Skript

Vorkurs Informatik

Wintersemester 2020/21

Sandra Kiefer
Rafael Franzke
Mario Holldack
Ronja Düffel
Stand: 18. Oktober 2020

Inhaltsverzeichnis

1. Programmieren und Programmiersprachen **7**
1.1. Programmiersprachen . . . 8
1.1.1. Deklarative Programmiersprachen . . . 8
1.1.2. Imperative Programmiersprachen . . . 8
1.2. Python . . . 9
1.2.1. IDLE . . . 10
IDLE Editor . . . 13

2. Grundlagen der Programmierung in Python **15**
2.1. Datentypen . . . 15
2.1.1. Wahrheitswerte . . . 15
2.1.2. Zahlen . . . 15
2.1.3. Zeichenketten . . . 17
2.2. Built-in-Funktionen . . . 18
2.2.1. Typumwandlung . . . 19
2.2.2. Benutzereingabe und Bildschirmausgabe . . . 20
2.3. Variablen . . . 21
2.4. Kommentare und Dokumenation . . . 23
2.5. Kontrollstrukturen . . . 24
2.5.1. Verzweigung . . . 25
if-Anweisung . . . 25
if . . . else Anweisung . . . 27
elif-Anweisung . . . 28
Mehrfachverzweigungen . . . 29
2.5.2. Schleifen . . . 32
while-Schleife . . . 32
for-Schleife . . . 35
2.5.3. Schleifen-Kontrollanweisungen . . . 37
2.6. Listen . . . 40
Zugriff auf Werte in einer Liste . . . 40
Werte in Listen verändern . . . 40
Werte aus Liste löschen . . . 41
2.7. Funktionen . . . 44
2.8. Gültigkeitsbereiche . . . 48
2.9. Module . . . 50
`import`-Anweisung . . . 50
`from...import`-Anweisung . . . 52
2.10. Datei Ein- und Ausgabe . . . 53

3. Rekursive Programmierung **59**
3.1. Rekursive Berechnung der Summe . . . 59
3.2. Fibonacci-Zahlen . . . 61

4. Debugging **63**
4.1. Syntaxfehler . . . 63
4.1.1. Typische Syntaxfehler in Python . . . 64

4.2. Semantikfehler 69
4.2.1. Typische Semantikfehler 71
TypeError 71
NameError 72
ValueError 73
4.3. Logische Fehler 74
4.3.1. Häufige Anfängerfehler 75
4.4. Debugging mit `print()` 75

5. Objektorientierte Programmierung **79**

6. Einführung **91**

7. Aussagenlogik **93**
7.1. Einführung 93
7.2. Aussagenlogik 96
7.2.1. Syntax der Aussagenlogik 97
7.2.2. Semantik der Aussagenlogik 97
7.2.3. Erfüllbarkeit, Allgemeingültigkeit und Äquivalenz 99
7.2.4. Fundamentale Rechenregeln 100

8. Mengen **103**

9. Relationen **115**

10. Funktionen **119**

11. Beweistechniken **127**
11.1. Direkter Beweis 127
11.1.1. Abgeschlossenheit einer Zahlenmenge bezüglich einer Verknüpfung 129
11.2. Indirekter Beweis / Beweis durch Kontraposition 130
11.3. Beweis durch Widerspruch 131

12. Induktion und Rekursion **133**
12.1. Vollständige Induktion 133
12.1.1. Wann kann man vollständige Induktion anwenden? 136
12.1.2. Was kann schief gehen? 136
12.2. Rekursion 137
12.2.1. Wozu Rekursion? 139
Türme von Hanoi 141

A. Appendix **145**
A.1. Unix und Linux 145
A.1.1. Dateien und Verzeichnisse 146
A.1.2. Login und Shell 147
A.1.3. Befehle 147
A.1.4. History und Autovervollständigung 150

B. Kochbuch **151**
B.1. Erste Schritte 151
B.2. Remote Login 154
B.2.1. Unix-artige Betriebssysteme (Linux, MacOS, etc) 155
B.2.2. Windows 156

C. Zum Lösen von Übungsaufgaben **159**
C.1. Neu im Studium 159
C.1.1. Wozu soll ich die Übungsaufgaben überhaupt (selbst) machen? 159
C.1.2. Was halten wir fest? 160
C.1.3. Konkrete Tipps 160
C.2. Wie geht man an eine Aufgabe in theoretischer Informatik heran? 161
C.2.1. Die Ausgangssituation 161
C.2.2. Konkret: Wie legen wir los? 162
C.2.3. Wann können wir zufrieden sein? 163
C.3. Von der Idee zum fertigen Programm 164
C.3.1. Was ist die Ausgangssituation? 164
C.3.2. Schritt für Schritt 164
C.4. Wie geht man ein größeres Programmierprojekt an? 166
C.4.1. Zerlegung des Problems in Teilprobleme 167
C.4.2. Präzisierung der Teilprobleme 168
C.4.3. Entwickeln einer Lösungsidee für die Teilprobleme und das Gesamtproblem 169
C.4.4. Entwurf eines groben Programmgerüsts 170
C.4.5. Schrittweise Verfeinerung 170
C.4.6. Umsetzung in eine Programmiersprache 171
C.4.7. Testen und Korrigieren von Fehlern 171

1. Programmieren und Programmiersprachen

In diesem Kapitel gehen wir kurz darauf ein, was der Vorgang des Programmierens eigentlich beinhaltet, was eine Programmiersprache ist und wie Programmiersprachen unterschiedliche Lösungsstrategien unterstützen. Anschließend werden wir die Entwicklungsumgebung IDLE für die Programmiersprache Python vorstellen. Genauere Details zum Programmieren in Python werden erst im nächsten Kapitel erläutert.

Auch wenn es für den Laien vielleicht so aussehen mag, Programmieren besteht nicht nur daraus Zeichenfolgen in einen Computer einzutippