).

1.1.3. Dateien und Verzeichnisse

Eines der grundlegenden Unix-Paradigmen lautet „everything is a file“. Die *Studentin*/ der *Student* wird die Gelegenheit haben, diese Aussage in voller Breite zu deuten und die Vor- und Nachteile dieses Ansatzes zu verstehen. Dies soll als Motivation dienen, den Unix-Dateibaum und einfache Dateioperationen kennenzulernen.

Jedes Unix-System[4] verwaltet einen Dateibaum. Der Dateibaum ist ein virtuelles Gebilde zur Datenverwaltung, ähnlich einem gefüllten Aktenschrank. Die Bausteine heißen *Dateien* (engl.: *file*). Die typische Datei enthält zunächst *Daten*, das ist oft Text, manchmal sind es auch Bilder oder Maschinenanweisungen. Ein Aktenschrank enthält Ordner, die Ordner Seiten, die Seiten Zeilen und die Zeilen Buchstaben. Unser Dateisystem sieht bestimmte Dateien vor, sogenannte *Verzeichnisse* (engl.: *directories*), die selbst Dateien enthalten können. Eine Datei hat immer einen *Namen*, das ist eine nichtleere Zeichenkette, die einem bestimmten Regelwerk unterworfen ist. Man stelle sich unter einem Namen dasselbe vor, wie unter dem Namen einer Stadt oder eines Haustieres. Man beachte, dass Unix zwischen Groß- und Kleinschreibung unterscheidet.

Jede Datei, insbesondere jedes Verzeichnis, befindet sich in einem Verzeichnis, dem *übergeordneten* Verzeichnis (engl.: *parent directory*). Nur das *Wurzelverzeichnis* (engl.: *root directory*) ist in sich selbst enthalten. Es trägt den eindeutigen Namen `/`. Zwei Dateien sind dieselben, genau dann wenn sie gleiche Namen tragen *und* im selben Verzeichnis enthalten sind. Aber wo ist der Baum? Zur Veranschaulichung hilft ein kleines Beispiel. Das Wurzelverzeichnis `/` enthalte zwei Verzeichnisse `A` und `B`. Beide Verzeichnisse enthalten je ein Verzeichnis mit dem Namen `1` und `2`. In dem Schema aus Abbildung 1.1 ist unschwer der Baum zu erkennen.

Mit dieser Erkenntnis lassen sich die beiden Verzeichnisse mit dem Namen `1` leicht auseinander halten – und zwar anhand ihres (*absoluten*) *Pfades*. Der Pfad des in `A` enthaltenen Verzeichnisses wird als `/A/1` notiert, der des anderen als `/B/1`. Der Schrägstrich `/` (engl.: slash) dient als Trennzeichen. *Relative* Pfade sind Pfade, die nicht mit dem Wurzelverzeichnis `/` anfangen. Ist das aktuelle Verzeichnis `/B`, so können mit `1` und `2` die Unterverzeichnisse eindeutig adressiert werden. Mit dem Symbol `..` für das übergeordnete Verzeichnis kann dann für `/` auch `..`, für `/A` auch `../A` oder für `/A/1` auch `../A/1` geschrieben werden. Relative Pfade beginnen oft mit `./`, dieser Punkt ist jedoch nur ein Symbol für „hier“ und

[4] Im Sinne eines Computers, auf dem Unix läuft.

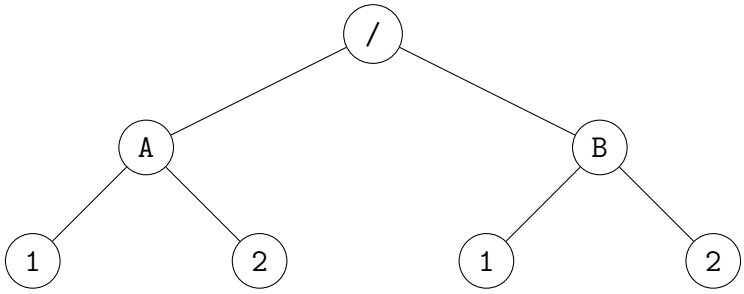


Abbildung 1.1.: Der Verzeichnisbaum der Erkenntnis

wird nur zur Verdeutlichung vorangestellt. Ebenso dienen zusätzliche `/` Trennzeichen nur der Optik; ist das aktuelle Verzeichnis etwa `/A`, so verweisen die folgenden Pfade alle auf dasselbe: `1`, `./../A/1`, `/../../A/1//`, `/A/../A/1` sind gleichbedeutend mit `/A/1`. Pfade zu Dateien, die keine Verzeichnisse sind, funktionieren natürlich ebenso, mit dem Unterschied, dass sie nicht in einem Verzeichnis enden.

1.1.4. Das Homeverzeichnis

Das Verzeichnis, um das sich alles dreht, ist das *Homeverzeichnis*. Dies ist das persönliche Verzeichnis des Benutzers, in dem dieser befugt ist, Dateien abzulegen oder zu verändern. Ebenso gibt es Dateien, die ein „gewöhnlicher“ Benutzer nicht verändern oder gar einsehen kann, etwa die Dateien anderer Benutzer oder Systemdateien. Das eigene Homeverzeichnis ist systemweit unter dem Pfad `/home/users4/ommz/<benutzername>` aufzufinden. Kürzel hierfür sind `~` und `$HOME`. Auch ist etwa `~musterfrau` ein Kürzel für das Homeverzeichnis von Melanie Musterfrau.

1.1.5. Editieren und Textdateien

Bislang ging es um Dateien, Verzeichnisse und das allgemeine Bedienen einer Shell. Im Fokus dieses Abschnittes stehen die Textdateien. Diese werden für uns relevant, da sie unter anderem den *Code* der geschriebenen Programme beherbergen werden. Ein *Texteditor* ist ein Programm, das das Erstellen und Verändern von Textdateien erleichtert.

Ein verbreiteter Texteditor heißt vi (sprich: [vi: ai]). Dieser steht nicht nur in der RBI zur Verfügung, er ist auf fast jedem Unix-System vorhanden. Mit dem Befehl **`vi`** wird der Editor[5] gestartet. Der Befehl **`vi /tmp/irgendeinedatei`** startet vi und öffnet sogleich eine Sicht auf die angegebene Datei. Diese Sicht nennt man *Buffer*, hier findet das Editieren statt. Nach dem Öffnen und nach dem Speichern stimmt der Inhalt des Buffers mit dem der korrespondierenden Datei überein. Unser Editor unterscheidet einige Betriebsmodi, die sich wie folgt beschreiben lassen. Wichtig dabei ist die Position des *Cursors*, auf die sich die meisten Aktionen beziehen.

[5] Oftmals handelt es sich schon um eine verbesserte Variante vim (für vi *improved*). Diese Unterscheidung soll uns hier nicht kümmern.

1. Im *Befehlsmodus* werden Tastatureingaben als Befehle aufgefasst. Unter einem Befehl kann man sich das vorstellen, was man einem Herausgeber (engl.: editor)zurufen würde, bäte man ihn um Änderungen an einem Text. In diesem Modus befindet sich vi nach dem Starten. vi versteht Befehle wie „öffne/speichere diese und jene Datei“ (**`:e diese`**, **`:w jene`**), „Lösche die Zeile, in der sich der Cursor befindet!“ (d , d) oder „Tausche den Buchstaben unterm Cursor durch den folgenden aus!“ (r). Natürlich ist es auch möglich, den Cursor zu bewegen, etwa mit den Pfeiltasten oder mit mannigfachen anderen Tasten, die Bewegungen in alle möglichen Positionen erlauben. Viele Neider halten **`:q!`** für den wichtigsten aller vi-Befehle. Dieser führt jedoch nur zum Programmabbruch (engl.: quit) ohne vorheriges Speichern.

2. Im `EINFÜGEN`- und `ERSETZEN`-*Modus* erscheinen die eingegebenen Zeichen direkt auf dem Bildschirm, im Buffer. Im ersteren wird Text neu hinzugefügt, im zweiteren werden bereits vorhandene Zeichen überschrieben. In diese Modi gelangt man vom Befehlsmodus aus mit den Tasten i bzw. R . Zwischen ihnen kann mit der Ins -Taste hin- und hergewechselt werden. Durch Betätigen der Esc -Taste gelangt man wieder zurück in den Befehlsmodus.

3. Die `VISUELL`en Modi erlauben das Markieren eines Bereiches des Buffers, um sodann Befehle abzusetzen, die sich auf den markierten Bereich beziehen. Befindet man sich im Befehlsmodus, so gelangt man mit der v -Taste in den gewöhnlichen visuellen Modus. Dieser ermöglicht das Markieren eines Textbereiches durch Bewegen des Cursors. Nach Eingabe und Ausführung eines Befehls – etwa x , „markierten Bereich löschen“ – findet man sich im Befehlsmodus wieder. Auch die Esc -Taste führt zurück in den Befehlsmodus.

Wir stellen fest, dass ein Einstieg in vi mit dem Lernen einiger Tastenkürzel und Schlüsselworte einhergehen muss. Das lohnt sich aber für alle, die oft mit Text arbeiten. Hat man einmal die Grammatik der Befehle, die vi akzeptiert, verstanden, so wird das Editieren von Text zum Kinderspiel und geht schnell von der Hand. Wer das Tippen im Zehnfingersystem beherrscht und einzusetzen vermag, weiss schon, dass sich anfänglicher Mehraufwand auszahlen kann.

Ein Schnelleinstieg in vi ist mit Hilfe einer vorbereiteten Anleitung möglich. Diese lässt sich in einer Shell mit dem Befehl **`vimtutor`** starten. In dieser Anleitung sind einige Funktionen des Editors zum sofortigen Ausprobieren aufbereitet worden. Danach empfiehlt es sich, in einer Kurzreferenz zu stöbern.

1.2. Python

„Python is a programming language that lets you work more quickly and integrate your systems more effectively. You can learn to use Python and see almost immediate gains in productivity and lower maintenance costs.“

So urteilen die Autoren von python über ihre Programmiersprache. In der Tat ist python eine Programmiersprache, die es ermöglicht, mit vergleichsweise geringem Lernaufwand Programme zu schreiben. Was python ist, wo man es her bekommt und wie es am besten einzusetzen ist, wird Gegenstand dieses Abschnittes sein.

1.2.1. Was ist Python?

Python ist eine *imperative*, *objektorientierte*, *interpretierte* Programmiersprache mit Einflüssen aus dem Bereich der *funktionalen* Programmierung. Imperatives Programmieren bedeutet, dass Programme aus Anweisungen aufgebaut werden, die nacheinander ausgeführt werden. Die *Objektorientierung* kann als eine Erweiterung des imperativen Programmierkonzepts aufgefasst werden. Das Konzept der Objektorientierung wird in der Veranstaltung *Grundlagen der Programmierung 1* behandelt – hier wollen wir uns auf das Verständnis des imperativen Programmierens beschränken. Wir werden dabei nicht auf die aus der funktionalen Programmierung entlehnten Aspekte von python eingehen. Einen Einblick in funktionale Programmierung vermittelt die Vorlesung *Grundlagen der Programmierung 2*.

Der Begriff *interpretierte* Programmiersprache geht auf die Methode des Ausführens von in derselben geschriebenen Programmen zurück. Um Programme schnell auf einem Computer auszuführen, bereitet man typischerweise die Befehle in der Sprache auf, die der Computer in seinem tiefsten Inneren versteht: die *Maschinensprache*. Diese Aufbereitung nennt man das *Kompilieren*. Stattdessen führt ein *Interpreter*, den man sich wie einen virtuellen, sehr flexiblen Computer vorstellen kann, die in python geschriebenen Befehle instantan aus. Der Interpreter erstellt also kein direkt auf der Maschine ausführbares Programm, sondern analysiert, deutet, und agiert selbst einige Abstraktionsebenen höher. Die langsamere Ausführgeschwindigkeit liegt also auf der Hand. Ein Vorteil besteht darin, dass *interaktiv* programmiert werden kann: Befehle können einzeln an den Interpreter übergeben werden und zwischendurch kann der Zustand des laufenden Programmes beliebig inspiziert werden. Echte Maschinen im Betrieb zu *beobachten* ist oft aufwändiger. (Siehe Abschnitt 1.2.3).

1.2.2. Installation

Die Rechner der RBI sind bereits vorbereitet, und es kann sofort losgehen. Ist man selbst für den Computer verantwortlich, auf dem man arbeiten möchte, ist noch etwas vorzubereiten.

Läuft auf unserem Computer ein Linux-Betriebssystem[6], so sind die benötigten Programme oftmals bereits installiert. Falls nicht, lassen sie sich je nach *Distribution* mit der mitgelieferten *Paketverwaltung*, etwa durch einen Aufruf[7] von **`apt-get install python`**, **`pacman -S python`**, **`emerge python`** oder **`yast -i python`** schnell nachinstallieren. Auf ähnliche Weise gelangt man auch schnell an einen Texteditor. Auf Windows und MacOs ist dies nicht der Fall. Unter `http://www.python.org/download` kann python inklusive eines

[6] Linux ist eine Unix-Variante, auf die unser eben Gelerntes zutrifft.

[7] als autorisierter Benutzer

rudimentären Editors (IDLE) heruntergeladen werden. Mit wenigen Mausklicks läuft also auch hier ein Interpreter. Wer noch Hemmungen hat, ein Linux-System auf seinen Rechner zu spielen, der kann auch mit Fink[8]oder cygwin [9]in den Genuss einer automatischen Paketverwaltung kommen.

1.2.3. Erste Schritte

Die Programmiersprache python lässt sich auf unterschiedliche Weisen benutzen. Die einfachste ist, den Interpreter direkt in einer Shell mit dem Befehl **`python`** im interaktiven Modus zu starten. Der Interpreter signalisiert seine Bereitschaft durch drei spitze Klammern >>>. Neben dem interaktiven Modus gibt es auch die Möglichkeit, ein python-*Skript* in einer Textdatei abzulegen und deren Pfad dem Interpreter als Argument zu übergeben. Namen von Dateien, die python-Skripte enthalten, werden wir stets mit der Endung `.py` versehen[10]. Dies ist eine gängige Konvention, die aber nur von kosmetischer Bedeutung ist. Angenommen, es existiert ein python-Skript mit dem Namen `dummy.py`, so kann dieses mit dem Aufruf **`python dummy.py`** ausgeführt werden. Theoretisch kann es mit dem Programmieren jetzt losgehen. Praktisch ist es sinnvoll, sich vorher einen Texteditor auszusuchen, um Programme auch ablegen zu können. Es gibt Editoren für jeden Geschmack. Die wohl am weitesten verbreiteten heißen vi und emacs. Dennoch wollen wir dem Anfänger nahelegen, die ersten Schritte mit python mit dem *integrated development environment for python* (kurz IDLE) zu gehen. Dieses trägt nicht weit, aber weit genug, um alle hier gesammelten Progamme nachzuempfinden.

1.2.4. IDLE

Das Programm IDLE ist eine reduzierte Umgebung zum Entwickeln von python-Skripten. Es stellt einen interaktiven Modus und einen Editiermodus zur Verfügung. Der interaktive Modus entspricht genau dem Verhalten des interaktiven Modus von python. Innerhalb des Editors wird die wohl am häufigsten genutzte Funktion *Run module* sein, die mit dem Tastenkürzel F5 verknüpft ist. Eine weitere sehr hilfreiche Funktion ist die automatische Vervollständigung. Sie kann zu jeder Zeit mit der Tastenkombination Ctrl + Leertaste aufgerufen werden und zeigt eine Liste mit Vorschlägen zum Vervollständigen an. Angenommen, der Buchstabe `p` wurde geschrieben, dann sind die Vorschläge der IDLE `pow`, `print` und `property`.

1.2.5. Das erste Python-Skript

Als Visitenkarte einer Programmiersprache kann man das in ihr geschriebene Programm mit dem Titel *Hello World* verstehen. Dieses Programm besteht meist aus wenigen Zei-

[8]„The Fink project wants to bring the full world of Unix Open Source software to Darwin and Mac OS X.“

[9]„Cygwin is a Linux-like environment for Windows.“

[10]Falls sie noch nicht mit `.py` enden.

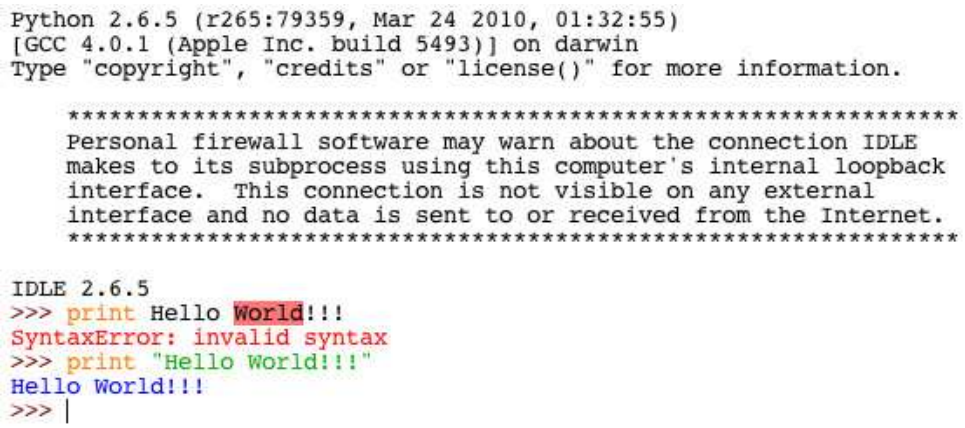


Abbildung 1.2.: Das *integrated development environment for* python im interaktiven Modus. Hier wurden Zwei Befehle ausgeführt (der bunte Text).

chen und seine Aufgabe ist nichts anderes, als nacheinander die Zeichen `H`, `e`, `l`, `l` usw, oft gefolgt von einem Ausrufezeichen, auf einem Bildschirm oder mittels eines Druckers oder einer Nähmaschine o. Ä. zu visualisieren. Dieses Programm kann einen Einblick in die *Syntax* der verwendeten Programmiersprache vermitteln, oder aber als Prüfstein für den Compiler oder Interpreter herhalten. Dies wollen wir jetzt anhand von python durchführen. Der python-Befehl, der für die Ausgabe verantwortlich ist, lautet `print`, obgleich wir die Ausgabe auf einem Bildschirm erhalten werden. Ähnlich wie Unix-Befehle, ist diesem Befehl über ein Argument mitzuteilen, was er zu tun hat. Ohne Argumente gibt `print` nur eine leere Zeile aus, und auch sonst alles, was wir ihm als Argumente „geben“. Da python Buchstaben zunächst als Teile von bedeutungstragenden *Namen* verstehen möchte, müssen wir, wann immer wir ein Zeichen buchstäblich meinen, dieses in Anführungszeichen einfassen. python unterscheidet nicht zwischen `"` und `'`, jedoch muss man sich jeweils für eine Sorte entscheiden.

Mit der Information, dass Argumente vom Befehl und untereinander schlicht mit Leerzeichen getrennt werden, können wir bereits Zeile 1 in Listing 1.1 als Hello-World-Programm identifizieren.

```
1 print("H" "e" 'l' "l" "o" " " " 'W' 'o' 'r' "l" "d" "!")
```

Listing 1.1: Hello World: Variante 1

Hat man noch im Hinterkopf, wie toll python eigentlich sein soll, so ahnt man schon, dass man dies auch lesbarer formulieren kann – wie in Listing 1.2.

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

Listing 1.2: Hello World: Variante 2

Zusammenfassend können wir dem Hello-World-Programm also entnehmen, dass python den Befehl `print` versteht und dass Anführungszeichen eine syntaktische Bedeutung haben.

Wer sich noch an 1.1.2 erinnern kann, der hat jetzt schon zwei Hello-World-Programmme gesehen.

2. Elementare Datentypen

Aus der Mathematik ist uns der Begriff der *Variablen* bereits bekannt. Dort werden Variablen im Sinne von Platzhaltern für unbekannte Werte verwendet. Anstelle eines festen Zahlenwerts werden etwa Symbole wie x, a, λ oder ϵ geschrieben. Durch die Verwendung von Buchstaben als Platzhalter lassen sich Gleichungen formulieren. Werte, die sich anstelle der Variablen einsetzen lassen, so dass eine Gleichung erfüllt ist, heißen Lösungen dieser Gleichung. Vor dem Lösen der Gleichung sind diese Werte unbekannt.

Im Gegensatz hierzu bezeichnet eine Variable in der Informatik, genauer, im Kontext der Programmierung, einen bestimmten Speicherbereich, in dem Informationen abgelegt werden können. Ein solcher Speicherbereich kann sowohl *gelesen* als auch *beschrieben* werden. Eine Variable hat also einen *Namen* (auch *Bezeichner*) und einen *Wert*. Möchte man einen Speicherbereich, der einen Namen trägt, mit Daten füllen, so ist es angesichts der Tatsache, dass ein Computer nur Wörter aus Nullen und Einsen speichern kann, zweckmäßig, festzulegen, wie diese Wörter zu interpretieren sind. Es wird also noch der *Typ* der Variablen festgelegt. Von technischen Einschränkungen abgesehen, kommen als Typ *Ganzzahl* (engl.: *integer*), *Gleitkommazahlen* (engl.: *float*), oder auch *Zeichenkette* (engl.: *string*) in Frage. Ebenso ist es möglich, Datentypen selbst festzulegen. Es lässt sich also das in Abbildung 2.1 gegebene Schema zur Beschreibung einer Variable festhalten.

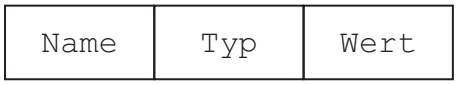


Abbildung 2.1.: Schematische Darstellung einer Variable als *Tripel* bestehend aus den Komponenten *Name*, *Typ* und *Wert*

Wie bei vielen interpretierten Programmiersprachen wird bei python die Typisierung *implizit* vorgenommen. Das bedeutet, dass der Interpreter – nicht der Programmierer – zu entscheiden hat, welchen Typ eine Variable innehaben soll. Auch muss der python-Programmierer dem Interpreter die Namen von Variablen nicht ankündigen. Ein Name wird zum Namen einer Variablen, sobald eine *Zuweisung* stattfindet, das ist eine Zeile, die dem Schema in Listing 2.1 entspricht.

```
1 <Name> = <Ausdruck>
```

Listing 2.1: Schematische Darstellung zum Erzeugen einer Variable in python

Man beachte, dass wir Platzhalter in spitze Klammern setzen. Diesen Vorgang nennt man in python auch die *Erzeugung* einer Variablen. Wird hier ein Name verwandt, der

bereits vergeben wurde, so wird python – falls notwendig – eine weitere Variable vom entsprechenden Typ erzeugen. Dieser neuen Variablen wird sodann der Name übertragen. Dies findet im Programm in Listing 2.2 statt.

```
1 X = "a"
2 X = 1
```

Listing 2.2: Recycling von Namen

Aus Listing 2.1 wird ersichtlich, dass das aus der Mathematik bekannte Gleichheitszeichen zweckentfremdet wird. Der Interpreter liest von rechts nach links: *Zuerst* wird der Ausdruck ausgewertet, *dann* wird ein geeigneter Speicherbereich mit dem Namen und dem erforderten Typen versehen und beschrieben. Der Wert einer Variablen wird ausgelesen und verarbeitet, sobald ihr Name in einem Ausdruck (wie in der zweiten Zeile von Listing 2.3) auftaucht. Man beachte, dass Namen von Variablen ähnlich wie Namen von Dateien in 1.1.3 nicht eindeutig zu sein brauchen. Der Rolle, die bei der Verwaltung von Dateien die Verzeichnisse spielen, entspricht bei der Programmierung die der *Gültigkeitsbereiche* (engl.: *scopes*). Wir werden hierauf in 4.4 kurz eingehen.

```
1 a = b
2 c = a
```

Listing 2.3: Auslesen einer Variablen in python

Im Folgenden werden *built-in*-Datentypen, das sind die Typen, die in python fest integriert sind[1], und zugehörige Operationen vorgestellt.

2.1. Boolean

Die kleinste informationstragende Einheit ist das *Bit*. Es kann genau zwei Zustände annehmen. Je nach Kontext werden diese Zustände *1* und *0*, *Wahr* und *Falsch*, oder *High* und *Low* oder ähnlich genannt. In python werden sie mit den Konstantensymbolen `True` und `False` verkörpert und sind vom Typ `bool`. Die Erzeugung zweier boolescher Variablen (`die_Sonne_scheint`, `es_regnet`) ist in Listing 2.4 gegeben.

```
1 die_Sonne_scheint = True
2 es_regnet = False
```

Listing 2.4: Erzeugung zweier Variablen vom Typ `bool`

2.1.1. Konjunktion und Disjunktion

Die *Konjunktion* (der *Und-Operator*) verknüpft zwei boolesche Variablen – z.B. `A` und `B` – miteinander nach der Regel „Das Ergebnis hat den booleschen Wert `True` genau dann,

[1] im Gegensatz zu Datentypen, die durch Erweiterungen bereitgestellt werden

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

Abbildung 2.2.: Wahrheitstafel der Konjunktion (`and`)

	A==False	A==True
B==False	False	False
B==True	False	True

Abbildung 2.3.: Wahrheitstafel der Konjunktion

wenn beide *Operanden* – die Argumente für den Operator – ebenfalls den booleschen Wert `True` haben“. In python notiert man die Konjunktion der Werte von `A` und `B` als `A and B`. Einen Operator, der auf diese Weise Werte miteinander verknüpft, nennt man *binären Infixoperator*: „Binär“, da er genau zwei Argumente nimmt, und „Infix-“, weil er zwischen die Argumente geschrieben wird.

Mit Hilfe einer *Wahrheitstafel* kann der Wert des Ausdruckes illustriert oder festgelegt werden. Hier wird eine vollständige Liste der möglichen Belegungen in einer Tabelle aufgeschrieben, inklusive eine Spalte für den Wert der logischen Verknüpfung von `A` und `B` (siehe auch Abbildung 2.2). Eine weitere Variante einer Wahrheitstafel zeigt die Abbildung 2.3.

```
1 die_Sonne_scheint and es_regnet
```

Listing 2.5: Konjunktion zweier Variablen vom Typ `bool`

Die *Disjunktion* (der *Oder-Operator*) verknüpft boolesche Variablen miteinander nach der Regel „Die Disjunktion hat den booleschen Wert `True` genau dann, wenn einer der Operanden den booleschen Wert `True` hat“. In python notiert man die Disjunktion der Werte von `A` und `B` als `A or B`. Abbildung 2.4 zeigt hierfür eine Wahrheitstafel.

A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

Abbildung 2.4.: Wahrheitstafel der Disjunktion (`or`)

```
1 die_Sonne_scheint or es_regnet
```

Listing 2.6: Disjunktion zweier Variablen vom Typ `bool`

2.1.2. Negation

Die beiden Werte *Wahr* und *Falsch* gehen durch *Negation* ineinander über. Die Negation wird in python durch das Schlüsselwort `not` realisiert. Sie negiert den booleschen Wert des Ausdruckes, auf den sie angewandt wird. Das folgende Code-Fragment verwendet die beiden oben definierten Variablen `die_Sonne_scheint` und `es_regnet` und weist sie einer neuen Variablen zu und negiert diese schließlich.

```
1 regenbogen_moeglich = die_Sonne_scheint and es_regnet
2 regenbogen_unmoeglich = not regenbogen_moeglich
```

Listing 2.7: Verknüpfung und Negation boolescher Variablen und Literale

Das `not` bezieht sich nur auf einen einzigen Operanden und wird deshalb als *unärer* Operator bezeichnet.

2.2. Integer

Die Sprache python stellt den elementaren Datentyp `int` zur Verfügung, der es erlaubt, ganze Zahlen (engl.: *integers*) zu verwalten. Die Zuweisung einer ganzen Zahl folgt den Regeln, die in Listing 2.1 veranschaulicht werden.

```
1 meaning_of_life = 42
```

Listing 2.8: Zuweisung einer ganzen Zahl

Für den Datentyp `int` stehen die Infixoperatoren `+`, `-`, `*`, `//` und `%` zur Verfügung. Angewandt auf zwei Ausdrücke vom Typ `int` ergeben sie wieder einen diesen Typs. Zur Vorrangregulierung verwendet python die runden Klammern `()`. Hinzu kommen noch einige weitere Operatoren und Funktionen, die im Folgenden näher betrachtet werden.

2.2.1. Grundrechenarten

Wie bereits erwähnt, folgen die Grundrechenarten und die Klammerung der gewohnten Notation. Zur Vervollständigung sei hier ein Beispiel gegeben.

```
1 a = 2
2 b = 4
3 c = 5
4 summe = a + b
5 differenz = c - a
6 produkt = b * c
7 quotient = c // a # division ohne rest
8 rest = c % a # rest der division
9 irgendwas = a*c - b/(b-a) - a
```

Listing 2.9: Beispiele für den Gebrauch von Grundrechenarten

2.2.2. Division und Modulo

Bei der in Listing 2.9 verwendeten Division handelt es sich um eine ganzzahlige Division. Für die gewöhnliche Division bedarf es noch der Einführung des Datentyps `float` in 2.3, da der Typ `int` das Ergebis gar nicht fassen könnte. Wird also wie im Beispiel `5` durch `2` geteilt, so erhalten wir mit dem `//`-Operator das Ergebnis nicht `2.5`, sondern schlicht `2`. Der Rest bei Division der ganzen Zahl a durch eine positive Zahl m ist in python der Wert des Ausdrucks `a % m`. Diese Operation heißt auch *Modulo-Operation*.

2.2.3. Potenzen

Eine weitere wichtige Funktion ist die *Potenzfunktion*. In python steht diese wie in Listing 2.10 beschrieben zur Verfügung.

```
1 # a ist eine Zahl ungleich Null
2 quadrat = a ** 2
3 invers = a ** -2
```

Listing 2.10: Berechnet a^2 und sein multiplikatives Inverses

2.2.4. Vergleiche

Außer mit ihnen zu rechnen, lassen sich Zahlen ordnen. Diese Ordnung basiert auf dem Vergleich ihrer Werte und so ist es nicht erstaunlich, dass es auch eine Möglichkeit gibt, Vergleiche beim Programmieren durchzuführen. Das Ergebnis eines solchen ist vom Typ `bool`. Es sagt also aus, ob der Vergleich zutrifft oder nicht. Mögliche Vergleichsoperatoren sind in Listing 2.11 gegeben.

```
1 a == b  # a ist gleich b
2 a != b  # a ist ungleich b
3 a > b   # a ist echt grösser als b
4 a < b   # a ist echt kleiner als b
5 a => b  # a ist grösser oder gleich b
6 a <= b  # a ist kleiner oder gleich b
```

Listing 2.11: Verschiedene Vergleiche in python

2.3. Float

Mit ganzen Zahlen lassen sich viele Dinge errechnen, aber bei der Division wurde schon ersichtlich, dass mindestens ein weiterer Datentyp benötigt wird, um beispielsweise das zu

erwartende Ergebnis von `5 / 2` speichern zu können. Um diese *Gleitkommazahlen* (engl.: *floats*) geht es in diesem Abschnitt. Die meisten Rechenoperationen mit Gleitkommazahlen folgen dem gewohnten Schema und sind in ihrer Verwendung identisch zu denen mit Zahlen vom Typ `int`. Notiert und ausgegeben werden sie in python gemäß der angloamerikanischen Notation für Zahlen mit einem „.“ als Dezimaltrennzeichen.

```
1 23 // 2   # 11, zur Erinnerung
2 23 / 2    # 11.5
3 23.0/2    # 11.5
4 23/2.0    # 11.5
```

Listing 2.12: Verschiedene Divisionen in python

Um sicher zu gehen, von welchem Typ eine Variable ist, lässt sich in python die `built-in`-Funktion `type` verwenden.

```
1 a = 5
2 b = 5.0
3 type(a)   # <type 'int'>
4 type(b)   # <type 'float'>
```

Listing 2.13: Verwenden der Funktion `type` in python

Es sei hier angemerkt, dass sich das Verhalten des Divisionsoperators `/` von python-Version 2 nach Version 3 geändert hat. Im Zweifelsfall verschafft der Aufruf `python -V` Klarheit über die installierte Version.

Der Abschnitt 2.2.2 bezog sich auf die Division ganzer Zahlen, die immer eine ganze Zahl liefert, und mit Zahlen vom Typ `int` lässt sich auch nichts anderes darstellen. Anders verhält es sich jetzt mit den Zahlen vom Typ `float`. Diese besitzen einen *Nachkommaanteil*. Auch solche Zahlen lassen sich *ganzzahlig* durcheinander teilen. Soll die Gleitkommazahl `5.0` durch `2.0` ganzzahlig geteilt werden, so lässt sich das in python wie in Listing 2.14 bewerkstelligen. Die ganzzahlige Division von Gleitkommazahlen gleicht im Ergebnis dem von Abrunden nach `float`-Division, dargestellt als `float`.

```
1 ergebnis = 5.0 // 2.0 # 2.0
2 ergebnis = 5.1 // 2.6 # 1.0
3 ergebnis = -5.1 // 2.6 # -2.0, Vorsicht mit Vorzeichen!
```

Listing 2.14: Verwenden der Ganzzahligen divison `//` in python

Typumwandlungen

Das Ergebnis einer `float`-Division ist stets vom Typ `float`. Interessiert man sich nur für die Stellen vor dem Dezimaltrennzeichen, oder möchte man nur diese speichern, so kann man unter dem erwarteten Informationsverlust die Gleitkommazahl in eine Ganzzahl *umwandeln* (engl.: to *cast*). In python geschieht diese Konvertierung durch Übergabe eines Wertes an eine Funktion, die den gleichen Namen trägt, wie der Typ, in den man umwandeln will:

```
1 print(int(23.5))   # gibt 23 aus
2 print(float(123))  # gibt 123.0 aus
```

Listing 2.15: Verwenden von Typumwandlungen in python

2.3.1. Built-In-Funktionen

Eine Funktion, die fest in den Interpreter eingebaut ist, heißt *eingebaute Funktion* (engl.: *built-in function*). Eine solche Funktion steht jederzeit, unabhängig von geladenen Modulen, zur Verfügung. Neben den bereits vorgestellten Operatoren und Funktionen gibt es einige hilfreiche eingebaute Funktionen, die sich nachstehend aufgelistet finden und sowohl für Variablen vom Typ `int` als auch für Variablen vom Typ `float` definiert sind.

- `abs(x)`
  Gibt den Betrag $|x|$ der Zahl `x` zurück.
- `max(a,b)`
  Gibt den größeren der beiden Werte `a, b` zurück.
- `min(a,b)`
  Gibt den kleineren der beiden Werte `a, b` zurück.
- `round(x, n)`
  Gibt den auf `n` Nachkommastellen gerundeten Wert von `x` zurück.

Im Modul `math` finden sich weitere mathematische Funktionen, etwa zum Berechnen des Sinus, Cosinus oder Logarithmus. Verwenden lässt sich das Modul `math` nach dem Aufruf von `import math`. Ein vollständige Liste zu den dort enthaltenen Funktion findet sich in der python-Dokumentation (http://docs.python.org/library/math.html).

2.4. Zeichenketten

Neben Wahrheitswerten, Ganz- und Gleitkommazahlen gibt es auch einen Datentyp zur Darstellung von Zeichen und *Zeichenketten*: den Datentyp *String* (`str`). Die technische Umsetzung der Zeichenrepräsentation basiert auf einer Tabelle, die Zeichen Zahlen zuordnet. Die am weitesten verbreitete Zeichentabelle heißt *ASCII*[2]. Dies ist eine Tabelle, die ursprünglich aus 128 Einträgen (7 Bit) bestand, jedoch bald zu einer Tabelle mit 256 Einträgen (8 Bit) erweitert wurde. Mittlerweile ist der UTF[3]8-Code weit verbreitet. Er unterstützt acht, 16 oder 32 Bit pro Zeichen und kann damit bis zu 1 114 112 Zeichen verwalten. Die python-Syntax erlaubt mehrere Möglichkeiten, Variablen vom Typ `str` anzulegen. Generell handelt es sich um eine in Anführungszeichen eingefasste Zeichenkette.

[2] American Standard Code for Information Interchange
[3] UCS Transformation Format

Python erlaubt einfache ', doppelte ", dreifach einfache ''' oder dreifach doppelte """ Anführungszeichen zum Deklarieren eines Strings. Wichtig ist, dass ein String nur mit derselben Sequenz von Anführungsstrichen abgeschlossen werden kann, mit der er auch eingeleitet wurde.

```
1 ein_string = 'Zur Kreuzigung links rum. Jeder nur EIN Kreuz!'
2 noch_einer = "And now for something completely different: ..."
3 langer_str = '''Ich bin nicht der "Messias"!
4 Würdet ihr mir bitte zuhören!'''
5
6 letzer_str = """Ich möchte eine Frau sein.
7 Ich möchte, dass ihr...
8 dass ihr mich von jetzt an 'Loretta' nennt."""
```

Listing 2.16: Gültige Zuweisungen von Strings in python

Wie aus Listing 2.16 ersichtlich wird, liegt der Vorteil der dreifach eingefassten Strings darin, dass sich in ihnen Anführungszeichen verwenden lassen. Ebenso werden die Zeilenumbrüche erfasst und mit ausgegeben, beispielsweise mit einem Aufruf von `print`. Alternativ lassen sich Zeilenumbrüche in jedem String mit der *Escape-Sequenz* `\n` festlegen. Eine vollständige Liste der in python zulässigen *Escape-Sequenzen* findet sich in der Dokumentation docs.python.org/reference/lexical_analysis.html.

Weitere Operationen sind

- `+` (*Konkatenation*) hängt zwei Strings hintereinander, etw:
  `"Text"+ "noch ein text"`.
- `<Zeichenkette>.isalpha()`
  `True`, wenn der Text nur aus Buchstaben besteht
- `<Zeichenkette>.isdigit()`
  `True`, wenn der Text nur aus Ziffern besteht.
- `<Zeichenkette>.split(<Trenner>)`
  bricht die Zeichenkette an dem angegebenen Trenner[4] (engl.: *delimiter*) auf, und gibt eine Liste der resultierenden Teilzeichenketten zurück.
- `join(['T','e','x','t'])`
  fügt die Elemente einer Liste zu einem String zusammen.
- `chr(i)` gibt einen String der Länge eins mit dem zu `i` zugehörigen ASCII-Zeichen zurück.
- `ord(c)` gibt die Ordnungsnummer des Zeichens `c` in der verwendeten Zeichentabelle zurück.

[4]Eine Zeichenkette.

Für einen vollständigen Überblick bietet sich neben der Dokumentation ein Aufruf der python-Hilfe an: `help(str)` oder z.B. `help('')`.

2.5. Listen

Eine sehr große Rolle spielt in python der Datentyp `list`. Er speichert eine sortierte, endliche Liste von Referenzen auf Variablen oder Konstanten. Eine Liste wird in Listing 2.17 initialisiert.

```
1 a = 1
2 b = "siebzehn"
3 liste = [ a, True, b, 5 ]
```

Listing 2.17: Der Listentyp in python – Initialisierung

Es ist möglich, Listen mit dem binären Operator `+` aneinanderzuketten. Auch ist es möglich, Elemente an eine vorhandene Liste anzuhängen (engl.: append). Dies geschieht durch den direkten Aufruf der der Liste zugehörigen `append`-Funktion. Man beachte, dass letztere die Liste verlängert – es wird keine neue Liste mit den entsprechenden Einträgen erstellt. Die Funktion `len`, angewandt auf eine Liste, gibt deren Länge zurück. Zugriff auf Elemente der Liste findet durch Anhängen einer Zellennummer in eckigen Klammern an den Listennamen statt.

```
1 liste = [1,2]
2 noch_eine_liste = liste + [ 3, 4 ] # neue liste wird erstellt
3 noch_eine_liste.append(5) # verändert noch_eine_liste
4 len(noch_eine_liste) # liefert 5
5 noch_eine_liste[0] # liefert 1
6 noch_eine_liste[1] # liefert 2
7 noch_eine_liste[-1] # liefert 5
8 noch_eine_liste[-2] # liefert 4
```

Listing 2.18: Der Listentyp in python – Operationen label

Auch hier ist die Lektüre der python-Dokumentation oder der durch **`help(list)`** ausgegebenen Kurzreferenz empfehlenswert.

3. Ein- und Ausgabe

Im Folgenden soll darauf eingegangen werden, wie man in python Programme schreibt, die auf Dateien zugreifen.

3.1. Lesen und Schreiben von Dateien

Dateien lassen sich *anlegen*, *lesen*, *schreiben* und auch wieder *löschen*. Ihr Lebenszyklus ähnelt also dem einer Variablen. Der Unterschied besteht darin, dass Dateien auch nach dem Beenden eines Programms weiterhin bestehen, wohingegen die in einer Variablen gespeicherte Information verloren geht. Auch in python können Dateien manipuliert werden: Die wichtigsten Befehle sind `open`, `read`, `readlines`, `write` und `close`. Einen ersten Eindruck zum Umgang verschaffen die nachstehenden Listings 3.1, 3.2 und 3.3

```
1 file = open("dummy.txt", 'r')     # Öffnen der Datei dummy.txt
2 print(file.read())                # Lesen und Ausgeben des Inhaltes
3 file.close()                      # Schliessen der Datei
```

Listing 3.1: Auslesen und Anzeigen einer Datei

```
1 file = open("dummy.txt", 'r')     # Öffnen der Datei
2 lines = file.readlines()          # Einlesen der Liste der Zeilen
3 for line in lines:
4     print(line)                   # Ausgeben des Inhaltes
5 file.close()                      # Schliessen der Datei
```

Listing 3.2: Auslesen und Anzeigen einer Datei (siehe Kapitel 4.2.3)

```
1 file = open("/tmp/dummy.txt", 'w') # Öffnen/Anlegen der Datei
2 file.write("Romanes eunt domus")   # Etwas in die Datei schreiben
3 file.close()                       # Speichern der Datei
```

Listing 3.3: Editieren und Speichern einer Datei

Der erste Parameter des Befehls `open` ist vom Typ `str` und bezeichnet den Pfad inklusive des Dateinamens der Datei, die es zu öffnen gilt. Der zweite Parameter legt den Modus fest, in dem die Datei geöffnet werden soll. Python unterscheidet die Modi `r` (*read*), `w` (*write*), `a` (*append*). Existiert die angegebene Datei nicht, wird versucht sie anzulegen.

3.2. Benutzereingaben und -ausgaben

Neben den Dateien, die sich in unserem Dateibaum befinden, gibt es auch noch etwas ungewöhnlichere Vertreter. Die Rede ist von *Standardeingabe* (engl. *standard input*), *Standardausgabe* (engl. *standard output*), und der *Standardfehlerausgabe* (engl. *standard error*). Diese „Dateien“ dienen einem Programm zur Kommunikation mit der Außenwelt. Python stellt hierzu im Modul `sys` die drei Dateideskriptoren `stdin`, `stdout` und `stderr` zur Verfügung. Diese verhalten sich wie Deskriptoren von gewöhnlichen Dateien, die zum Schreiben bzw. Lesen geöffnet wurden. Um das Öffnen und Schließen kümmert sich python.

So gibt das python Programm

```
1 import sys
2 sys.stdout.write("Ich spuck dir ins Auge und blende dich.")
```

dasselbe aus wie der folgende Code.

```
1 print("Ich spuck dir ins Auge und blende dich.")
```

Viel interessanter ist an dieser Stelle jedoch das Einlesen über `stdin`. Dies geschieht in python mit `input` oder `raw_input`. Der Unterschied zwischen beiden Funktion ist, dass `raw_input` immer einen Wert vom Typ `str` zurück gibt, während `input` immer versucht, den Typ der Eingabe zu ermitteln und umzuwandeln. Für den Fall, dass der Benutzer nur Zahlen eingegeben hat, wird ein Wert vom Typ `int` oder `float` zurückgegeben. Identisch sind die Parameter beider Funktionen. Der als `prompt` bezeichnete Parameter eröffnet die Möglichkeit, bevor eine Eingabe vom Benutzer getätigt werden soll, diesen mit einem Aufforderungshinweis darauf aufmerksam machen.

```
1 print("Wie alt bist du?")
2 alter = raw_input()
3 print("Du bist " + alter + " Jahre alt.")
```

Listing 3.4: Abfragen von Benutzereingaben ohne `prompt`

```
1 alter = raw_input("Wie alt bist du?")
2 print("Du bist " + alter + " Jahre alt.")
```

Listing 3.5: Abfragen von Benutzereingaben mit `prompt`

Die in Listing 3.4 und 3.5 dargestellten Programme sind *äquivalent*, d. h. sie tun dasselbe. Es ist jedoch aus Gründen der Übersicht Listing 3.5 der Vorzug zu geben.

4. Kontrollstrukturen

Kontrollstrukturen nennt man Konstruktionen in Programmen, die den Ablauf des Programms steuern. Die meisten Programmiersprachen erlauben z.B. die explizit *bedingte Ausführung* von Anweisungsblöcken – eine Konstruktion, die der alltäglichen Umgangssprache entlehnt ist: *Wenn* du willst, *dann* kauf dir ein Eis. In python-Syntax notiert man dies wie in Listing 4.1.

```
1 if eis_gewollt:
2     kaufe_eis()
```

Listing 4.1: Beispiel einer `if`-Kontrollstruktur

Wir finden also die *Bedingung* `eis_gewollt` und die *bedingte Anweisung* `kaufe_eis()`. Listing 4.2 zeigt den allgemeinen Aufbau einer `if`-Kontrollstruktur. Man nennt ihn die *Syntax* der Kontrollstruktur.

```
1 if␣<Bedingung>:
2 ␣␣␣␣<Anweisungsblock>
```

Listing 4.2: Syntax der `if`-Kontrollstruktur

Ein *Anweisungsblock* ist eine Liste von Befehlen. Diese wird in python durch eine nicht leere Menge von Befehlen, die in Zeilen untereinander stehen und gleich weit eingerückt[1] sind, dargestellt (engl.: to *indent* – einrücken).

Die *Bedingung* muss hier ein Ausdruck sein, der ausgewertet einen Wahrheitswert liefert. Ist dieser `True`, so wird der Anweisungsblock ausgeführt, sonst nicht.

4.1. Verzweigungen

Nun können wir die `if-elif-else`-Kontrollstruktur (Listing 4.3) in voller Pracht erklären.

```
1 if␣<Bedingung>:
2 ␣␣␣␣<Anweisungsblock>
3 elif␣<Bedingung>:
```

[1] Es sei darauf hingewiesen, dass das Einrücken mit Leerzeichen anstatt mit Tabulatorzeichen hilft, potentielle Fehler zu vermeiden. Der Grund dafür verbirgt sich in der Tatsache, dass verschiedene Programme, die zum Editieren der Texdateien eingesetzt werden, nicht sauber und flexibel mit Tabulatorzeichen umgehen können. Beharrt etwa ein solches Programm darauf, Tabulatoren in zu ändernden Zeilen stets durch vier Leerzeichen zu ersetzen, so wird der Interpreter – trotz korrekt erscheinenden Codes – fehlerhafte Einrückungen monieren.

```
4 ␣␣␣␣<Anweisungsblock>
5 else:
6 ␣␣␣␣<Anweisungsblock>
```

Listing 4.3: Syntax der `if-elif-else`-Kontrollstruktur

Wird die erste Bedingung als `True` ausgewertet, so wird der nachfolgende Anweisungsblock ausgeführt. Nur, wenn die erste Bedingung `False` ergibt, wird die Bedingung hinter `elif` (kurz für engl. „else if“) ausgewertet. Ergibt diese `True`, so wird der zweite Anweisungsblock durchlaufen. Der dritte Anweisungsblock wird ausgeführt, falls alle Bedingungen `False` ergeben. Die drei Anweisungsblöcke werden in diesem Kontext auch `if`-, `elif`-, und `else`-*Zweig* genannt. Man beachte, dass python beliebig viele `elif`-Zweige zulässt. So lassen sich in Kürze umfangreiche Fallunterscheidungen implementieren.

In Aktion könnte die neu gelernte Konstruktion wie folgt aussehen.

```
1 if x<0:
2     print("Das Vorzeichen ist negativ")
3 elif x>0:
4     print("Das Vorzeichen ist positiv")
5 else:
6     print("Die Zahl ist 0")
```

Listing 4.4: Ermittelt das Vorzeichen der Zahl `x` mit Verwendung von `elif`.

Alle mit `elif`-Anweisungen erzeugten Programme lassen sich auch mit dem reinen `if-else`-Konstrukt umsetzen. Bei vielfachen Abfragen erhöht sich jedoch auch die Tiefe der Einrückung, was schnell zu unübersichtlichem Code führt. Daher sollte von `elif` Gebrauch gemacht werden, wann immer es sich anbietet.[2].

4.2. Schleifen

Neben den Verzweigungen bilden die Schleifen das zweite Standbein der imperativen Programmierung. Generell werden drei Typen von Schleifen unterschieden, die im Folgenden anhand des Beispiels des Aufzählens der ersten 100 Zahlen erläutert werden.

4.2.1. Vorprüfende Schleifen

Hier steht die zu prüfende Bedingung vor dem Schleifenrumpf. Folglich muss der Rumpf[3] nicht ausgeführt werden, falls sich die Bedingung bereits beim ersten Aufruf als *Falsch* herausstellt. In python wird diese Form mit dem Schlüsselwort `while` eingeleitet.

```
1 while␣<Bedingung>:
2 ␣␣␣␣<Anweisungsblock>
```

[2] python kennt die elegantere `switch`-Anweisung leider nicht!
[3] So nennt man den Anweisungsblock einer Schleife auch.

Listing 4.5: Schematische Darstellung der vorprüfenden Schleife

Der konkrete python-Code zum Ausgeben der ersten 100 Zahlen unter Verwendung der `while`-Schleife sieht wie folgt aus:

```
1 x = 0
2 while x<100:
3     print(x)
4     x = x+1
```

Listing 4.6: Gibt die ersten 100 Zahlen mit Hilfe einer vorprüfenden Schleifen aus

Wird `x` mit `100` initialisiert[4], wird der Schleifenrumpf nie ausgeführt, da die Bedingung `x<100` nicht erfüllt ist. Wird im Rumpf versäumt `x` zu inkrementieren (`x = x+1`), bricht die Schleife nie ab, und das Resultat ist eine Endlosschleife.

4.2.2. Nachprüfende Schleifen

Im Gegensatz zur vorprüfenden Schleife wird der Rumpf der nachprüfenden Schleife mindestens einmal ausgeführt. In python ist dieses Konstrukt nicht mit einem eigenen Schlüsselwort belegt. In anderen Programmiersprachen wird die nachprüfende Schleife meistens mit dem Schlüsselwort `do` oder `repeat` eingeleitet, worauf der Anweisungsblock folgt, dem sich ein `while` oder `until` mit der zu prüfenden Bedingung anschließt. In python hingegen lässt sich eine nachprüfende Schleife wie in *Listing* 4.7 angegeben improvisieren.

```
1 while␣True:
2 ␣␣␣␣<Anweisungsblock>
3 ␣␣␣␣if␣not␣<Bedingung>:
4 ␣␣␣␣␣␣␣␣break
```

Listing 4.7: Schematische Darstellung der nachprüfenden Schleife

Neu ist das Schlüsselwort `break`. Es veranlasst python, die Ausführung der Schleife abzubrechen. Neben `break` gibt es noch das Schlüsselwort `continue`. Mit `continue` wird der python-Interpreter angewiesen mit der nächsten Iteration fortzufahren. Wird dieses `break` weggelassen, oder nie erreicht, weil zum Beispiel eine entsprechende Wahl für die Abbruchbedingung getroffen wurde, oder ein entsprechend platziertes `continue` im Wege steht, so resultiert daraus in beiden Fällen erneut eine Endlosschleife. Die *nachprüfende* Schleife bezieht ihren Namen daher, dass das Abbruchkriterium am Ende des Schleifenrumpfes überprüft wird. Das Ausgeben der ersten 100 Zahlen geschieht in python zum Beispiel mit folgendem Code:

```
1 x = 0
2 while True:
```

[4] fangen wir also mit `x = 100` an

```
3     print(x)
4     x = x+1
5     if x>=100:
6         break
```

Listing 4.8: Gibt die ersten 100 Zahlen mit Hilfe einer nachprüfende Schleifen aus

Unter Verwendung einer nachprüfenden Schleife, die die Zahlen von 10 bis einschließlich 20 ausgibt, lässt sich in python schreiben:

```
1 x = 0
2 while True:
3     x = x+1
4     if x<10:
5         continue
6     print(x)
7     if not x<20:
8         break
```

Listing 4.9: Gibt die Zahlen 10 bis einschließlich 20 mit Hilfe einer nachprüfenden Schleifen aus

Mehr Sinn ergibt es, `x` gleich mit einem Wert von `10` zu initialisieren. Doch oftmals liegen die Dinge nicht so offensichtlich vor, weswegen Konstrukte wie in *Listing* 4.9 dargestellt durchaus in der Praxis anzutreffen sind.

4.2.3. Zählerschleifen

Der wohl am meisten verwendete Schleifen-Typ ist die *Zählerschleife*. Sie kommt im Allgemeinen immer dann zum Einsatz, wenn ein *Index*[5] innerhalb eines bekannten Bereiches (*Startwert*, *Endwert*) durchlaufen werden soll. In python lässt sich eine solche Zählerschleife schematisch wie folgt beschreiben.

```
1 for␣Index␣in␣range(<Startwert>,␣<Endwert>[,␣<Inkrement>]):
2 ␣␣␣␣<Anweisungsblock>
```

Listing 4.10: Schematische Darstellung der Zählerschleife

Neu an dieser Stelle ist der Befehl `range`, mit dessen Hilfe sich der gewünschte Wertebereich erzeugen lässt. Für die ersten 100 Zahlen unter Zuhilfenahme einer Schleife resultiert der folgenden python Code:

```
1 for x in range(0, 100):
2     print(x)
```

Listing 4.11: Gibt die ersten 100 Zahlen (ab 0) mit Hilfe einer Zählerschleife aus

[5] in den vorangegangen Beispielen war dies die Variable `x`

Das in 4.10 angegebene *Inkrement* wurde in 4.11 weggelassen, da es *per default* auf 1 gesetzt ist. Sollen hingegen alle Zahlen in *Zweierschritten* zwischen 0 und 100 ausgegeben werden so könnte das Fragment aus 4.11 zu

```
1 for x in range(0, 100, 2):
2     print(x)
```

angepasst werden.

Zu beachten ist, dass python die Menge der zu durchlaufenden Zahlen vorab generiert. Daher ist es **nicht** möglich, den Wert der *Index*-Variablen `x` während des Schleifendurchlaufes zu verändern, wie es bei der `while`-Schleife der Fall gewesen ist. Streng genommen handelt es sich daher bei der `for`-Schleife nicht um eine Zählerschleife, sondern um einen Iterator, der die Elemente einer mittels `range` erzeugten Liste aufzählt. Das Überspringen mit Hilfe von `continue` oder Abbrechen durch einen Aufruf von `break` ist jedoch auch bei Iteratoren uneingeschränkt möglich.

4.3. Funktionen

Ein Merkmal des imperativen Programmierens bilden die sogenannten *Funktionen*. Darunter lässt sich ein mit einem Namen versehener Anweisungsblock verstehen. Beim Programmieren muss oft ein bereits gelöstes Problem erneut – an einer anderen Stelle im Code – gelöst werden. Der naive Ansatz wäre es, das Codefragment, welches zur Lösung des Problems geführt hat, an die betreffende Stelle zu kopieren. Daraus ergeben sich jedoch Nachteile: Zum einen wird ein solches Programm schnell unübersichtlich, da es aus wesentlich mehr Zeilen Code besteht. Zum anderen ist es schlecht zu warten. Angenommen, es hat sich ein Fehler in dem kopierten Code-Fragment eingeschlichen, so muss dieser Fehler nicht nur einmal korrigiert werden, sondern überall dort, wo das Fragment eingefügt wurde.

Wie bereits oben erwähnt, lässt sich eine Funktion als ein mit einem Name versehener Anweisungsblock auffassen. Dies ist jedoch keine vollständige Beschreibung, denn neben dem Namen und dem Anweisungsblock kann eine Funktion über eine beliebige Anzahl – durch Kommata getrennter – *Parameter* verfügen. Eingeleitet wird eine Funktion mit dem reservierten Wort `def`, gefolgt vom gewünschten Namen der Funktion. In runden Klammern werden die *formalen* Parameter der Funktion eingefasst. Benötigt die Funktion keinen Parameter, so bleiben die Klammern leer. Der Deklarationsteil einer Funktion wird mit einem Doppelpunkt abgeschlossen, dem sich (eingerückt) der Anweisungsblock der Funktion anschließt.

```
1 def␣Name(<Parameter>[,␣<noch␣ein␣Parameter>,␣...␣]):
2 ␣␣␣␣<Anweisungsblock>
```

Listing 4.12: Schematischer Aufbau einer Funktion

Eine Funktion, die das Quadrat einer Zahl `n` berechnet und auf dem Bildschirm ausgibt, kann in python wie in Listing 4.13 erzeugt werden.

```
1 def gib_quadrat_aus(n):
2     print("Das Quadrat von " + n + " ist " + n*n)
```

Listing 4.13: Berechnet das Quadrat der Zahl `n` und gibt dieses aus

Aufrufen lässt sich die Funktion zum Beispiel mit `quadrat(5)`. Sollen die Quadrate der ersten 100 Zahlen ausgegeben werden, so kann man das z.B. so umsetzen:

```
1 def gib_quadrat_aus(n):
2     print("Das Quadrat von " + n + " ist " + n*n)
3
4 for i in range(0, 100):
5     gib_quadrat_aus(i)
```

Listing 4.14: Berechnet jeweils das Quadrat der Zahlen von 0 bis 100 und gibt diese **aus**

Funktion können auch einen Rückgabewert besitzen[6], ähnlich den aus der Mathematik bekannten Funktionen. So liefert beispielsweise die Funktion $sin(x)$ den Sinus einer Gleitkommazahl x. Dieses *Zurückliefern* geschieht in python mit dem Befehl `return`, gefolgt von dem *Rückgabewert*.

Um das Beispiel aus Listing 4.13 erneut aufzugreifen lässt sich eine Funktion zum Berechnen des Quadrates der Zahl `n` als

```
1 def quadrat(n):
2     return n*n
```

Listing 4.15: Berechnet das Quadrat der Zahl `n` und gibt dieses **zurück**

schreiben. Analog dazu wird aus dem Beispiel aus Listing 4.14:

```
1 def quadrat(n):
2     return n*n
3
4 for i in range(0, 100):
5     print("Das Quadrat von " + i + " ist " + quadrat(i))
```

Listing 4.16: Berechnet die Quadrate der Zahlen von 0 bis 100 und gibt diese **aus**

4.4. Gültigkeitsbereiche

Der *Gültigkeitsbereich* (engl.: *scope*) einer Variablen ist der Bereich im Code, in dem diese sich bei Nennung ihres Namens angesprochen fühlt. Ein Gültigkeitsbereich ist – ähnlich einem Anweisungsblock – eine Menge von Befehlszeilen, die in einem bestimmten Zusammenhang stehen.

[6] In manchen Programmiersprachen werden Funktionen ohne Rückgagebwert auch als Prozeduren bezeichnet.

In python ist dem Gültigkeitsbereich einer Variablen besondere Aufmerksamkeit zu widmen. Dies liegt an der Praxis der impliziten Typdeklaration: Die Möglichkeit, den Gültigkeitsbereich einer Variablen manuell festzulegen, scheidet aus. Daher muss der Interpreter mittels eines komplexen Regelwerks ableiten, was der Programmierer wohl gemeint hat. Dieses Regelwerk sprengt den Rahmen dieses Skripts, jedoch wollen wir hier anhand eines Beispiels in Listing 4.17 möglichen Kopfschmerzen vorbeugen.

```
1 def tuwas(c):
2     a = 2
3     print "a   in tuwas:", a
4     print "b   in tuwas:", b
5     print "c   in tuwas:", c
6     c[0]=2
7 a = 1
8 b = 1
9 c = [1]
10 print "a  vor tuwas:", a
11 print "b  vor tuwas:", b
12 print "c  vor tuwas:", c
13 tuwas(c)
14 print "a nach tuwas:", a
15 print "b nach tuwas:", b
16 print "c nach tuwas:", c
```

Listing 4.17: Kopfschmerzen?

Wir haben es hier mit zwei Gültigkeitsbereichen zu tun. Einer ist innerhalb der Funktion `tuwas`, einer außerhalb. Welche Variablen sich der Interpreter in jedem einzelnen Fall ausgesucht hat, erkennt man am besten an der Ausgabe des Programms in Abbildung 4.18.

```
1 a  vor tuwas: 1
2 b  vor tuwas: 1
3 c  vor tuwas: [1]
4 a   in tuwas: 2
5 b   in tuwas: 1
6 c   in tuwas: [1]
7 a nach tuwas: 1
8 b nach tuwas: 1
9 c nach tuwas: [2]
```

Listing 4.18: Ausgabe des Programms

5. Rekursion und Iteration

Ein Programm, das ein Problem löst, so wie ein Backrezept einen Kuchen backt, nennt man einen *Algorithmus*. Kleine Algorithmen habt Ihr in z.B. in den Listings 4.8, 4.9, 4.11, 4.2.3, 4.14 und 4.16 kennengelernt. Bei den bislang beschriebenen Algorithmen handelt es sich um *iterative* (iterare (lat.) – wiederholen) Implementierungen: Es wird explizit festgelgt, unter welchen Bedingungen oder wie oft ein Anweisungsblock durchlaufen werden soll.

Mit Hilfe von Funktionsdeklarationen kann man den Programmcode solcher Algorithmen zusammenfassen und mit einem Namen versehen. So kann man z.B. Teile eines langen Programms in Funktionen auslagern. Das ist sehr nützlich, um Programme übersichtlich und modular zu halten. Funktionsdeklarationen eröffnen aber noch eine weitere Möglichkeit: Im Anweisungsblock einer Funktion kann die Funktion selbst aufgerufen werden. So ist also möglich – ähnlich wie mit Iteratorschleifen – die wiederholte Ausführung von Anweisungen zu veranlassen. Dieses Prinzip nennt sich *Rekursion*. Der Begriff Rekursion (recurrere (lat.) – zurückkehren) benennt das Zurückkehren in die Zeilen, die soeben schon ausgeführt wurden. Um hieraus einen praktischen Nutzen zu ziehen, müssen wir zunächst ein Problem formulieren, dessen Lösung wir dann als Rekursion in python implementieren werden.

5.1. Fakultät und Matrikelnummern

Stellen wir uns vor, wir wollen 50 Neulingen die Matrikelnummern 1-50 zuordnen. Wieviele Möglichkeiten gibt es hierfür? Legen wir einfach los! Wir nehmen uns einen Neuling und geben ihm die 1. Wir rechnen nach, dass jetzt noch 49 Nummern zu vergeben sind. Aber was, wenn wir einen anderen Neuling gewählt hätten? Dann wären auch 49 übrig geblieben. Egal, welche der 50 möglichen Wahlen wir getroffen haben, es bleiben immer 49 Nummern übrig. Wir sehen also, dass es 50 Mal soviele Möglichkeiten gibt, die 50 Nummern zu vergeben, wie es Möglichkeiten gibt, 49 Nummern an 49 Neulinge zu vergeben. Ebenso liegt nahe, dass es für die Vergabe von x Nummern x Mal soviele Möglichkeiten wie für die Vergabe von $x-1$ Nummern gibt. Aber halt! Haben wir keine Nummer zu vergeben, so fürt uns diese Rechnung in die Verlegenheit, -1 Nummern vergeben zu müssen. Vor dem Anwenden unserer Rechenregel bedarf es einer Fallunterscheidung. Da wir Funktionen und Fallunterscheidungen bereits in python auszudrücken wissen, gelingt es, diese Überlegung in fünf Zeilen wie in Listing 5.1 darzustellen.

```
1 def f(n):
2     if n<=1:
3         return 1
4     return n * f(n-1)
```

```
5 print("Es gibt", f(50), "Möglichkeiten")
```

Listing 5.1: Rekursive Berechnung der Fakultät `n` in python

Die Funktion `f` nennt man *Fakultät* (engl.: *factorial*). Sie ist auch unter ihrem englischem Namen bereits in python verfügbar[1]

Ein Aufruf von `f(n)` wird also den Interpreter veranlassen, `f(n-1)` auszuwerten, falls der Wert von `n` groß genug ist. Liefert der Ausdruck `n<=0`, so findet kein weiterer Aufruf von `f` statt. Man nennt diesen Ausdruck das *Abbruchkriterium*.

Das Abbruchkriterium der Rekursion entspricht in etwa der Schleifenbedingung in Kapitel 4.2. In der python-Welt gibt es jedoch einen nicht unwesentlichen Unterschied: Jedes Mal, wenn der Interpreter `f` aufruft, muss er sich merken, wo er herkam, um zum Beispiel mit dem Befehl `return` an die richtige Stelle zurückspringen zu können. Das für diesen Mechanismus zuständige Gedächtnis des Interpreters nennt man den *Stapel* (engl.: *stack*). Treibt man es zu weit mit der Tiefe der Rekursion, so ist dieser Stapel voll und der Programmablauf wird außerplanmäßig beendet. Mit jedem Funktionsaufruf werden Daten auf den Stapel gelegt (engl.: push) und nach einem `return` wieder heruntergenommen (engl.: pop). Dies ist in Abbildung 5.1 illustriert. Wird das Abbruchkriterium nie erreicht, werden ständig neue Einträge auf den Stapel gelegt, solange bis kein freier Speicher zur Verfügung steht. Es kommt zu dem oben erwähnten Fehler, einem *Stapelüberlauf* (engl.: *stack overflow*).

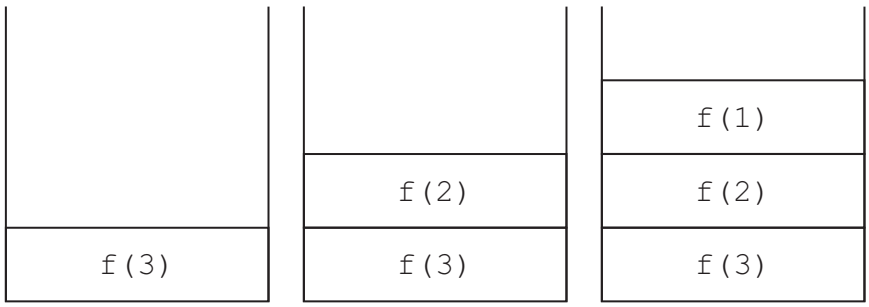
Zum Vergleich geben wir in Listing 5.2 auch ein Programm, welches die Fakultät iterativ berechnet an. Man beachte, dass diesem Algorithmus nicht direkt anzusehen ist, dass er dasselbe implementiert, wie der in Listing 5.1. Dies liefert eine beliebte Übungsaufgabe.

```
1 def f(n):
2     if n<1:
3         return 1
4     tmp = 1
5     for i in range(1, n+1):
6         tmp = tmp * i
7     return tmp
8 print("Es gibt", f(50), "Möglichkeiten")
```

Listing 5.2: Iterative Berechnung der Fakultät

Sieht man von der Problematik des Stapels ab, so besteht der Unterschied zwischen Iterationen und Rekursionen darin, dass rekursiv formulierte Problemlösungen besser die Struktur des Problems wiederspiegeln. Sie sind häufig leichter zu lesen als ein iterativer Ansatz, da zur Lösung verwendete Zwischenergebnisse im Programmcode erkennbar sind.

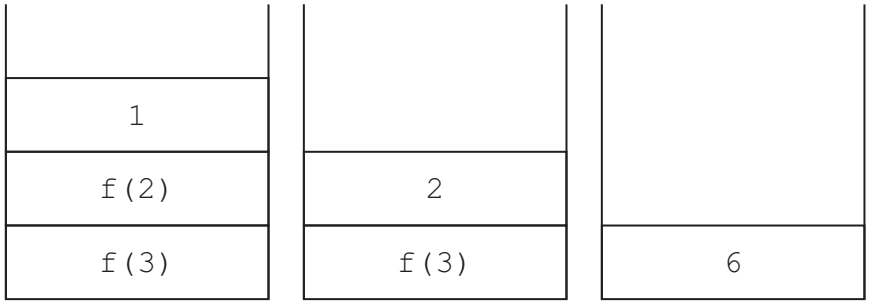
[1] Nachdem ein `from math import factorial` ausgeführt wurde.



(a) Aufruf der Funktion `f` mit dem Wert `3`,

(b) erzeugt einen Aufruf der Funktion `f` mit dem Wert `2`,

(c) erzeugt einen Aufruf der Funktion `f` mit dem Wert `1`,



(d) `f(1)` genügt dem Abbruchkriterium und ergibt den Wert `1`.

(e) Daher ergibt sich für `f(2)` ein Wert von `2`

(f) Und schließlich wird `f(3)` zu `6` ausgewertet

Abbildung 5.1.: Verlauf des *Stapels* für einen Aufruf der rekursiven Funktion `f` zur Berechnung der Fakultät von `3`

5.2. Fibonacci und die Kaninchen

Kaninchen vermehren sich – wie die Hasen. Dies ist auch Leonardo Fibonacci im 13. Jahrhundert aufgefallen. Er wollte sodann wissen, wie viele Kaninchen nach einem festen Zeitraum vorhanden sind, wenn man mit einer bestimmten Anzahl anfängt. Fibonacci hat dies anhand einfach gestrickter Kaninchen untersucht[2]. Seine Kaninchen sind unsterblich und produzieren ab ihrem zweiten Lebensmonat monatlich ein weiteres Kaninchen. Nur das Urkaninchen materialisiert sich spontan zu Beginn des ersten Monats.

Wir überlegen uns, dass im ersten Monat also genau ein Kaninchen existiert. Im zweiten Monat gibt es ein weiteres, da das Urkaninchen inzwischen eines geworfen hat. Wir überschlagen, dass die Anzahl der Tiere in einem Monat unserer Wahl die Summe der Kaninchenanzahl im Vormonat und der Anzahl der Kaninchen, die zu Monatsbeginn hinzugekommen sind, sein muss. Für die hinzukommenden waren aber gerade die hinreichend alten Kaninchen verantwortlich, das sind genau die, die im Vorvormonat schon da waren.

Zusammenfassend müssen wir also die Kaninchenzahl vom Vormonat mit der Kaninchen-

[2]Wir wollen die Geschichte etwas vereinfachen, um leichter das Wesentliche zu erkennen.

zahl vom Vorvormonat addieren, um die Kaninchenpopulation eines Monats zu berechnen. Wieder präsentieren wir ein python-Programm, das diese Rechnung durchführt, und wieder müssen wir bei kleinen Zahlen etwas vorsichtig sein.

```
1 def anzahl_kaninchen(monat):
2     if monat<1:
3         return 0
4     if monat=1:
5         return 1 # das Urkaninchen
6     return anzahl_kaninchen(monat-1) + anzahl_kaninchen(monat-2)
7
8 print("Im vierten Monat gibt es", anzahl_kaninchen(4), "Kaninchen.")
```

Listing 5.3: Anzahl der Kaninchen im vierten Monat

6. Laufzeit

6.1. Maximum finden

Stellen wir uns vor, wir haben eine Liste von Zahlen und interessieren uns für eine größte darin enthaltene Zahl. Unser *Problem* ist also das Auffinden der Zahl, die sich im Vergleich mit den anderen Zahlen unserer Liste jeweils als nicht kleiner erweist.

Python stellt eine Funktion zur Verfügung, die hierfür eine Lösung implementiert, wie das Beispiel 6.1 illustriert.

```
1 meine_liste = [1, 5, 2, 1, 6, 5]
2 loesung = max(meine_liste)
3 print(loesung)
```

Listing 6.1: Maximum berechnen mit python

Jetzt wollen wir aber wissen, wie lange ein Computer beschäftigt ist, bis er die Lösung ermittelt hat. Hierzu müssen wir unser Programm aus Befehlen zusammensetzen, von denen wir genau sagen können, wie viel Zeit deren Ausführung in Anspruch nimmt. Wir kennen Zuweisungen, etwas Arithmetik und Fallunterscheidungen, nur diese sollen daher verwendet werden. Außerdem wollen wir unser Programm derart zerpflücken, dass in jeder Zeile nur noch höchstens eine solche einfache Operation steht[1]. Nehmen wir mal vereinfachend an, diese Dinge dauerten alle gleich lang und zwar 1μs.

Wir konstruieren den folgenden Code

```
1  # meine_liste ist eine liste mit nichtnegativen zahlen drin.
2  # meine_liste_laenge ist die länge dieser liste
3  hier_bin_ich = 0
4  maximum = 0
5  bin_fertig = hier_bin_ich == meine_liste_laenge
6  while not bin_fertig:
7      aktueller_eintrag = meine_liste[hier_bin_ich]
8      grosses_gefunden = maximum < aktueller_eintrag
9      if grosses_gefunden:
10         maximum = meine_liste[hier_bin_ich]
11     hier_bin_ich = hier_bin_ich + 1
12     bin_fertig = hier_bin_ich == meine_liste_laenge
```

Listing 6.2: Maximum berechnen explizit

[1] Auf Eleganz müssen wir natürlich verzichten.

Wie lange dauert die Ausführung? Da alle Befehle bei uns gleich lang dauern, brauchen wir nur zu *zählen*! Die ersten drei Zeilen sind einfach, diese werden nur einmal ausgeführt. Wie oft wird der `while`-Befehl aufgerufen? Genaues Hinsehen liefert `meine_liste_laenge`+ 1 Mal. Die Ausführung des Rumpfes der `while`-Schleife dauert höchstens 6μs, mindestens jedoch 5μs. Zählen wir alles zusammen, erhalten wir also eine Laufzeit zwischen $(3 + (\texttt{meine_liste_laenge} + 1) \cdot 5)\mu$s und $(3 + (\texttt{meine_liste_laenge} + 1) \cdot 6)\mu$s.

Wir stellen fest: Ist die Liste sehr lang, ist drei, fünf und sechs dagegen klein. Wir müssen also ungefähr zwischen $5 \cdot \texttt{meine_liste_laenge}\mu$s und $6 \cdot \texttt{meine_liste_laenge}\mu$s warten. Man sagt, die Laufzeit ist *linear* in der Länge der Liste: Eine doppelt so lange Liste lässt uns doppelt so lange warten. Dies ist eine ganz einfache Laufzeitabhängigkeit – da überrascht es nicht, dass man sie auch im Alltag wieder findet: Man beobachtet etwa, dass das Annähen dreier Knöpfe dreimal so lange dauert, wie das Annähen eines einzigen. Um dem Begriff Laufzeit einen praktischen Nutzen abzuringen, werden wir ein Programm mit anderem Laufzeitverhalten demonstrieren.

6.2. Finden von Zahlen

Wie eben sei eine Liste von Zahlen zusammen mit der Länge der Liste gegeben. Diesmal wollen wir annehmen, die Liste sei *aufsteigend sortiert*, d. h. Listeneinträge mit größerem Index sind nicht kleiner als die mit kleinerem Index. So ist z.B: die Liste `[1, 4, 7, 7, 9]` aufsteigend sortiert.

Wir fragen uns, ob eine Zahl, die wir `gesuchte_zahl` nennen wollen, in dieser Liste enthalten ist.

Unser erstes Programm implementiert eine Ad-hoc-Lösung.

```
1  # meine_liste ist eine liste mit nichtnegativen zahlen drin.
2  # meine_liste_laenge ist die länge dieser liste
3  # gesuchte_zahl ist die Zahl, nach der gesucht wird
4  hier_bin_ich = 0
5  gefunden = False
6  fertig = hier_bin_ich == meine_liste_laenge
7  while not fertig:
8    aktueller_eintrag = meine_liste[hier_bin_ich]
9    gefunden = aktueller_eintrag == gesuchte_zahl
10   fertig = hier_bin_ich == meine_liste_laenge
11   fertig = fertig or gefunden
12   hier_bin_ich = hier_bin_ich + 1
```

Listing 6.3: Lookup Naiv

Diesmal müssen wir beim Zählen vorsichtiger sein: Wie oft der `while`-Befehl ausgeführt wird, hängt von `gesuchte_zahl` ab. Diese Abhängigkeit soll uns nicht kümmern: Wir untersuchen einfach nur den *schlimmsten Fall* (engl. *worst case*). Mit dem schlimmsten Fall meinen wir den Fall, in dem wir am längsten auf das Ergebnis warten müssen, in dem also

die Laufzeit am größten sein wird. In unserem Beispiel ist dies der, in dem die gesuchte Zahl nicht in der Liste vorkommt, wie man sich leicht überlegen kann. Kurzum, auf der Suche nach der nicht vorhandenen Zahl benötigt der Algorithmus $(3 + \texttt{meine_liste_laenge} \cdot 6 + 1)\mu$s.

Für die, denen das zu lange dauert, gibt es auch einen anderen Ansatz: Schauen wir uns das folgende Programm an:

```
1  # meine_liste ist eine aufsteigend sortierte liste nichtnegativer zahlen
2  # gesuchte_zahl ist eine zahl
3  meine_liste_laenge = len(meine_liste)
4  unterer_rand = 0
5  oberer_rand = meine_liste_laenge - 1
6  gefunden = False
7  fertig = False
8  while ( not gefunden ):
9      mitte = (oberer_rand + unterer_rand) / 2
10     aktueller_eintrag = meine_liste[mitte]
11     gefunden = aktueller_eintrag == gesuchte_zahl
12     if gefunden:
13         break
14     if oberer_rand == unterer_rand:
15         break
16     if aktueller_eintrag < gesuchte_zahl:
17         unterer_rand = mitte + 1
18     else:
19         if aktueller_eintrag > gesuchte_zahl:
20             oberer_rand = mitte - 1
21         else:
22             gefunden = True
```

Listing 6.4: Binäres Suchen

Wo beginnt diese Suche? In der Mitte der Liste! Wird in der Mitte nichts gefunden, so verrät uns ein Größenvergleich, ob wir weiter unten oder weiter oben suchen wollen. Wir merken uns während der Suche den Rand des Suchfensters in den Variablen `unterer_rand` und `oberer_rand`. Dies wird wiederholt, bis in der Mitte die gesuchte Zahl auftaucht, oder im schlimmsten Falle, bis das Suchfenster nicht mehr halbiert werden kann. Etwas kompliziert wirkt dieser Algorithmus dadurch, dass so viele Fälle bedacht werden müssen. Dass der Algorithmus das Richtige tut, könnt Ihr in einer ruhigen Minute nachvollziehen oder ausprobieren – hier interessieren wir uns nur für die Laufzeit.

Wieder steht in jeder Zeile nur ein Befehl, aber diesmal wird das Zählen etwas kniffliger werden. Dafür erlauben wir uns, ein wenig zu überschlagen: Nach dem `while` in Zeile 8 kommen ja nur noch 14 Zeilen. Es kann passieren, dass manche gar nicht in jedem Durchlauf ausgeführt werden. Dies ignorieren wir und verbuchen fünf Befehle vor dem `while`, sowie *höchstens* 14 Programmzeilen, die so lange durchlaufen werden, bis schließlich ein `break`-Befehl dem Treiben ein Ende bereitet, wenn die Liste nicht mehr halbiert werden kann. Wie

oft aber wird die Liste halbiert? Wir überschlagen das wie folgt: Einmal halbieren entspricht einer Liste der Länge zwei. Zweimal halbieren entspricht einer Liste der Länge vier. Dreimal der Länge acht. Wenn wir x Mal halbieren müssen, liegt das also daran, dass unsere Liste mindestens 2^x lang ist. Umgekehrt: hat unsere Liste die Länge n, so reichen bestimmt $\log_2(n)$ Halbierungen. Wir müssen also alles in allem *höchstens* $5 + 14 \cdot \log_2(n)$ Befehle ausführen um ans Ziel zu gelangen. Interessant ist, dass dies, obwohl wir so großzügig überschlagen haben, immer noch *viel* besser ist als unser erster Versuch in Listing 6.3.

Zusammenfassend kann man sagen: Der zweite Algorithmus ist schneller, *aber* der erste funktioniert auch mit unsortierten Listen. Wir haben also in unserem Beispiel eine Zusatzinformation in eine bessere Laufzeit umgemünzt.

Teil II.

Theoretische Informatik

7. Logik

7.1. Einleitung

Die Logik behandelt die allgemeinen Prinzipien des korrekten Argumentierens. Diese Prinzipien gelten allein aufgrund der *Form* der Aussagen und sind unabhängig vom konkreten Inhalt.

Beispiel 1 (Form einer Aussage) *Aus den Aussagen „Wenn Paul in der Sonne liegt, dann holt er sich einen Sonnenbrand“ und „Paul liegt in der Sonne“ können wir schließen: „Paul holt sich einen Sonnenbrand“.*

Setzen wir nun

$A :=$ *„Paul liegt in der Sonne“, und*

$B :=$ *„Paul holt sich einen Sonnenbrand“,*

so hat der obige Schluss als zusammengesetzte Aussage die Form

$$\textit{Wenn } ((A \to B) \wedge A), \textit{ dann } B.$$

Lies: Wenn aus A B folgt und A wahr ist, dann ist auch B wahr.

Das Zeichen „$\to$“ heißt **Implikation**. Man liest „$A \to B$“ als „wenn A, dann B“ (gleichbedeutend: „aus A folgt B“, bzw. „A impliziert B“). Das Zeichen „$\wedge$“ heißt **Konjunktion**. Man liest es als „und“.

Aufgabe 1

1. *Verfahre wie im Beispiel 1 mit den Aussagen*
   a) *„Wenn Lucy Gitarre spielt, dann klopfen die Nachbarn an die Wand“, und*
   b) *„Wenn es regnet, dann wird die Straße nass“.*
2. *Finde ein weiteres passendes Beispiel.*
3. *Lässt sich die Aussage „Wenn $((A \to B) \wedge A)$, dann B“ mit den uns bekannten Zeichen noch weiter formalisieren?*

Jeder Schluss der Form aus Beispiel 1 ist wahr, unabhängig davon, welche Aussagen wir für A und für B nehmen!

Ziel ist es, eine Vorschrift anzugeben, mittels der wir für eine komplexe Aussage berechnen können, ob sie wahr oder falsch ist. Wir berechnen das in Abhängigkeit davon, ob die

atomaren Teilaussagen der komplexen Aussage wahr oder falsch sind. Ob eine komplexere Aussage wahr ist, hängt nur davon ab, ob ihre Teilaussagen wahr oder falsch sind, und davon, wie die Teilaussagen verknüpft sind. Was das heißt, wollen wir uns an den folgenden fünf Beispielen anschauen.

Beispiel 2 (Konjunktion) *Die Aussage*

„Orlando di Lasso lebte im 16. Jahrhundert und *er leitete die bayerische Hofkapelle“*

ist wahr, falls es sowohl wahr ist, dass Orlando di Lasso im 16. Jahrhundert lebte, als auch, dass er die bayerische Hofkapelle leitete.

Allgemein gilt: Wenn wir zwei Aussagen A und B durch Konjunktion verknüpfen, dann ist die entstandene komplexere Aussage genau dann wahr, wenn beide *Aussagen A und B wahr sind: $(A \wedge B)$ ist genau dann wahr, wenn sowohl A als auch B wahr sind.*

Was ist eigentlich eine Aussage? Eine Aussage ist ein Satzgebilde, das wahr oder falsch sein kann. Beispiele für Aussagen sind

„Lucy spielt Gitarre“,

„Deutschland ist nicht Fußballweltmeister“,

„9 geteilt durch 3 ist 2“,

„Der Algorithmus ist korrekt und er ist effizient“, und

„Im Winter regnet es oder es schneit“.

Dagegen sind „Wie ist das Wetter auf Kreta?“ und „Autsch!“ keine Aussagen in unserem Sinne.

Beispiel 3 (Implikation) *Die Aussage*

*„*Wenn *wir die Express-Fähre ab Hirtshals nehmen,* dann *werden wir nach Bergen gelangen.“*

ist wahr, falls „Wir nehmen die Express-Fähre ab Hirtshals“ und „Wir werden nach Bergen gelangen“ wahr sind, ***oder falls „Wir nehmen die Express-Fähre ab Hirtshals“ falsch ist****. Die Faustregel hierzu ist: Aus Wahrem kann man nur Wahres folgern, und aus Falschem kann man alles folgern.*

Allgemein: $(A \to B)$ ist genau dann wahr, wenn sowohl A als auch B wahr sind, oder wenn A falsch ist.

Neben Implikation und Konjunktion verwenden wir auch die Zeichen „$\neg$“ und „$\vee$“. Das Zeichen „$\neg$“ heißt **Negation** und man liest es als „nicht“. Das Zeichen „$\vee$“ heißt **Disjunktion** und man liest es als „oder“.

Beispiel 4 (Negation) *Die Aussage*

„Deutschland ist nicht *Fußballweltmeister“*

ist genau dann wahr, wenn die Aussage „Deutschland ist Fußballweltmeister“ falsch ist.

Allgemein gilt: $\neg A$ ist genau dann wahr, wenn A falsch ist.

Beispiel 5 (Disjunktion) *Die Aussage*

„Im Wintersemester wird die Vorlesung ‘Logik und Datenbanken’ oder *die Vorlesung ‘Logik in der Informatik’ angeboten“*

ist wahr, falls im Wintersemester die Vorlesung ‘Logik und Datenbanken’ angeboten wird oder falls im Wintersemester die Vorlesung ‘Logik in der Informatik’ angeboten wird, oder auch falls im Wintersemester beide Vorlesungen angeboten werden.

Allgemein gilt: $(A \vee B)$ ist genau dann wahr, wenn A wahr ist, oder B wahr ist, oder wenn A und B wahr sind.[1]

Schließlich verwenden wir noch das Zeichen „$\leftrightarrow$“, die sogenannte **Biimplikation**. Man liest es als „genau dann, wenn“. „$A \leftrightarrow B$“ ist nur eine Abkürzung für „$\big((A \to B) \wedge (B \to A)\big)$“.

Beispiel 6 (Biimplikation) *Die Aussage*

„Deutschland schneidet genau dann *bei der PISA-Studie besser ab,* wenn *der Staat wieder mehr Geld in die Bildung investiert hat.“*

ist wahr, falls sowohl die Aussage „Wenn der Staat wieder mehr Geld in die Bildung investiert hat, dann schneidet Deutschland bei der PISA-Studie besser ab“ wahr ist, als auch die Aussage „Wenn Deutschland bei der PISA-Studie besser abschneidet, dann hat der Staat wieder mehr Geld in die Bildung investiert“ wahr ist.

Allgemein gilt: $(A \leftrightarrow B)$ ist genau dann wahr, wenn $(A \to B) \wedge (B \to A)$ wahr ist.

Die Zeichen $\wedge, \vee, \neg, \to$ und $\leftrightarrow$ heißen auch **Junktoren**. Mit Hilfe der Junktoren kann man bereits sehr viele Aussagen formalisieren und analysieren.

Beispiel 7 *Wir formalisieren den Satz*

„Wenn die Klimaanlage im Rechenzentrum ausfällt, dann schalten sich die Computer automatisch ab und es wird Alarm ausgelöst“.

Dazu setzen wir
$A :=$ „Die Klimaanlage im Rechenzentrum fällt aus“
$B :=$ „Die Computer schalten sich automatisch ab“, und
$C :=$ „Es wird Alarm ausgelöst“.

Dann hat die Aussage die Gestalt $(A \to (B \wedge C))$.

[1] Beachte: Die Disjunktion wird nicht als ausschließendes Oder („entweder ... oder ..., aber nicht beides“) verwendet.

Anwendungen Logik spielt in der Informatik eine wichtige Rolle. Anwendungen sind z.B.

- Modellierung von Wissen, etwa in der künstlichen Intelligenz
- Automatische Verifikation (automatisches Testen, ob ein System die Spezifikationen erfüllt)
- Kontrollfluss von Computerprogrammen
- Logikbauteile in der Hardware
- Datenbanken: Auswerten von Anfragen
- Mathematische Beweise
- Korrektes Argumentieren (nicht nur hilfreich für die Informatik...)

7.2. Aussagenlogik

Die formale Sprache, die wir im vorhergehenden Abschnitt anhand von Beispielen kennengelernt haben, heißt Aussagenlogik. Wir wollen die Aussagenlogik in diesem Abschnitt exakt definieren. Um dies zu tun, müssen wir die Syntax und die Semantik der Aussagenlogik festlegen.

Syntax Mit der Syntax legen wir fest, welche Zeichenreihen gültige Formeln sind. So soll etwa „$(A \vee B)$“ eine gültige Zeichenreihe sein, wohingegen die Zeichenreihe „$\vee()AB$“ nicht erlaubt sein soll.

Eine Analogie zur Syntax der Aussagenlogik ist der Satzbau im Deutschen: Die Reihenfolge „Subjekt, Prädikat, Objekt“ (etwa: „Lucy spielt Gitarre“) ist gültig, wohingegen die Reihenfolge „Objekt, Subjekt, Prädikat“ („Gitarre Lucy spielt“) nicht erlaubt ist.

Definition 1 (Syntax der Aussagenlogik) *Wir nehmen an, dass wir eine unendliche Menge $A, B, A_1, A_2, A_3, \ldots, B_1, B_2, B_3, \ldots$ von **Atomen** (auch **aussagenlogische Variablen** genannt) gegeben haben. Aussagenlogische **Formeln** sind Zeichenreihen, die aus den Atomen, den Junktoren $\wedge, \vee, \neg, \rightarrow$ und $\leftrightarrow$ und den beiden Klammern (und) nach den folgenden Regeln aufgebaut werden.*

- *Atome sind Formeln.*
- *Wenn F_1 und F_2 Formeln sind, dann ist auch $(F_1 \wedge F_2)$ eine Formel (**Konjunktion**).*
- *Wenn F_1 und F_2 Formeln sind, dann ist auch $(F_1 \vee F_2)$ eine Formel (**Disjunktion**).*
- *Wenn F_1 und F_2 Formeln sind, dann ist auch $(F_1 \rightarrow F_2)$ eine Formel (**Implikation**).*

- *Wenn F_1 und F_2 Formeln sind, dann ist auch $(F_1 \leftrightarrow F_2)$ eine Formel (**Biimplikation**).*
- *Wenn F eine Formel ist, dann ist auch $\neg F$ eine Formel (**Negation**).*

So sind z.B. $(A \vee B)$ und $(A_1 \wedge A_3) \vee (A_2 \rightarrow (\neg A_1))$ aussagenlogische Formeln, während $(A_2\neg \vee A_1)$ und $(A_2\vee \leftrightarrow A_1)$ keine aussagenlogischen Formeln sind.

Semantik Die Semantik legt fest, welche Bedeutung die erlaubten Formeln haben. Mit „Bedeutung“ meinen wir hier nur, ob die Formel wahr oder falsch ist – jeweils in Abhängigkeit davon, ob ihre Atome wahr oder falsch sind. Wir werden von konkreten Inhalten der atomaren Aussagen absehen und **belegen** die Atome einfach entweder mit „wahr“ oder „falsch“, den sogenannten **Wahrheitswerten**. Wir verwenden 1 für „wahr“ und 0 für „falsch“.

Definition 2 (Semantik der Aussagenlogik) *Sei F eine aussagenlogische Formel. In Abhängigkeit von den Wahrheitswerten der Teilformeln von F lässt sich der Wahrheitswert von F berechnen. Das geschieht nach bestimmten Regeln, die wir hier durch **Wahrheitstafeln** angeben.*

Die Tafeln sind folgendermaßen zu lesen: in den Spalten links von Doppelstrich stehen alle möglichen Kombinationen der Wahrheitswerte der Teilformeln, und rechts steht für jede Kombination der Wahrheitswerte der Teilformeln, welcher Wahrheitswert sich daraus für die gesamte Formel ergibt.

F_1	F_2	$(F_1 \wedge F_2)$	$(F_1 \vee F_2)$	$(F_1 \rightarrow F_2)$	$(F_1 \leftrightarrow F_2)$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

F	$\neg F$
0	1
1	0

Wenn wir also in der Formel $(A \vee B)$ die Variable A mit 0 belegen und die Variable B mit 1, dann sagt uns die zweite Zeile in der vierten Spalte der ersten Tabelle, dass der Wahrheitswert der gesamten Formel 1 ist. Die Formel $(A \vee \neg B)$ hingegen hat bei derselben Belegung den Wahrheitswert 0: die letzte Tabelle sagt uns, dass $\neg B$ den Wahrheitswert 0 hat, und dann ist nach der ersten Zeile der zweiten Spalte in der ersten Tabelle der Wahrheitswert der gesamten Formel 0.

Die Semantik ist also schlicht eine Vorschrift, die uns erlaubt, den Wahrheitswert einer Formel in Abhängigkeit von den Wahrheitswerten ihrer Variablen zu berechnen.

Aufgabe 2

- *Wie lässt sich die Schlussfolgerung aus Beispiel 1 als aussagenlogische Formel formalisieren?*
- *Stelle eine Wahrheitstafel für die Formel*

$$\big(((A \rightarrow B) \wedge A) \rightarrow B\big)$$

auf. Welcher Wahrheitswert ergibt sich für die gesamte Formel in Abhängigkeit von der Belegung der Atome A und B?

Definition 3
- *Eine aussagenlogische Formel, deren Wahrheitswert bei jeder Belegung der Variablen* 1 *ist, heißt* ***Tautologie****.*
- *Eine aussagenlogische Formel, deren Wahrheitswert bei jeder Belegung der Variablen* 0 *ist, heißt* ***Kontradiktion****.*
- *Eine aussagenlogische Formel heißt* ***erfüllbar****, wenn es eine Belegung der Variablen gibt, bei der die Formel den Wahrheitswert* 1 *hat.*

Beispiel 8
- *Die Formel* $(A \vee \neg A)$ *ist eine Tautologie.*
- *Die Formel* $\neg(A \vee \neg A)$ *ist eine Kontradiktion.*
- *Die Formel* $\big(((A \to B) \wedge A) \to B\big)$ *ist eine Tautologie.*

Oft wird auch $\dot\vee$ verwendet, um die **ausschließende Disjunktion** („entweder ... oder..., aber nicht beides“) zu bezeichnen. $A \dot\vee B$ ist eine Abkürzung:

Definition 4 $A \dot\vee B := \big((A \vee B) \wedge \neg(A \wedge B)\big)$.[2]

Die ausschließende Disjunktion wird auch oft mit **XOR** bezeichnet (Engl.: exclusive or).

Aufgabe 3
1. *Stelle die Wahrheitstafel für* $A \dot\vee B$ *auf.*
2. *Lässt sich „*$\to$*“ auch definieren, indem man nur die Verknüpfungen „*$\vee$*“ und „*$\neg$*“ verwendet?*

Definition 5 *Seien* α *und* β *aussagenlogische Formeln. Sei* M *die Menge der Variablen, die in* α *vorkommen, und sei* N *die Menge der Variablen, die in* β *vorkommen. Die Formeln* α *und* β *heißen* ***äquivalent****, wenn für jede Belegung der Variablen in* $M \cup N$*, die Wahrheitswerte von* α *und* β *übereinstimmen. Wir schreiben dann auch „*$\alpha \equiv \beta$*“.*

Beispiel 9 *Sei* $\alpha := \big(A \wedge (A \vee B)\big)$ *und sei* $\beta := A$*. Sei* $N := \{A, B\}$ *die Menge der Variablen von* α*, und sei* $M := \{A\}$ *die Menge der Variablen von* β*. Dann sind für jede Belegung der Variablen von* $\{A, B\}$ *die Wahrheitswerte von* α *und* β *die gleichen, also gilt* $\alpha \equiv \beta$*. Das lässt sich z.B. mit einer Wahrheitstafel nachprüfen. Man stellt eine Tabelle für* α *und* β *auf und vergleicht Zeile für Zeile die Wahrheitswerte, die sich für* α *und* β *ergeben. Wenn sie in allen Zeilen übereinstimmen, dann sind die Formeln äquivalent (siehe Abbildung 7.1).*

[2] Das Zeichen „:=“ heißt: Das Symbol, das auf der Seite von „:=“ mit dem Doppelpunkt steht, wird neu definiert, und zwar als das, was auf der anderen Seite von „:=“ steht.

A	B	$\big(A \wedge (A \vee B)\big)$	A
0	0	0	0
0	1	0	0
1	0	1	1
1	1	1	1

Abbildung 7.1.: Wahrheitstafel zu Beispiel 9.

7.3. Logik und korrektes Argumentieren

Für korrektes Argumentieren (im Alltag und in mathematischen Beweisen) ist es hilfreich zu wissen, welche Folgerungen sich schon allein aufgrund der Form des Arguments ergeben – unabhängig von konkreten Inhalt. So hat bereits Aristoteles festgestellt:

Beispiel 10 *Prämisse 1: Für alle x gilt: wenn x Eigenschaft E hat, dann hat x auch Eigenschaft F.*
Prämisse 2: Das Objekt a hat Eigenschaft E.
Wenn Prämissen 1 und 2 gelten, dann gilt: Das Objekt a hat auch Eigenschaft F.

Das können wir nun nach Belieben mit Inhalt füllen, und die Aussage ist immer wahr. Wir können z.B. $E :=$„ist Mensch“ festsetzen, $F :=$ „ist sterblich“ und $a :=$Sokrates. Dann ergibt sich:

Prämisse 1: Für alle x gilt: wenn x ein Mensch ist, dann ist x sterblich.
Prämisse 2: Sokrates ist ein Mensch.
Wenn Prämissen 1 und 2 gelten, dann gilt: Sokrates ist sterblich.

Wir führen jetzt einige abkürzende Schreibweisen ein. Die Schreibweisen sind z. T. der Prädikatenlogik entlehnt, die Ihr in der Vorlesung *Diskrete Modellierung* kennen lernen werdet. Sie werden aber auch häufig einfach als abkürzende Schreibweisen in Argumentationen verwendet und sind hilfreich, um Gedanken und Argumente übersichtlich zu strukturieren. Dazu wollen wir sie hier auch verwenden.

- „$\forall$“ liest man als „für alle“. Das Zeichen $\forall$ heißt **Allquantor**. (Häufig sieht man auch die Abkürzung „f. a.“ für „für alle“.)
- „$\exists$“ liest man als „es gibt“ (oder gleichbedeutend „es existiert“). Das Zeichen $\exists$ heißt **Existenzquantor**.
- „$E(x)$“ liest man als „x hat die Eigenschaft E“.
- „$\wedge$“, „$\vee$“, „$\neg$“ haben die Bedeutungen „und“, „oder“ und „nicht“
- „ $\Longrightarrow$ “ liest man als „daraus folgt“, und „ $\Longleftrightarrow$ “ als „genau dann, wenn“.

Die Zeichen „$\forall$“ und „$\exists$“ nennt man auch **Quantoren**. Die Zeichen $\Longrightarrow$ und $\Longleftrightarrow$ erinnern natürlich an $\rightarrow$ und $\leftrightarrow$ aus der Aussagenlogik. Um umgangssprachliches Argumentieren nicht mit der Syntax der Aussagenlogik zu verwechseln, wollen wir für das umgangssprachliche Argumentieren die Zeichen $\Longrightarrow$ und $\Longleftrightarrow$ verwenden. Diese Trennung ist besonders dann wichtig, wenn man umgangssprachlich etwas über logische Formeln aussagen möchte. In diesem Fall würde man in der Umgangssprache auch auf die Abkürzungen $\wedge$, $\vee$, $\neg$, $\forall$ und $\exists$ verzichten, um eine Verwechslung zu vermeiden.

Beispiel 11 *Unter Verwendung der Abkürzungen würde Aristoteles' allgemeines Schema aus Beispiel 10 so aussehen:*

Prämisse 1: $\forall x(E(x) \Longrightarrow F(x))$
Prämisse 2: $E(a)$
Folgerung: $F(a)$.

Zusammengefasst: $\big(\forall x(E(x) \Longrightarrow F(x)) \wedge E(a)\big) \Longrightarrow F(a)$.

Mit Inhalt gefüllt sieht es so aus:
$\big(\forall x(\mathit{Mensch}(x) \Longrightarrow \mathit{sterblich}(x)) \wedge \mathit{Mensch}(\mathit{Sokrates})\big) \Longrightarrow \mathit{sterblich}(\mathit{Sokrates})$.

Aufgabe 4 *Formalisiere die folgenden umgangssprachlichen Aussagen soweit wie möglich, wobei Du die oben eingeführten Abkürzungen verwenden sollst.*

1. *Alle Menschen sind klug und schön.*
2. *Es gibt einen Dinosaurier, der fliegen kann.*
3. *Wenn der Schalter an ist, dann leuchtet entweder der rote oder der grüne Laser.*
4. *Für jede natürliche Zahl gibt es eine größere.*
5. *Es gibt eine Katze, die klug ist, aber nicht schön.*
6. *Es gibt einen Kandidaten, der von allen Wahlberechtigten gewählt wurde.*
7. *Ein Stern heißt genau dann Hesperus, wenn er Phosphorus heißt.*
8. *Es gibt ein kleinstes Element.*
9. *Wenn der Roboter das Tor sieht und den Ball hat, dann schießt er. Wenn er das Tor nicht sieht und den Ball hat, dann dreht er sich.*

8. Die Sprache der Mathematik

In diesem Kapitel werden wir die grundlegenden Begriffe *Mengen*, *Relationen* und *Funktionen* einführen und einige ihrer Eigenschaften kennenlernen. Diese Begriffe treten überall in der Informatik auf, und Sicherheit im Umgang mit ihnen ist deshalb sehr nützlich. Auch mathematische Beweise spielen in der Informatik eine wichtige Rolle, z.B. wenn man *beweisen* möchte, dass ein Algorithmus das Richtige tut. Ein **Beweis** einer Aussage gibt einem die 100%**ige Sicherheit**, dass die Aussage stimmt. So ist die Richtigkeit der Aussage „Algorithmus A ist korrekt“ in vielen Fällen sehr wichtig, etwa wenn A in der Flugzeugsteuerung zum Einsatz kommen soll, oder in medizinischen Geräten, bei Banktransaktionen, etc.

Eine 100%ige Sicherheit kann man nur erlangen, wenn man 100%ig **exakt** spezifiziert, was für Eigenschaften man haben möchte. Es darf dabei kein Zweifel möglich sein. Dafür eignet sich die Sprache der Mathematik hervorragend! Deshalb verwenden wir sie in der Informatik auch oft und gerne. Die Mathematik benutzt **Definitionen**, die genau festlegen, über welche Objekte und welche Eigenschaften wir sprechen. Sobald das geklärt ist, formulieren wir unter Verwendung der Definitionen unsere Aussage, die wir gerne beweisen möchten. So eine Aussage heißt in der Mathematik auch **Satz**. Sätze haben oft die Form:

„Unter den Voraussetzungen V hat das Objekt O die Eigenschaft E“.

Den Satz können wir dann versuchen zu **beweisen**: Wir nehmen an, dass die Voraussetzungen V erfüllt sind und zeigen dann durch eine Folge von richtigen Argumenten, dass unter diesen Voraussetzungen das Objekt O wirklich die Eigenschaft E hat. Dazu verwenden wir auch die Methoden der Logik, die wir im vorangegangenen Kapitel kennengelernt haben.

Der Struktur **Definition – Satz – Beweis** begegnet man beim exakten Argumentieren immer wieder. Um eine Definition richtig zu verstehen, macht man sich gerne **Beispiele**. Das wollen wir hier auch immer wieder tun.

8.1. Mengen

Definition 6 *Eine **Menge** ist eine Ansammlung verschiedener Objekte, auch **Elemente** genannt.*

Ist X eine Menge und z ein Element von X, so schreiben wir $z \in X$. Ist z kein Element von X, so schreiben wir $z \notin X$.

Beispiele 1

- *Die Menge aller Bücher.*

- *Die Menge aller Primzahlen.*
- *Die Menge aller Farne.*
- *Die Menge aller Legosteine.*
- *Die Menge aller Klavierstimmer.*
- *Die Menge aller Pixel.*
- *Die Menge* $M := \{$*Lucy, Paul, Sasha*$\}$
- $\emptyset := \{\} =$ *die leere Menge*
- $\mathbb{N} := \{0, 1, 2, 3, 4, \dots\} =$ *die Menge der natürlichen Zahlen.*
- $\mathbb{Z} := \{0, 1, -1, 2, -2, 3, -3, 4, -4 \dots\} =$ *die Menge der ganzen Zahlen.*
- $\mathbb{Q} := \{\frac{a}{b} \mid a, b \in \mathbb{Z}, b \neq 0\} =$ *die Menge der rationalen Zahlen.*
- $\mathbb{R} :=$ *Menge der reellen Zahlen.*
- $\mathbb{R}_{\geq 0} :=$ *Menge der nichtnegativen reellen Zahlen.*
- *Die Menge* $N := \{z \in \mathbb{Z} \mid$ *es gibt ein* $k \in \mathbb{Z}$ *so dass* $z = 3k\}$.

Definition 7 *Seien X und Y Mengen.*

1. *X und Y sind **gleich**, in Zeichen $X = Y$, falls X und Y dieselben Elemente enthalten.*
2. *X heißt **Teilmenge** von Y, in Zeichen $X \subseteq Y$, wenn jedes Element von X auch ein Element von Y ist.*
3. *X heißt **echte Teilmenge** von Y, in Zeichen $X \subsetneq Y$, wenn $X \subseteq Y$ ist, aber nicht $X = Y$ gilt.*

Beachte: Mengen sind ausschließlich durch ihre Elemente definiert. Es kommt nicht auf die Reihenfolge der Elemente an. So gilt z.B. $\{1, 2, 3\} = \{2, 1, 3\} = \{3, 2, 1\}$. Zudem „zählt“ ein Element nie mehrfach in derselben Menge: $\{1, 1, 2\} = \{1, 2\}$.

Satz 1 *Seien X, Y und Z Mengen, für die $X \subseteq Y$ und $Y \subseteq Z$ gilt. Dann gilt auch $X \subseteq Z$.*

Anschaulich mag es klar sein, dass Satz 1 gilt. Wir wollen den Satz aber nun *beweisen* – schließlich müssen wir sicherstellen, dass wir die Definitionen auch so gewählt haben, dass er gilt (dass also die Definitionen unsere Anschauung richtig *modellieren*)! Bei dem Beweis wollen wir also nicht unsere bildliche Anschauung als Argumentationsbasis nehmen, sondern nur unsere Definitionen.

Beweis von Satz 1. Wir nehmen an, dass X, Y und Z Mengen sind mit $X \subseteq Y$ und $Y \subseteq Z$. Nun wollen wir zeigen, dass dann auch $Y \subseteq Z$ gilt. Dazu müssen wir nach Definition 7.2 zeigen, dass **jedes** Element von X auch ein Element von Z ist. Sei also $x \in X$ ein beliebiges Element. Nach Voraussetzung ist $X \subseteq Y$, also gilt $x \in Y$ (nach Definition 7.2). Nach Voraussetzung ist auch $Y \subseteq Z$, also gilt $x \in Z$ (nach Definition 7.2). Da $x \in X$ beliebig gewählt war, gilt also für alle Elemente von X, dass sie auch Elemente von Z sind, also gilt $X \subseteq Z$, und das war zu zeigen. □

Der Beweis von Satz 1 ist ein typischer **direkter Beweis**: Unter der Annahme, dass die Voraussetzungen erfüllt sind, haben wir durch eine Folge von korrekten Argumenten bewiesen, dass dann die Behauptung gilt.

Tatsächlich ist der Argumentationsspielraum bei Beweisen sehr klein. *Jeder* Schritt muss begründet werden! Die Begründungen, die wir im Beweis von Satz 1 verwendet haben, sind
1. Anwendung der Voraussetzungen (hier: X und Y sind Mengen, $X \subseteq Y$ und $Y \subseteq Z$) und
2. Anwendung von Definitionen (hier: Definition 7.2).
Dazu kann man später dann noch
3. Anwendung bereits bewiesener Sätze
hinzunehmen.

Satz 2 *Seien X, Y und Z Mengen. Dann gilt*

1. *Wenn $X \subseteq Y$ und $Y \subsetneq Z$ gilt, dann gilt auch $X \subsetneq Z$.*
2. *$X = Y$ gilt genau dann, wenn $X \subseteq Y$ und $Y \subseteq X$ gelten.*

Beweis. Seien X, Y und Z Mengen.
Zu 1: Wir nehmen an, dass $X \subseteq Y$ und $Y \subsetneq Z$ gelten. Wir wollen nun zeigen, dass dann auch $X \subsetneq Z$ gilt. Nach Definition 7.3 gilt auch $Y \subseteq Z$. Damit können wir Satz 1 verwenden, und es folgt $Y \subseteq Z$.

Jetzt müssen wir noch zeigen, dass nicht $X = Z$ gilt. Dann sind wir fertig. Wäre $X = Z$, so wäre $X \subseteq Y \subseteq Z = X$, also wäre auch $Y = Z$. Das ist aber ein Widerspruch zur Voraussetzung $Y \subsetneq Z$. Also kann $X = Z$ nicht gelten, und damit sind wir fertig.

Zu 2: Für Aussagen von Typ „genau dann wenn“ müssen wir immer zwei Implikationen beweisen. Hier sind das:
„$\Rightarrow$“: „Wenn $X = Y$ gilt, dann gelten auch $X \subseteq Y$ und $Y \subseteq X$“, und
„$\Leftarrow$“: „Wenn $X \subseteq Y$ und $Y \subseteq X$ gelten, dann gilt auch $X = Y$“.
Beweis von „$\Rightarrow$“: Angenommen, es gilt $X = Y$. Dann enthalten X und Y nach Definition 7.1 dieselben Elemente. Also ist jedes Element von X auch Element von Y, also $X \subseteq Y$, und jedes Element von Y ist auch Element von X, also $Y \subseteq X$. Das war zu zeigen.
Beweis von „$\Leftarrow$“: Angenommen, es gelten $X \subseteq Y$ und $Y \subseteq X$. Dann ist jedes Element von X auch ein Element von Y und jedes Element von Y ist auch ein Element von X. Also enthalten X und Y dieselben Elemente, und somit $X = Y$. □

Der zweite Teil des Beweises von Satz 1.1 ist ein typischer **Beweis durch Widerspruch**. Man will eine Aussage A beweisen (im Beweis von Satz 1.1 die Aussage $X \neq Z$). Dazu

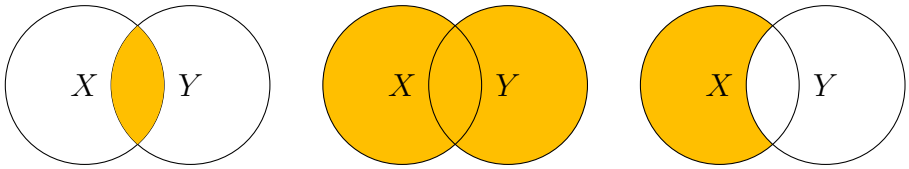


Abbildung 8.1.: Die Mengen $X \cap Y$, $Y \cup Y$ und $X \setminus Y$.

macht man die Annahme, dass $\neg A$ (das Gegenteil) wahr ist (hier: $X = Z$). Dann folgert man daraus durch korrekte Argumentation einen offensichtlichen Widerspruch (hier: $Y = Z$, ein Widerspruch zur Voraussetzung). Also war die Annahme $\neg A$ falsch, und somit muss die Aussage A selber wahr sein.

Aufgabe 5 *Seien X, Y und Z Mengen. Beweise: Wenn $X \subsetneq Y$ und $Y \subseteq Z$ gilt, dann gilt auch $Y \subsetneq Z$.*

Definition 8 *Seien X und Y Mengen.*

1. *Der **Schnitt** von X und Y ist die Menge*

$$X \cap Y := \{z \mid z \in X \text{ und } z \in Y\}.$$

2. *Die **Vereinigung** von X und Y ist die Menge*

$$X \cup Y := \{z \mid z \in X \text{ oder } z \in Y\}.$$

3. *Die **Differenz** von X und Y ist die Menge*

$$X \setminus Y := \{z \mid z \in X \text{ und } z \notin Y\}.$$

4. *X und Y heißen **disjunkt**, falls $X \cap Y = \emptyset$.*

Abbildung 8.1 zeigt Schnitt, Vereinigung und Differenz von Mengen.

Aufgabe 6 *Seien X, Y und Z Mengen. Beweise die folgenden Aussagen:*

1. *$X \cap X = X$ und $X \cup X = X$.*
2. *$X \cap Y = Y \cap X$ und $X \cup Y = Y \cup X$.*
3. *$X \cap (Y \cap Z) = (X \cap Y) \cap Z$ und $X \cup (Y \cup Z) = (X \cup Y) \cup Z$.*

4. $X \cap (X \cup Y) = X$ *und* $X \cup (X \cap Y) = X$.

5. $X \cap (Y \cup Z) = (X \cap Y) \cup (X \cap Z)$ *und* $X \cup (Y \cap Z) = (X \cup Y) \cap (X \cup Z)$.

6. *Die Mengen* $X \setminus Y$ *und* $Y \setminus X$ *sind disjunkt.*

Definition 9 *Eine Menge X heißt* ***endlich****, wenn X nur endlich viele Elemente enthält, d.h. wenn es eine Zahl $n \in \mathbb{N}$ gibt, so dass X genau n Elemente enthält. Die* ***Mächtigkeit*** *einer Menge X ist definiert als*

$$|X| := \begin{cases} \text{Anzahl der Elemente in } X, & \text{falls } X \text{ endlich ist} \\ \infty, & \text{sonst.} \end{cases}$$

Hierbei ist ∞ lediglich eine Abkürzung für „unendlich“.

Beispiel 12

- $|\{Lucy, Paul, Sasha\}| = 3$
- $|\emptyset| = 0$
- $|\{z \in \mathbb{Z} \mid \text{es gibt ein } k \in \mathbb{Z}, \text{ so dass } z = 3k\}| = \infty$

Was macht man eigentlich mit falschen Behauptungen? Man widerlegt sie! Das tut man, indem man ein Beispiel angibt, das zeigt, dass die Behauptung nicht gilt.

Beispiel 13 *Behauptung: Für alle Mengen X und Y gilt:*

$$|Y \cup Y| = |X| + |Y|\,.$$

Diese Behauptung ist falsch, z.B. gilt für die Mengen $\{1,2\}$ und $\{2\}$:

$$|\{1,2\} \cup \{2\}| = |\{1,2\}| = 2 \neq 3 = |\{1,2\}| + |\{2\}|.$$

Satz 3 (Summenregel) *Seien X und Y Mengen. Es gilt $|X \cup Y| = |X| + |Y|$ genau dann, wenn X und Y disjunkt sind.*

Beweis. Seien X und Y Mengen. „$\Rightarrow$“: Es gelte $|X \cup Y| = |X| + |Y|$. Wir müssen zeigen, dass dann X und Y disjunkt sind.

Wir führen den Beweis durch Widerspruch: Angenommen, X und Y sind nicht disjunkt. Dann ist $X \cap Y \neq \emptyset$. Sei also $a \in X \cap Y$. Dann zählt a einmal in $|X|$ und einmal in $|Y|$. Also trägt a genau 2 zu $|X| + |Y|$ bei, während a nur 1 zu $|X \cup Y|$ beiträgt. Jedes Element in $X \setminus Y$ trägt 1 zu $|X| + |Y|$ bei und ebenso 1 zu $|X \cup Y|$. Dasselbe gilt für jedes Element in $Y \setminus X$. Somit gilt also $|X| + |Y| > |X \cup Y|$ und insbesondere $|X| + |Y| \neq |X \cup Y|$, ein Widerspruch zur Voraussetzung. Also sind X und Y disjunkt.

„$\Leftarrow$“: Sei $X \cap Y = \emptyset$. Wir zeigen
1) $|X \cup Y| \leq |X| + |Y|$ und

2) $|X| + |Y| \leq |X \cup Y|$.
Dann folgt $|X \cup Y| = |X| + |Y|$ und wir sind fertig.

Zu 1): Sei $a \in X \cup Y$. Dann zählt a einmal in $|X \cup Y|$. Nach Definition der Vereinigung ist $a \in X$ oder $a \in Y$. Also wird a in $|X|$ oder in $|Y|$ gezählt. Somit gilt $|X \cup Y| \leq |X| + |Y|$.

Zu 2): Sei $a \in X$. Da X und Y disjunkt sind, ist $a \notin Y$. Also trägt a genau 1 zu $|X| + |Y|$ bei. Nach Definition der Vereinigung ist $a \in X \cup Y$. Also wird a in $|X \cup Y|$ auch einmal gezählt. Sei nun $b \in Y$. Wie eben gilt: Da X und Y disjunkt sind, ist $b \notin X$. Also trägt b genau 1 zu $|X| + |Y|$ bei. Nach Definition der Vereinigung ist auch $b \in X \cup Y$. Also wird b in $|X \cup Y|$ auch einmal gezählt. Somit gilt $|X| + |Y| \leq |X \cup Y|$. □

Die Summenregel wird oft wie folgt angewendet. Wir wollen die Mächtigkeit einer Menge Z bestimmen. Dazu klassifizieren wir die Elemente von Z nach Eigenschaften E_1 und E_2, die sich gegenseitig ausschließen:
$Z_1 = \{a \in Z \mid a \text{ hat Eigenschaft } E_1\}$ und $Z_2 = \{a \in Z \mid a \text{ hat Eigenschaft } E_2\}$ und jedes der Elemente in Z habe genau eine der beiden Eigenschaften, also $Z = Z_1 \cup Z_2$ und $Z_1 \cap Z_2 = \emptyset$. Angenommen, wir kennen $|Z_1|$ und $|Z_2|$. Dann können wir $|Z|$ als Summe von $|Z_1|$ und $|Z_2|$ berechnen.

Als Anwendung wollen wir die fundamentale Rekursion für Binomialkoeffizienten beweisen.

Definition 10 *Mit $\binom{n}{k}$ bezeichnen wir die Anzahl der k-elementigen Teilmengen einer n-elementigen Menge.*

Satz 4 (Rekursion für Binomialkoeffizienten) *Für alle $n \in \mathbb{N}$ mit $n \geq 1$ gilt:*

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

Beweis. Sei $n \geq 1$ und sei X eine n-elementige Menge. Wegen $n \geq 1$ enthält X mindestens ein Element a. Mit Y bezeichnen wir die Menge aller k-elementigen Teilmengen von X. Wir klassifizieren die k-elementigen Teilmengen von X danach, ob sie a enthalten oder nicht. Sei dazu $Y_1 := \{A \subseteq X \mid a \in A \text{ und } |A| = k\}$ und $Y_2 := \{B \subseteq X \mid a \notin B \text{ und } |B| = k\}$. Dann sind Y_1 und Y_2 disjunkt und es gilt $Y = Y_1 \cup Y_2$. Also gilt nach der Summenregel (Satz 3): $\binom{n}{k} = |Y| = |Y_1| + |Y_2|$. Was ist $|Y_1|$? was ist $|Y_2|$? Wir erhalten alle Mengen aus Y_1, indem wir alle $(k-1)$-elementigen Teilmengen von $X \setminus \{a\}$ um das Element a ergänzen. Also ist $|Y_1| = \binom{n-1}{k-1}$. Wir erhalten alle Mengen aus Y_2, indem einfach wir alle k-elementigen Teilmengen von $X \setminus \{a\}$ nehmen. Also ist $|Y_2| = \binom{n-1}{k}$. Somit folgt $\binom{n}{k} = |Y| = |Y_1| + |Y_2| = \binom{n-1}{k-1} + \binom{n-1}{k}$, und das war zu zeigen. □

Aufgabe 7 *Seien X und Y Mengen.*

1. *Beweise: $|X \cup Y| \leq |X| \cup |Y|$.*
2. *Wie groß ist $|X \cap Y|$ höchstens? Wie groß mindestens? Gib jeweils einen Beweis an.*

Aufgabe 8 (Mensasalat) *In der Mensa gibt es eine Salattheke, an der man sich den Salat selber kombinieren kann. Es gibt dort die folgenden Zutaten zur Auswahl:*

- *Eisbergsalat*
- *Feldsalat*
- *Tomate*
- *Paprika*
- *Gurke*
- *Bohnen*

Wieviele Möglichkeiten gibt es, daraus eine Salatmischung zusammenzustellen?

Definition 11 *Die* ***Potenzmenge*** *einer Menge X ist die Menge*

$$\mathcal{P}(X) := \{Y \mid Y \subseteq X\},$$

also die Menge aller Teilmengen von X.

Beispiel 14 *Die Potenzmenge von $\{$Lucy, Paul, Sasha$\}$ ist*

$$\begin{aligned}&\big\{\emptyset,\\ &\{Lucy\}, \{Paul\}, \{Sasha\},\\ &\{Lucy,\ Paul\}, \{Paul,\ Sasha\}, \{Lucy,\ Sasha\},\\ &\{Lucy,\ Paul,\ Sasha\}\big\}.\end{aligned}$$

8.2. Relationen

Definition 12

1. *Für beliebige Objekte a und b bezeichnet (a, b) das geordnete* ***Paar*** *mit Komponenten a und b.*
2. *Zwei geordnete Paare (a, b) und (c, d) sind gleich, $(a, b) = (c, d)$, falls $a = c$ und $b = d$ gilt.*
3. *Sei $n \in \mathbb{N}$. Für n Objekte $a_1, \ldots, a_n$ bezeichnet $(a_1, \ldots, a_n)$ das geordnete* ***Tupel*** *mit Komponenten a_1, $a_2 \ldots a_n$. Die Zahl n heißt die* ***Länge*** *des Tupels. Hat ein Tupel die Länge n, so sprechen wir auch von einem n-****Tupel****.*
4. *Sei $n, m \in \mathbb{N}$. Zwei Tupel $(a_1, \ldots, a_n)$ und $(a_1, \ldots, a_m)$ sind gleich, falls $n = m$ und $a_i = b_i$ für alle $i = 1, \ldots, n$ gilt.*

Geordnete Paare sind also 2-Tupel. Manchmal werden Tupel auch als (endliche) **Folgen** bezeichnet. Im Gegensatz zu Mengen spielt bei Tupeln die Reihenfolge eine wichtige Rolle. Zwei Tupel mit denselben Objekten, aber in unterschiedlicher Reihenfolge angeordnet, sind verschieden!

Beispiel 15
- *Es gilt* $(1,2,3) \neq (3,2,1)$*, aber* $\{1,2,3\} = \{3,2,1\}$
- *Es gilt* $(1,2,1) \neq (1,2)$*, aber* $\{1,2,1\} = \{1,2\}$

Manchmal ist die Reihenfolge für uns wichtig:

Beispiel 16 *Stell Dir vor, Du machst einen Interrail-Urlaub in Griechenland, und Du bist gerade in Athen. Du bist noch unentschlossen, wo Du als nächstes hinfahren möchtest. Auf Nachfrage bekommst Du die Auskunft, dass vom Hauptbahnhof von 7:00 bis 19:00 Uhr jede volle Stunde ein Zug fährt, und dass jeden Tag die Ziele Chalkida, Kalavrita, Markopoulo, Patras und Thessaloniki angefahren werden. Das hilft Dir zwar, Dein nächstes Reiseziel auszusuchen, aber hilfreicher wäre es gewesen, wenn man Dir die Ziele in der Reihenfolge der tatsächlichen Abfahrten gesagt hätte, etwa: Thessaloniki, Patras, Chalkida, Markopoulo, Thessaloniki, Thessaloniki, Patras, Markopoulo, Kalavrita, Thessaloniki, Patras, Chalkida, Markopoulo.*

Definition 13 *Sei* $n \in \mathbb{N}$ *und seien* $X_1, \ldots, X_n$ *Mengen. Das* ***kartesische Produkt*** *von* $X_1, \ldots, X_n$ *ist die Menge*

$$X_1 \times \ldots \times X_n := \{(a_1, \ldots, a_n) \mid a_i \in X_i\}.$$

Beispiel 17 *Sei* $X = \{1,2,3,4,5,6,7,8\}$ *und* $Y = \{a,b,c,d,e,f,g,h\}$*, dann ist* $X \times Y$ *die Menge aller Felder eines Schachbretts.*

Aufgabe 9
1. *Was ist das kartesische Produkt von* $\{$*Apfel, Birne, Mango*$\}$ *und* $\{$*getrocknet, frisch*$\}$*?*
2. *Sei* X *eine Menge. Was ist* $X \times \emptyset$*?*
3. *Finde weitere Beispiele für kartesische Produkte.*

Satz 5 (Produktregel) *Sei* $X = X_1 \times \ldots \times X_n$ *ein kartesisches Produkt. Dann gilt*

$$|X| = |X_1| \cdot \ldots \cdot |X_n| .$$

Beweis. Die Elemente von X haben die Form $(a_1, \ldots, a_n)$ mit $a_i \in X_i$. Wir haben $|X_1|$ Möglichkeiten a_1 zu wählen. Für jede dieser Möglichkeiten haben wir $|X_2|$ Möglichkeiten, a_2 zu wählen, etc. Also haben wir insgesamt $|X_1| \cdot \ldots \cdot |X_n|$ Möglichkeiten, Elemente der Form $(a_1, \ldots, a_n)$ zu wählen. □

Definition 14 *Ein* ***Bitvektor*** *ist ein Tupel aus 0en und 1en. Die Anzahl von 0en und 1en heißt* ***Länge*** *des Bitvektors.*

Beispiel 18 *Sei $n \in \mathbb{N}$: Wieviele Bitvektoren der Länge n gibt es? Das wollen wir uns mit Hilfe von Satz 5 überlegen. Sei dazu $X = \{0, 1\}$. Die Bitvektoren der Länge n sind genau die Elemente des n-fachen Produkts $\underbrace{X \times \ldots \times X}_{n \text{ Faktoren}}$. Nach Satz 5 ist $|X \times \ldots \times X| = \underbrace{|X| \cdot \ldots \cdot |X|}_{n \text{ Faktoren}} = \underbrace{2 \cdot \ldots \cdot 2}_{n \text{ Faktoren}} = 2^n$. Also gibt es genau 2^n solche Bitvektoren der Länge n.*

Aufgabe 10 *Es gibt drei Wege von Sparta nach Theben und fünf Wege von Theben nach Athen. Sei X die Menge der Wege von Sparta nach Korinth und Y die Menge der Wege von Korinth nach Athen. Alle Wege von Sparta nach Athen führen über Korinth. Was beschreibt $X \times Y$? Was beschreibt $|X \times Y|$?*

Definition 15 *Seien $X_1, \ldots, X_n$ Mengen. Eine* ***Relation*** *von $X_1, \ldots, X_n$ ist eine Teilmenge R von $X_1 \times \ldots \times X_n$. Die Zahl n heißt die* ***Stelligkeit*** *von R. Um die Stelligkeit zu betonen, sagen wir auch gelegentlich, dass R eine n-**stellige** Relation ist.*

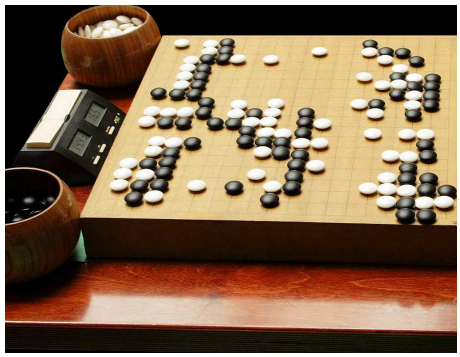


Abbildung 8.2.: Go-Spiel. `en.wikipedia.org/wiki/File:Go-Equipment-Narrow-Black.png`

Beispiel 19 *Sei $X = \{1, 2, 3, \ldots, 19\}$. Dann ist $X \times X$ die Menge aller Felder eines Go-Spielplans. (Abbildung 8.2 zeigt ein Go-Spiel.) Wir nehmen an, dass gerade ein Spiel gespielt wird. Bei jedem Zug ist die Menge der von Weiß (Schwarz) belegten Kreuzungen eine Relation von X und X.*

Beispiel 20 *Auf den reellen Zahlen ist „$\leq$“ die Relation*

$$\{(a, b) \mid a, b \in \mathbb{R},\ a \leq b\} \subseteq \mathbb{R} \times \mathbb{R}.$$

Aufgabe 11

1. *Erkläre, wie man die Eigenschaft „liegt zwischen“ als 3-stellige Relation von $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$ auffassen kann.*
2. *Finde weitere Beispiele für Relationen.*

8.3. Funktionen

Definition 16 *Seien X und Y Mengen. Eine **Funktion von** X **nach** Y ist eine Relation $f \subseteq X \times Y$ mit der Eigenschaft, dass für **jedes** Element $a \in X$ **genau ein** Element $b \in Y$ existiert mit $(a, b) \in f$.*

Aufgabe 12 *Welche der folgenden Relationen sind Funktionen?*

1. $\{(m, n) \in \mathbb{N} \times \mathbb{Z} \mid m \text{ ist Teiler von } n\}$
2. $\{(m, n) \in \mathbb{N} \times \mathbb{N} \mid n = 2m^3 + m^2 + 1\}$
3. $\{(m, n) \in \mathbb{Z} \times \mathbb{N} \mid m \text{ ist gerade und } n = 2m\}$
4. $\{(x, y) \in \mathbb{R} \times \mathbb{R} \mid x^2 + y^2 = 1\}$
5. $\{((x, y), z) \in (\mathbb{R} \times \mathbb{R}) \times \mathbb{R} \mid z = x\}$

Definition 17 *Seien X und Y Mengen.*

- *Wir schreiben auch $f\colon X \to Y$ um auszudrücken, dass f eine Funktion **von** X **nach** Y ist.*
- *Ist $f\colon X \to Y$ eine Funktion und ist $a \in X$, so schreiben wir auch $f(a) = b$ für das eindeutig bestimmte $b \in Y$ mit $(a, b) \in f$.*
- *Ist $f\colon A \to B$ eine Funktion, so heißt die zugehörige Relation*

  $$\{(a, f(a)) \mid a \in A\} \subseteq A \times B$$

  *auch der **Graph der Funktion** f.*
- *Ist $f\colon X \to Y$ eine Funktion, so heißt X **Definitionsbereich** von f und Y heißt **Bildbereich** von f.*

Abbildung 8.3 zeigt die Graphen der Funktionen $f, g, h\colon [0, 2] \to [0, 8]$, gegeben durch $f(x) = e^2$, $g(x) = x^2$ und $h(x) = 1$. (Hierbei ist für zwei reelle Zahlen x und y die Menge $[x, y] := \{z \in \mathbb{R} \mid x \leq z \leq y\}$.)

Was müssen wir also alles angeben, wenn wir eine Funktion angeben wollen? 1. den Definitionsbereich, 2. den Bildbereich, und 3. eine Relation f von Definitionsbereich und Bildbereich mit der Eigenschaft, dass für **jedes** Element $a \in X$ **genau ein** Element $b \in Y$ existiert mit $(a, b) \in f$. Letzteres geschieht oft durch eine **Zuordnungsvorschrift**.

Definition 18 *Eine **Zuordnungsvorschrift** ist eine Vorschrift, die jedem Element einer Menge X genau ein Element einer Menge Y zuordnet.*

Beispiel 21 (Funktionen)

- *$f_1\colon \mathbb{N} \to \mathbb{N}$, und für alle $n \in \mathbb{N}$ ist $f_1(n) = n^2$.*

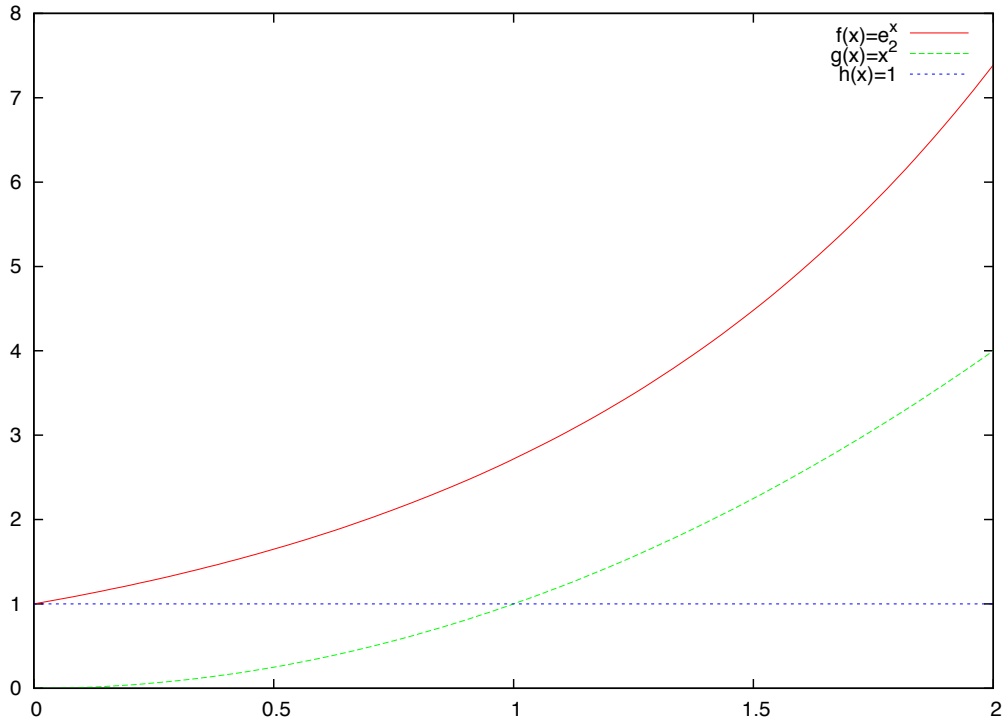


Abbildung 8.3.: Die Graphen der Funktionen $f(x) = e^2$, $g(x) = x^2$ und $h(x) = 1$.

- $f_2\colon \mathbb{Z} \to \mathbb{N}$, *und für alle* $z \in \mathbb{Z}$ *ist* $f_2(z) = 3$.
- $f_3\colon \mathbb{R} \to \mathbb{N}$, *und für alle* $x \in \mathbb{R}$ *ist* $f_3(x) = x^2$.
- $\text{alter}\colon \{m \mid m \text{ ist Mensch}\} \to \mathbb{N}$, *für alle Menschen* m *ist* $\text{alter}(m)$ *das Alter von* m *in Jahren.*

Beachte, dass f_1 und f_3 aus Beispiel 21 verschieden sind (warum?). Oft verwendet man eine verkürzte Schreibweise, etwa

$$f\colon \mathbb{N} \to \mathbb{N},\ f(n) = n^2$$

oder

$$\begin{aligned} f\colon \mathbb{N} &\to \mathbb{N} \\ n &\mapsto n^2 \end{aligned}$$

Aufgabe 13 *Finde weitere Beispiele für Funktionen.*

Wann sind zwei Funktionen gleich? Wann ist eine Funktion kleiner als die andere?

Definition 19 *Seien* X *und* Y *Mengen, und seien* $f, g\colon X \to Y$ *Funktionen.*

- *Die Funktionen* f *und* g *sind* ***gleich****, wenn für alle* $x \in X$ *gilt:* $f(x) = g(x)$.
- *Seien* $X, Y \subseteq \mathbb{R}$. *Die Funktion* f *ist* ***kleiner oder gleich*** g*, in Zeichen* $f \leq g$*, wenn für alle* $x \in X$ *gilt:* $f(x) \leq g(x)$.
  Die Funktion f *ist* ***kleiner*** *als* g*, in Zeichen* $f < g$*, wenn für alle* $x \in X$ *gilt:* $f(x) < g(x)$.
- *„*$f \geq g$*“ und „*$f > g$*“ sind analog definiert.*

Um Funktionen überhaupt vergleichen zu können, müssen sie also denselben Definitionsbereich und denselben Bildbereich haben, denn für die anderen Fälle haben wir das nicht definiert.

Beispiel 22 *Für die Funktionen* f, g *und* h *aus Abbildung 8.3 gilt* $f > g$, $g < f$, $h \leq f$ *und* $f \geq h$.

Aufgabe 14 *Gib die vollständigen Definitionen für „*$f \geq g$*“ und „*$f > g$*“ an.*

Aufgabe 15 *Für welche zwei der folgenden Funktionen gilt* $=, \leq, \geq, >$, *oder* $<$*?*

- $f\colon \mathbb{N} \to \mathbb{R},\ f(n) = n - 2$
- $g\colon \mathbb{N} \to \mathbb{R},\ g(n) = \log_2(n)$
- $h\colon \mathbb{N} \to \mathbb{Z},\ h(n) = n - 2$

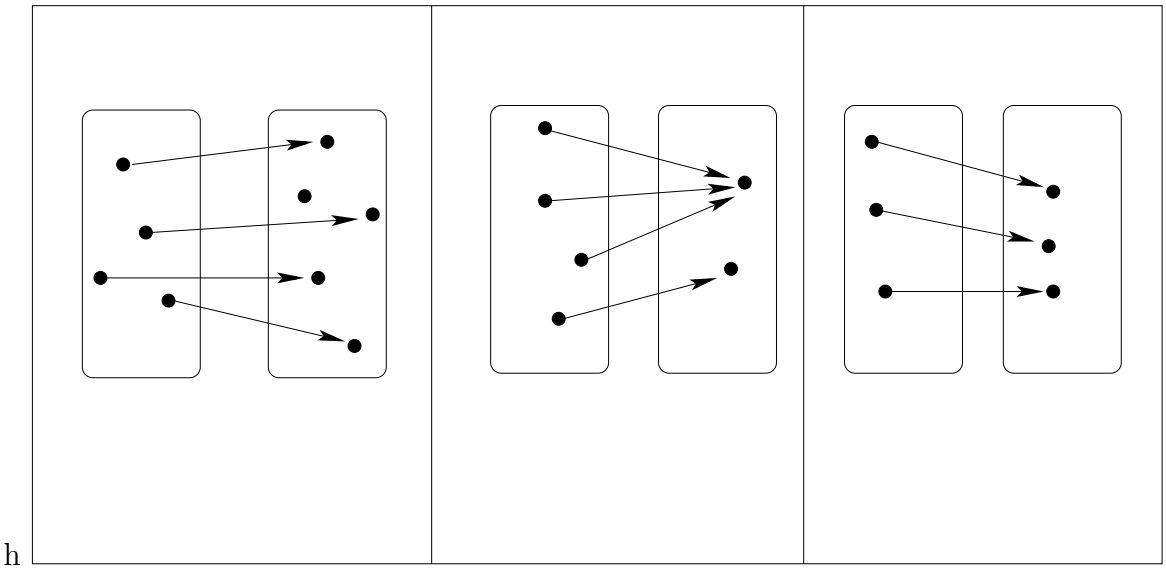


Abbildung 8.4.: Funktion f ist injektiv, g ist surjektiv und h ist bijektiv.

- $i\colon \mathbb{N} \to \mathbb{R}$, $i(n) = n^2 + n - 2 - n^2$
- $j\colon \mathbb{N} \to \mathbb{R}$, $j(n) = n$
- $k\colon \mathbb{N} \to \mathbb{R}$, $k(n) = 2^n$

Definition 20 (Eigenschaften von Funktionen) *Sei $f\colon X \to Y$ eine Funktion.*

1. *f heißt **injektiv**, falls es für jedes $b \in Y$* höchstens *ein $a \in X$ gibt mit $f(a) = b$.*
2. *f heißt **surjektiv**, falls es für jedes $b \in Y$* mindestens *ein $a \in X$ gibt mit $f(a) = b$.*
3. *f heißt **bijektiv**, falls es für jedes $b \in Y$* genau *ein $a \in X$ gibt mit $f(a) = b$.*

In Abbildung 8.4 werden die drei Begriffe veranschaulicht.

Beispiel 23
- *Die Funktion $g\colon \mathbb{R} \to \mathbb{R}$, $g(x) = e^x$ ist injektiv, aber nicht surjektiv und nicht bijektiv.*
- *Die Funktion $h\colon \mathbb{R} \to \mathbb{R}_{\geq 0}$, $h(x) = x^2$, ist surjektiv, aber nicht injektiv und nicht bijektiv.*
- *Die Funktion $f\colon \mathbb{R} \to \mathbb{R}$, $f(x) = x - 2$, ist bijektiv.*

Satz 6 *1. Für jede Funktion $f\colon X \to Y$ gilt*

$$f \text{ ist bijektiv} \iff f \text{ ist injektiv und surjektiv.}$$

2. Seien X und Y endliche Mengen. Dann gilt

$$|X| = |Y| \iff \text{es gibt eine bijektive Funktion } f\colon X \to Y.$$

Beweis. Zu Teil 1: Sei $f\colon X \to Y$ eine Funktion.
„$\Rightarrow$“: Ist f bijektiv, so gilt nach Definition 20, dass es für jedes $b \in Y$ genau ein $a \in X$ mit $f(a) = b$.

Dann gibt es insbesondere für jedes $b \in Y$ höchstens ein $a \in X$ mit $f(a) = b$, also ist f injektiv.

Zudem gibt es insbesondere für jedes $b \in Y$ mindestens ein $a \in X$ mit $f(a) = b$, also ist f auch surjektiv.

„$\Leftarrow$“: Ist f gleichzeitig injektiv und surjektiv, so gibt es für jedes $b \in Y$ höchstens ein $a \in X$ mit $f(a) = b$ (nach Definition 20.1) und gleichzeitig gibt es für jedes $b \in Y$ mindestens ein $a \in X$ mit $f(a) = b$ (nach Definition 20.2). Also gibt es für jedes $b \in Y$ genau ein $a \in X$ mit $f(a) = b$, und somit ist f bijektiv.

Zu Teil 2: Seien X und Y endliche Mengen.
„$\Rightarrow$“: Sei $n \in \mathbb{N}$ die natürliche Zahl mit $|X| = |Y| = n$. Wir müssen nun eine bijektive Funktion von X nach Y angeben. Numeriere die Elemente von X durch, etwa $X = \{a_1, \dots a_n\}$, und nummeriere die Elemente von Y durch, etwa $Y = \{b_1, \dots b_n\}$. Dann ist die Funktion $f\colon X \to Y$, gegeben durch $f(a_i) = b_i$ (für alle $i = 1, \dots, n$), bijektiv.

„$\Leftarrow$“: Sei $f\colon X \to Y$ bijektiv. Wir müssen nun zeigen, dass X und Y gleich viele Elemente haben. Dazu zählen wir die Elemente von X durch, indem wir an das i-te Element die Zahl i schreiben. Wenn wir an das Element $a \in X$ die Nummer i schreiben, dann schreiben wir gleichzeitig an das Element $f(a) \in Y$ auch die Nummer i. Weil f injektiv ist, erhält kein Element von Y mehr als eine Nummer. Weil f surjektiv ist, erhält jedes Element von Y eine Nummer. Damit erhält jedes Element genau eine der Nummern, die auch in X vergeben wurden. Also hat Y genauso viele Elemente wie X.

□

Eine typische Anwendung von Satz 6.2 ist die folgende: Wir wollen Mächtigkeit einer Menge X bestimmen. Können wir X bijektiv auf eine Menge Y abbilden, so ist $|X| = |Y|$ und es reicht, die Mächtigkeit von $|Y|$ zu kennen.

Beispiel 24 *Sei X eine endliche Menge. Was ist die Mächtigkeit der Potenzmenge $\mathcal{P}(X)$? Sei dazu $n := |X|$. Wir nummerieren die Elemente von X durch, etwa $X = \{a_1, \dots, a_n\}$. Sei Y die Menge aller Bitvektoren der Länge n. Nach Beispiel 18 wissen wir, dass $|Y| = 2^n$ ist. Wir definieren jetzt eine Funktion $f\colon \mathcal{P}(X) \to Y$ wie folgt: Für eine Teilmenge $Z \subseteq X$ ist $f(Z)$ derjenige Bitvektor der Länge n, der an der i-ten Stelle eine 1 stehen hat, falls $a_i \in Z$ ist, und eine 0 sonst. Man muss sich jetzt noch überlegen, dass f bijektiv ist. Damit wissen wir dann, dass $|\mathcal{P}(X)| = 2^n$ ist.*

Aufgabe 16 *Beweise, dass die Funktion f aus Beispiel 24 bijektiv ist.*

9. Asymptotische Laufzeitanalyse

9.1. Beispiel: Fahrplanauskunft

In der Informatik werden Algorithmen (Programme) untersucht. Was möchten wir über die Algorithmen wissen? Stelle Dir vor, Du hast gerade für eine große Stadt ein U-Bahn-Netz mit 16 Linien entworfen. Nun benötigst Du ein Programm für die Fahrplanauskunft. Drei Anbieter bieten Dir dafür die Programme P_1, P_2 und P_3 an, und Du sollst eines auswählen.

Folgendes Wissen ist für uns wichtig:

- Wieviel Rechenzeit brauchen die Programme, um eine Fahrplanauskunft zu geben?
- Sind die Programme korrekt: rechnet der Algorithmus richtig, oder übersieht er etwa Verbindungen?

In diesem Kapitel wollen wir uns mit dem ersten Kriterium beschäftigen. Um die Rechenzeit eines Programms anzugeben, wollen wir, grob gesagt, die maximale Anzahl der Rechenschritte bzw. Befehle abschätzen, die ein Programm ausführen muss, um aus der Anfrage die Antwort zu berechnen.

Wir nehmen an, dass ein Rechner einen Rechenschritt in 10^{-6} Sekunden, d.h. in einer millionstel Sekunde ausführen kann. Programm P_1 braucht bei n Bahnlinien n^2 Rechenschritte für die Berechnung einer Fahrplanauskunft. Programm P_2 braucht bei n Bahnlinien n^3 Rechenschritte, und Programm P_3 braucht 2^n Rechenschritte. Für welches Programm entscheidest Du sich?

Anzahl n der Bahnlinien	$P_1\colon n^2$	$P_2\colon n^3$	$P_3\colon 2^n$
16	256	4.096	65.536

Nun stelle Dir vor, Du möchtest in die Fahrplanauskunft gerne noch die vorhandenen 16 Straßenbahnlinien mit aufnehmen. Dann musst Du insgesamt 32 Linien berücksichtigen:

Anzahl n der Linien	$P_1\colon n^2$	$P_2\colon n^3$	$P_3\colon 2^n$
16	256	4.096	65.536
32	1.024	32.768	$\geq 4 \cdot 10^9$

Bei 16 Linien braucht Programm P_2 also etwas weniger als eine 200stel Sekunde. D.h. wenn 200 Personen gleichzeitig eine Fahrplanauskunft haben möchten, dann müssen sie weniger als eine Sekunde warten.

Bei 32 Linien braucht Programm P_2 mehr als eine Zehntelsekunde. Wenn also 100 Personen gleichzeitig eine Anfrage stellen, müssen sie mehr als 10 Sekunden warten. Programm P_3 braucht mehr als eine Stunde!

Aufgabe 17 *Stell Dir vor, Du möchtest in die Fahrplanauskunft gerne zusätzlich noch die vorhandenen* 32 *Buslinien mit aufnehmen. Mit U-Bahn, Straßenbahn und Bus zusammen gibt es dann 48 Linien. Setze die Tabelle fort, indem Du die Anzahl der Schritte für die drei Programme bei* 48 *Linien einträgst, und schätze ab, wie lange man bei den Programmen auf eine Antwort warten muss (hierbei nehmen wir weiterhin an, dass unser Rechner einen Rechenschritt in* 10^{-6} *Sekunden ausführt).*

Bei *wenigen* Linien ist es relativ egal, welches Programm wir verwenden: wir müssen nie sehr lange warten. Bei vielen Linien macht es einen großen Unterschied!

Wenn wir die Laufzeit von Programmen analysieren, interessieren wir uns also vor allem für **große** Eingaben. An den großen Eingaben (hier bei $n = 32$) sieht man, dass Programm P_3 ein wahrer „Laufzeitfresser“ ist, wohingegen P_1 noch eine ganz vernünftige Laufzeit hat. (Wir sprechen deshalb auch von der **asymptotischen** Laufzeitmessung: uns interessiert die Laufzeit für wachsende Eingaben $n \to \infty$.)

In Wirklichkeit hängt die Laufzeit eines Programms nicht nur von der Größe der Eingabe ab, sondern auch von der Rechenleistung des Rechners, von der verwendeten Programmiersprache und vom Compiler. Wir wollen aber die Laufzeit nicht für jeden Rechner, jede Programmiersprache und jeden Compiler einzeln berechnen, denn das wäre uns zu mühsam. Wir begnügen uns deshalb mit einer ungefähren Laufzeitbestimmung. Was **ungefähr** heißt, werden wir noch mathematisch exakt fassen. Die Idee ist, dass wir beispielsweise zwischen zwei Rechnern, von denen der eine doppelt so schnell ist wie der andere, nicht unterscheiden wollen. Deshalb wollen wir auch nicht unterscheiden, ob ein Programm die Laufzeit $f(n) = n^2$ oder die Laufzeit $g(n) = 2 \cdot n^2$ hat. Das wäre unnötige Genauigkeit an der falschen Stelle. Wir sagen in diesem Fall einfach, dass beide Programme **quadratische** Laufzeit haben (das nennen wir später Laufzeit $\Theta(n^2)$).

Allgemeiner: sei $c \in \mathbb{R}$ eine konstante Zahl und sei $f\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ eine Funktion. Wir wollen zwischen der Laufzeit f und $c \cdot f$ nicht unterscheiden.[1] Diese Laufzeiten behandeln wir als „gleich“ – wir tun die Funktionen zusammen in eine Klasse (die genaue Definition hierfür werden wir im folgenden Abschnitt kennenlernen, in Definition 22.2).

Wir interessieren uns aber sehr dafür, ob die Laufzeit eines Programmes gegenüber der des anderen viel schlechter ist, also im Vergleich „abhaut“. An unserem Beispiel haben wir gesehen, dass die Laufzeit von P_3 im Vergleich zur Laufzeit von P_1 viel schlechter ist (sie „haut uns im Vergleich ab“). Auch im Vergleich zu P_2 ist die Laufzeit von P_3 schlechter.

Aufgabe 18 *Die folgenden Funktionen stellen Laufzeiten von sechs verschiedenen Algorithmen dar. Zeichne die Graphen der Funktionen* $f_1, \ldots, f_6$.

1. $f_1\colon \mathbb{N}_{>0} \to \mathbb{R}_{\geq 0}$, $f_1(n) = \log_2 n$
2. $f_2\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$, $f_2(n) = \sqrt{n}$
3. $f_3\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$, $f_3(n) = n$

[1] Erinnerung: Seien $X, Y \subseteq \mathbb{R}$, und sei $c \in \mathbb{R}$. Ist $f\colon X \to Y$ eine Funktion, so ist $c \cdot f\colon X \to Y$ die Funktion, die $r \in X$ auf $c \cdot f(r)$ abbildet.

4. $f_4 \colon \mathbb{N} \to \mathbb{R}_{\geq 0},\ f_4(n) = n^3$
5. $f_5 \colon \mathbb{N} \to \mathbb{R}_{\geq 0},\ f_5(n) = 2^n$
6. $f_6 \colon \mathbb{N} \to \mathbb{R}_{\geq 0},\ f_6(n) = n!$

Wie ändern sich die Funktionswerte der sechs Funktionen, wenn man n verdoppelt?

9.2. Asymptotische Laufzeitanalyse

Bei der Laufzeitmessung

- ... interessieren wir uns nicht dafür, was bei kleinen Eingabelängen geschieht
- ... behandeln wir Funktionen als „gleich“, die sich nur um einen konstanten Faktor unterscheiden

Tatsächlich sind zwei Funktionen, die sich um einen konstanten Faktor $\neq 1$ unterscheiden, natürlich nicht gleich (vgl. Definition 19). Wir wollen sie aber in die gleiche Klasse von Funktionen tun. Ebenso wollen wir die Relation $f \leq g$ (und $f \geq g$) etwas aufweichen: wir interessieren uns für Funktionen f, die fast überall kleiner gleich (größer gleich) der Funktion g sind – bis auf einen konstanten Faktor.

Wir wollen dies nun in einer exakten Definition fassen.

Definition 21 *Seien $f, g \colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ Funktionen, die einer **Eingabelänge** $n \in \mathbb{N}$ eine nicht-negative **Laufzeit** $f(n)$, bzw. $g(n)$ zuweisen.*

1. *Es gilt $f \in \mathcal{O}(g)$ (lies: f ist in **groß-Oh** von g), wenn es eine positive Konstante $c \in \mathbb{R}_{>0}$ und eine natürliche Zahl $n_0 \in \mathbb{N}$ gibt, so dass*

$$f(n) \leq c \cdot g(n)$$

für alle $n \geq n_0$ gilt.
*Wir sagen dann auch: f **wächst höchstens so schnell wie** g.*

Beispiel 25 *Seien $f, g, h \colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ Funktionen, gegeben durch $f(n) = n$, $g(n) = n^2$ und $h(n) = n + 3$. Dann gilt*

1. *$f \in \mathcal{O}(g)$,*
2. *$h \in \mathcal{O}(g)$.*

Beweis von 1: Wähle als Konstante $c := 1$ und wähle $n_0 := 0$. Dann gilt für alle $n \geq n_0$: $f(n) = n \leq 1 \cdot n^2 = 1 \cdot g(n)$. Also ist $f \in \mathcal{O}(g)$, und das war zu zeigen.

Beweis von 2: Wähle als Konstante $c := 1$ und wähle $n_0 := 3$. Dann gilt für alle $n \geq n_0$: $f(n) = n \leq 1 \cdot n^2 = 1 \cdot g(n)$. Also ist $h \in \mathcal{O}(g)$, und das war zu zeigen.

Definition 22 *Seien $f, g\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ Funktionen, die einer Eingabelänge $n \in \mathbb{N}$ eine nicht-negative Laufzeit $f(n)$, bzw. $g(n)$ zuweisen.*

1. *Es gilt $f \in \Omega(g)$ (lies: f ist in **groß-Omega** von g), wenn $g \in \mathcal{O}(f)$.*
   *Wir sagen dann auch: f **wächst mindestens so schnell wie** g.*

2. *Es gilt $f \in \Theta(g)$ (lies: f ist in **Theta** von g), genau dann, wenn $f \in \mathcal{O}(g)$ und $g \in \mathcal{O}(f)$.*
   *Wir sagen dann auch: f und g **wachsen gleich schnell**.*

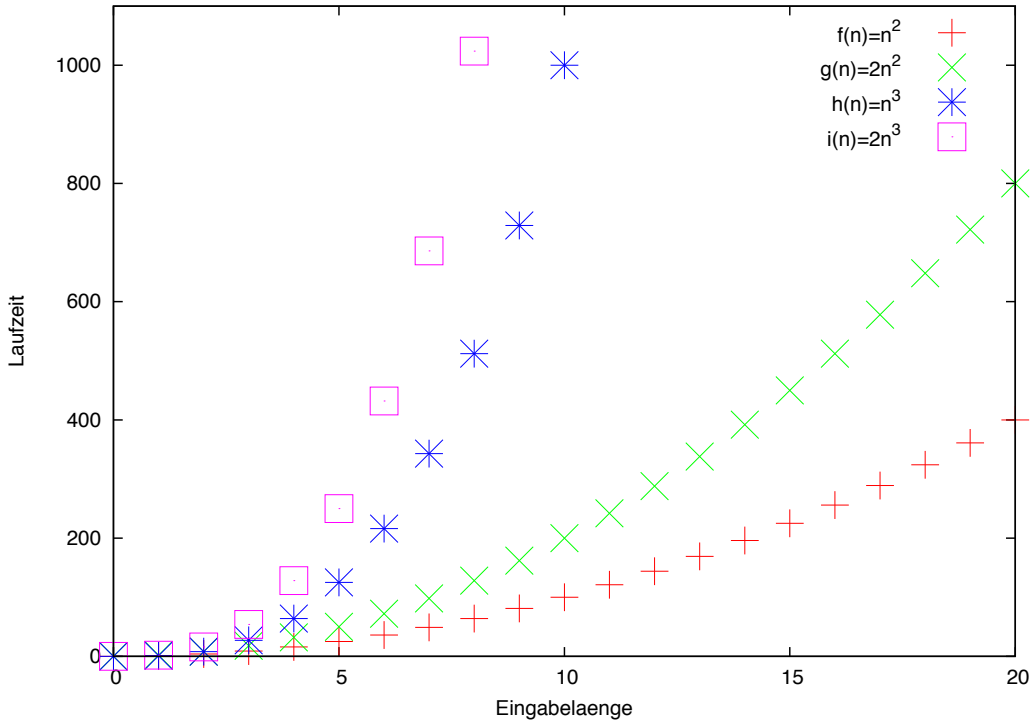


Abbildung 9.1.: Die Abbildung zeigt die Funktionen $f, g, h, i\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$. Die Funktionen $f(n) = n^2$ und $g(n) = 2n^2$ wachsen asymptotisch gleich schnell. Die Funktion $h(n) = n^3$ (und ebenso $i(n) = 2n^3$) wächst deutlich schneller als $f(n)$ und als $g(n)$. Die Funktionen $h(n) = n^3$ und $i(n) = 2n^3$ wachsen asymptotisch gleich schnell.

Im letzten Fall heißen f und g **asymptotisch gleich**. Sie sind in derselben Klasse. Abbildung 9.1 zeigt einige Beispiele.

Aufgabe 19 *Für ein Problem stehen zwei verschiedene Algorithmen A und B zur Verfügung. Algorithmus A hat die Laufzeit $f\colon \mathbb{N} \to \mathbb{N}$ mit $f(n) = n + 5$, und Algorithmus B hat die Laufzeit $g\colon \mathbb{N} \to \mathbb{N}$ mit $g(n) = n^2 + 1$. Du musst Dich für einen Algorithmus entscheiden. Welchen würdest Du wählen? Warum?*

Wie sieht man aber einem Programmcode an, welche Laufzeit das Programm hat? Im Praxisteil der Kurses habt Ihr schon an zwei Beispielen die Anzahl der Zuweisungen in Abhängigkeit von der Eingabelänge gezählt. Wir wollen uns nun die Beispiele noch einmal anschauen und deren asymptotische Laufzeit bestimmen.

Beispiel 26 *Wir zählen die Anweisungen, die das Programm aus Listing 9.1 in Abhängigkeit von der Eingabelänge* $n :=$ `meine_list_laenge` *ausführt. Anschließend bestimmen wir die asymptotische Laufzeit.*

In Zeilen 4 bis 6 stehen drei Zuweisungen. Die Schleife in Zeilen 7 bis 12 wird so oft durchlaufen, bis `fertig` *gilt. Dies ist der Fall, so lange weder das Ende der List erreicht wurde, noch die gesuchte Zahl gefunden wurde. Schlimmstenfalls steht die gesuchte Zahl ganz am Ende der Liste. Schlimmstenfalls durchlaufen wir die Schleife also so oft, wie die Liste lang ist, also* n *Mal, und in der Schleife werden jedes Mal sechs Anweisungen ausgeführt. Spätestens beim* $(n+1)$*-ten Mal ist dann in mit Zeile 7 die Suche beendet.*

Also ist $g(n)$*, die gesamte Anzahl der ausgeführten Anweisungen in Abhängigkeit von* n*, gegeben durch* $g(n) = 3 + n \cdot 6 + 1 = 6n + 4$*.*

Nun machen wir eine asymptotische Laufzeitanalyse. Sei dazu $f \colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ *die Abbildung* $n \mapsto n$*.*
Behauptung: Es gilt $g \in \Theta(f)$*. Dazu müssen wir nach Definition 22 zeigen, dass*
1) $g \in \mathcal{O}(f)$ *und*
2) $f \in \mathcal{O}(g)$
gilt.
Zu 1: Wähle als Konstante $c := 7$ *und wähle* $n_0 := 4$*. Dann gilt für alle* $n \geq n_0$*:* $g(n) = 6n + 4 \leq 7n$*. Also ist* $g \in \mathcal{O}(f)$*. Zu 2: Da* $f \leq g$ *gilt, folgt auch* $f \in \mathcal{O}(g)$*.*
Insgesamt haben wir bewiesen, dass $g \in \Theta(f)$ *gilt. Man sagt in diesem Fall auch, dass der Algorithmus eine* ***lineare*** *Laufzeit in* n *hat.*

```
1  # meine_liste ist eine liste mit nichtnegativen zahlen drin.
2  # meine_liste_laenge ist die länge dieser liste
3  # gesuchte_zahl ist die Zahl, nach der gesucht wird
4  hier_bin_ich = 0
5  gefunden = False
6  fertig = hier_bin_ich == meine_liste_laenge
7  while not fertig:
8    aktueller_eintrag = meine_liste[hier_bin_ich]
9    gefunden = aktueller_eintrag == gesuchte_zahl
10   fertig = hier_bin_ich == meine_liste_laenge
11   fertig = fertig or gefunden
12   hier_bin_ich = hier_bin_ich + 1
```

Listing 9.1: Lookup Naiv

Auch das nächste, schwerere Beispiel ist aus Abschnitt 6.2 bekannt.

Beispiel 27 *Betrachten wir die binäre Suche aus Listing 9.2. Wir haben uns überlegt, daß die Suche in einer sortierten Liste der Länge n unsere Rechnenmaschine höchstens $b(n) := (5 + 14 \cdot \log_2(n))\mu$s beschäftigen wird. Es gilt $b \in \mathcal{O}(\ln)$: Hierzu reicht es, sich zu überlegen, dass wann immer $n > \mathrm{e}^5$ ist, die Ungleichung $5 + 14 \cdot \log_2(n) < 25 \cdot \ln(n)$ gilt. Letztere Ungleichung ist aber äquivalent zu $(5 + 14 \cdot \log_2(n))/ln(n) < 25$, was bestimmt wahr ist, falls $5/\ln(\mathrm{e}^5) + 14 \cdot \log_2(n)/\log_\mathrm{e}(n) < 25$ gilt. Ziehen wir in Erwägung, dass der rechte Summand unabhängig von n immer kleiner ist als $\log_2(\mathrm{e})$, sehen wir das aber ganz leicht, denn $1 + 14 \cdot \log_2(\mathrm{e}) < 25$.*

```
1  # meine_liste ist eine aufsteigend sortierte liste nichtnegativer zahlen
2  # gesuchte_zahl ist eine zahl
3  meine_liste_laenge = len(meine_liste)
4  unterer_rand = 0
5  oberer_rand = meine_liste_laenge - 1
6  gefunden = False
7  fertig = False
8  while ( not gefunden ):
9      mitte = (oberer_rand + unterer_rand) / 2
10     aktueller_eintrag = meine_liste[mitte]
11     gefunden = aktueller_eintrag == gesuchte_zahl
12     if gefunden:
13         break
14     if oberer_rand == unterer_rand:
15         break
16     if aktueller_eintrag < gesuchte_zahl:
17         unterer_rand = mitte + 1
18     else:
19         if aktueller_eintrag > gesuchte_zahl:
20             oberer_rand = mitte - 1
21         else:
22             gefunden = True
```

Listing 9.2: Binäres Suchen

10. Induktion und Rekursion

10.1. Vollständige Induktion

Vollständige Induktion ist eine Beweistechnik, die sehr praktisch ist, um Eigenschaften für alle natürlichen Zahlen nachzuweisen. Das sind typischerweise Aussagen der Form „für alle $n \in \mathbb{N}$ gilt $E(n)$“, wobei E irgendeine Eigenschaft ist.

Um zu zeigen, dass alle natürlichen Zahlen n eine Eigenschaft E haben, reicht es zu zeigen, dass 0 die Eigenschaft E hat, und dass sich die Eigenschaft von 0 auf 1, von 1 auf 2, von 2 auf 3, etc. überträgt. Dann überträgt sie sich auf alle natürlichen Zahlen.

Üblicherweise besteht ein Induktionsbeweis aus zwei Teilen, dem **Induktionsanfang** und dem **Induktionsschritt**. Im Induktionsanfang wird nachgewiesen, dass 0 die Eigenschaft E hat. Im Induktionsschritt zeigen wir dann für alle $n \geq 0$: *Wenn* n die Eigenschaft E hat, dann hat auch $n+1$ die Eigenschaft E. Damit sind wir dann fertig. Im Induktionsschritt nehmen wir also an, dass n schon die Eigenschaft E hat (das ist die sogenannte **Induktionsvoraussetzung**), und zeigen, dass sich unter dieser Voraussetzung die Eigenschaft auch auf $n+1$ überträgt (das ist die sogenannte **Induktionsbehauptung**). Da wir im Induktionsanfang die Eigenschaft schon für den Startwert 0 nachgewiesen haben, wissen wir nach dem Induktionsschritt dann, dass sie sie auf 1, 2, ... immer weiter überträgt. Deshalb gilt sie dann für alle n. Mit unserer formalen Notation können wir das Induktionsprinzip auch folgendermaßen zusammenfassen: Wenn $E(0)$ und $\forall n\big((E(n) \implies E(n+1)\big)$, dann gilt $\forall n E(n)$. Oder noch kompakter:

$$\big(E(0) \wedge \forall n : \big(E(n) \implies E(n+1)\big)\big) \implies \forall n E(n).$$

Manchmal wollen wir eine Aussage $E(n)$ nur für alle natürlichen Zahlen $n \geq 1$ oder für alle $n \geq 2$ oder .... Dann fangen wir statt bei 0 einfach bei 1, 2 oder unserem Wunschstartwert an.

Bevor wir uns den dersten Beweis mit vollständiger Induktion anschauen, wollen wir zunächst noch zwei abkürzende Schreibweisen für Summen und Produkte einführen.

Definition 23 (Summen und Produkte) *Sei $n \in \mathbb{N}$, $n \neq 0$. Seien $a_1, \ldots, a_n$ beliebige Zahlen. Dann ist*

- $\Sigma_{i=1}^{n} a_i := a_1 + \ldots + a_n$, *und*
- $\Pi_{i=1}^{n} a_i := a_1 \cdot \ldots \cdot a_n$.

Zudem ist festgelegt, dass die leere Summe (d.h. die Summe ohne Summanden) gleich 0 *ist und dass das leere Produkt (d.h. das Produkt ohne Faktoren) gleich* 1 *ist. Also gilt:*

- $\Sigma_{i=1}^{0} a_i = 0$, *und*
- $\Pi_{i=1}^{0} a_i = 1$.

Beispiel 28 *Es gilt*

1. $\Sigma_{i=1}^{10} i = 1+2+3+4+5+6+7+8+9+10 = 55$,
2. $\Sigma_{i=1}^{10} 1 = 1+1+1+1+1+1+1+1+1+1 = 10$,
3. $\Pi_{i=1}^{5} i = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$, *und*
4. $\Pi_{i=1}^{5} 2 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 32$.

Summen sind sehr hilfreich, um die Anzahl von Anweisungen in Schleifen zu bestimmen. Das folgende Beispiel zeigt wie das geht.

```
1 # n ist eine positive natürliche Zahl
2 j = 0
3 i = 1
4 bedingung = i <= n
5 while( bedingung )
6     j = i+1
7     i = i+1
8     bedingung = i <= n
```

Listing 10.1: Schleife

Beispiel 29 *Wir zählen die Anweisungen, die das Programm aus Listing 10.1 in Abhängigkeit von n ausführt. Anschließend bestimmen wir die asymptotische Laufzeit.*

In Zeilen 1 bis 4 stehen zusammen drei Zuweisungen. Dann werden die Zeilen 5 bis 8 (die Schleife) n Mal ausgeführt. Dabei werden $\Sigma_{i=1}^{n} a_i$ Anweisungen ausgeführt, wobei der i-te Summand a_i die Anzahl der Zuweisungen im i-ten Durchlauf der Schleife ist. Bei jedem Durchlauf der Schleife werden vier Befehle ausgeführt (die Befehle in Zeilen 5 bis 8). Somit ist $a_i = 4$ für alle $i \in \{1, \ldots, n\}$. Ganz zum Schluss wird noch einmal Zeile 5 ausgeführt (die Schleifenbedingung ist dann nicht mehr erfüllt).

Also ist $g(n)$, die gesamte Anzahl der ausgeführten Anweisungen in Abhängigkeit von n, gegeben durch

$$g(n) = 3 + \Sigma_{i=1}^{n} 4 + 1 = 4 \cdot n + 4.$$

Sei $f\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ mit $f(n) = n$. Dann gilt $g \in \Theta(f)$. Der Algorithmus hat also lineare Laufzeit (vgl. Beispiel 26).

Den folgenden Satz werden wir nun mit vollständiger Induktion beweisen.

Satz 7 (Kleiner Gauß) *Für alle Zahlen $n \in \mathbb{N}$ gilt:*

$$\Sigma_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Beweis.
Induktionsanfang. Behauptung: Der Satz gilt für $n = 1$.
Beweis der Behauptung: $\Sigma_{i=1}^{1} i = 1 = \frac{1(1+1)}{2}$, und das war zu zeigen.
Induktionsschritt.
Induktionsvoraussetzung: Der Satz gilt bereits für n, es gilt also $\Sigma_{i=1}^{n} i = \frac{n(n+1)}{2}$. Unter dieser Voraussetzung müssen wir zeigen, dass er auch für $n+1$ gilt, also ist die
Induktionsbehauptung: Es gilt $\Sigma_{i=1}^{n+1} i = \frac{(n+1)\big((n+1)+1\big)}{2}$.
Beweis der Induktionsbehauptung:

$$\begin{aligned}
\Sigma_{i=1}^{n+1} i &= \Sigma_{i=1}^{n} i + (n+1) \\
&= \frac{n(n+1)}{2} + (n+1) \\
&= \frac{n(n+1) + 2(n+1)}{2} \\
&= \frac{(n+1)\big((n+1)+1\big)}{2},
\end{aligned}$$

und das war zu zeigen.
Bei der zweiten Gleichung haben wir die Induktionsvoraussetzung verwendet (wir haben $\Sigma_{i=1}^{n} i$ durch $\frac{n(n+1)}{2}$ ersetzt), alles andere sind Umformungen. □

Der Name „Kleiner Gauß“ basiert auf einer Gegebenheit aus der Grundschulzeit des Carl Friedrich Gauß: „Gauss besuchte zuerst 1784, nachdem er sein siebentes Lebensjahr zurückgelegt, die Catharinen-Volksschule, in welcher der erste Elementar-Unterricht ertheilt wurde und die damals unter der Leitung eines gewissen Büttner gestanden hat. [...] In dieser Schule, die noch sehr den Zuschnitt des Mittelalters gehabt zu haben scheint, blieb der junge Gauss zwei Jahre lang ohne durch etwas Ausserordentliches aufzufallen. Erst nach jener Zeit brachte es der Gang des Unterrichts mit sich, dass er auch in die Rechenklasse eintrat [...] Der junge Gauss war kaum in die Rechenklasse eingetreten, als Büttner die Summation einer arithmetischen Reihe aufgab. Die Aufgabe war indess kaum ausgesprochen als Gauss die Tafel mit den im niedern Braunschweiger Dialekt gesprochenen Worten auf den Tisch wirft: »Ligget se'.«(Da liegt sie.) Während die andern Schüler emsig weiter rechnen, multipliciren und addiren, geht Büttner sich seiner Würde bewußt auf und ab, indem er nur von Zeit zu Zeit einen mitleidigen und sarcastischen Blick auf den kleinsten der Schüler wirft, der seine Aufgabe beendigt hatte. Dieser sass dagegen ruhig, schon eben so sehr von dem festen unerschütterlichen Bewusstsein durchdrungen, welches ihn bis zum Ende seiner Tage bei jeder vollendeten Arbeit erfüllte, dass seine Aufgabe richtig gelöst sei, und dass das Resultat kein anderes sein könne. Am Ende der Stunde wurden die Rechentafeln umgekehrt; die von Gauss mit einer einzigen Zahl lag oben und als Büttner

das Exempel prüfte, wurde das seinige zum Staunen aller Anwesenden als richtig befunden, während viele der übrigen falsch waren und alsbald mit der Karwatsche[1] rectificiert wurden."(W. Sartorius von Waltershausen, Gauss zum Gedächtnis, Leipzig, 1856).

Satz 8 (Summe ungerader Zahlen) *Beweise mit vollständiger Induktion, dass für alle* $n \in \mathbb{N}$ *gilt:* $\Sigma_{i=1}^{n}(2i-1) = n^2$.

Beweis.
Induktionsanfang. Behauptung: Der Satz gilt für $n = 1$.
Beweis der Behauptung: $\Sigma_{i=1}^{1}(2i-1) = 2 \cdot 1 - 1 = 1^2$, und das war zu zeigen.
Induktionsschritt.
Induktionsvoraussetzung: Der Satz gilt bereits für n, es gilt also $\Sigma_{i=1}^{n}(2i-1) = n^2$. Unter dieser Voraussetzung müssen wir zeigen, dass er auch für $n+1$ gilt.
Induktionsbehauptung: Es gilt $\Sigma_{i=1}^{n+1}(2i-1) = (n+1)^2$.
Beweis der Induktionsbehauptung:

$$\begin{aligned}\Sigma_{i=1}^{n+1}(2i-1) &= \Sigma_{i=1}^{n}(2i-1) + 2 \cdot (n+1) - 1 \\ &= n^2 + 2 \cdot (n+1) - 1 \\ &= n^2 + 2 \cdot n + 1 \\ &= (n+1)^2,\end{aligned}$$

und das war zu zeigen.
Bei der zweiten Gleichung haben wir die Induktionsvoraussetzung verwendet (wir haben $\Sigma_{i=1}^{n}(2i-1)$ durch n^2 ersetzt), alles andere sind Umformungen. □

10.2. Rekursion

Rekursion ist eine Technik, eine Funktion mittels sich selbst zu definieren. Typischerweise geben wir für kleine Eingaben die Funktionswerte konkret an, und für die größeren Eingaben geben wir dann eine Vorschrift an, wie wir deren Funktionswerte mit Hilfe der Funktionswerte der kleineren Eingaben berechnen.

Beispiel 30 *Wir definieren den Begriff* **Vorfahren** *rekursiv:*

- *Die Eltern sind Vorfahren, und*
- *die Eltern der Vorfahren sind wieder Vorfahren.*

Rekursion ist eine elegante Technik um Funktionen und Mengen zu definieren. In Abschnitt 5.1 haben wie die Fakultätsfunktion kennengelernt als diejenige Funktion, die uns die Anzahl der Möglichkeiten, n Matrikelnummern auf n Neulinge zu verteilen, liefert. Wir führen die Funktion jetzt formal korrekt mittels einer rekursiven Definition ein.

[1] aus Lederriemen geflochtene Peitsche mit kurzem Holzstiel

Definition 24 *Die Funktion* $f\colon \mathbb{N} \to \mathbb{N}$*, gegeben durch die Vorschrift*

$$f(n) := \begin{cases} 1, & \text{falls } n = 0 \\ n \cdot f(n-1), & \text{sonst,} \end{cases}$$

heißt ***Fakultätsfunktion****. Man schreibt für* $f(n)$ *auch* $n!$*.*

Was sagt uns diese Vorschrift? Wenn wir $f(0)$ berechnen wollen, dann ist $n = 0$ und wir sind im ersten Fall und können den Funktionswert $f(0) = 1$ einfach aus der ersten Zeile der Definition ablesen. Wenn wir $f(3)$ berechnen wollen, dann ist also $n = 3$ und wir sind im zweiten Fall. Wir setzen gemäß der zweiten Zeile der Definition ein: $f(3) = 3 \cdot f(2)$. Um das weiter auszurechnen, müssen wir nun $f(2)$ ausrechnen. Nun ist also $n = 2$. Wieder sind wir im zweiten Fall und setzen ein: $f(2) = 2 \cdot f(1)$, wieder sind wir im zweiten Fall und setzen ein: $f(1) = 1 \cdot f(0)$. Super – mit $f(0)$ sind wir beim Basisfall angelangt, den wir einfach in der ersten Zeile ablesen können. Insgesamt ergibt sich also

$$f(3) = 3 \cdot f(2) = 3 \cdot \big(2 \cdot f(1)\big) = 3 \cdot \big(2 \cdot \big(1 \cdot f(0)\big)\big) = 3 \cdot 2 \cdot 1 \cdot 1 = 6.$$

Was haben wir gemacht, um $f(3)$ zu berechnen? Wir haben solange den zweiten Teil der Definition rekursiv eingesetzt (für immer kleinere Zahlen $4, 3, 2, 1$), bis wir im Basisfall $f(0)$ angelangt sind, dessen Funktionswert wir direkt am ersten Teil der Definition ablesen können. Diesen konnten wir dann einsetzen um $f(3)$ auszurechnen. Genau so würde auch der Computer eine rekursive Funktion berechnen. In Python sieht das Programm so aus:

```
1 def fakultaet(n):
2     if( n<=1 ):
3         return 1
4     return( n*fakultaet( n-1 ) )
```

Listing 10.2: Funktion in python zur Berechnung der Fakultät in python

Aufgabe 20 *Berechne* $f(5)$ *analog zum obigen Beispiel.*

Satz 9 *Sei* f *die Fakultätsfunktion. Es gilt* $f(n) = \Pi_{i=1}^{n} i$*.*

Beweis. Wir benutzen vollständige Induktion.
Induktionsanfang. Behauptung: Der Satz gilt für $n = 0$.
Beweis der Behauptung: Nach Definition 24 ist $f(0) = 1$. Andererseits ist das Produkt $\Pi_{i=1}^{0} i$ leer, und somit gilt $\Pi_{i=1}^{0} = 1$ nach Definition 23. Also ist $f(0) = 1 = \Pi_{i=1}^{0} i$, und das war zu zeigen.
Induktionsschritt.
Induktionsvoraussetzung: Der Satz gilt bereits für n, es gilt also $f(n) = \Pi_{i=1}^{n} i$. Unter dieser Voraussetzung müssen wir zeigen, dass er auch für $n + 1$ gilt, also ist die

Induktionsbehauptung: Es gilt $f(n+1) = \Pi_{i=1}^{n+1} i$.
Beweis der Induktionsbehauptung:

$$\begin{aligned} f(n+1) &\overset{\text{Def. } 24}{=} (n+1) \cdot f(n) \\ &= (n+1) \cdot \Pi_{i=1}^{n} i \\ &= \Pi_{i=1}^{n+1} i, \end{aligned}$$

und das war zu zeigen.
Bei der zweiten Gleichung haben wir die Induktionsvoraussetzung verwendet: Wir haben $f(n)$ durch $\Pi_{i=1}^{n} i$ ersetzt. Alles andere sind Umformungen. □

Aufgabe 21 *Gib eine rekursive Funktion* $\text{sum}\colon \mathbb{N} \to \mathbb{N}$ *an, die bei Eingabe* $n \in \mathbb{N}$ *die Summe der Zahlen* 0 *bis* n *berechnet. Gib auch den Programmcode in python an.*

Ein bekannter Witz zur Erläuterung des Begriffs *Rekursion* lautet

Rekursion
siehe *Rekursion*.

Solche unendlichen Regresse (Endlosschleifen) wollen wir allerdings vermeiden – wir müssen wissen, wo wir anfangen sollen – wir brauchen immer Basisfälle, die auch früher oder später erreicht werden. Dass manchmal mehrere Basisfälle nötig sind, sieht man an der Fibonacci-Funktion: dort haben wir die zwei Basisfälle $n = 0$ und $n = 1$.

Definition 25 *Die Funktion* $\text{fib}\colon \mathbb{N} \to \mathbb{N}$*, gegeben durch die Vorschrift*

$$\text{fib}(n) := \begin{cases} 0, & \textit{falls } n = 0, \\ 1, & \textit{falls } n = 1, \\ \text{fib}(n-1) + \text{fib}(n-2), & \textit{sonst,} \end{cases}$$

heißt ***Fibonacci-Funktion****.*

Definition 25 ist die rekursive Definition der Funktion aus Abschnitt 5.2, die uns zu einer Anzahl n von Monaten die Anzahl der Kaninchen liefert.

Aufgabe 22 *Berechne* $\text{fib}(0)$*,* $\text{fib}(1)$*,* $\text{fib}(2)$*,* $\text{fib}(3)$ *und* $\text{fib}(4)$*.*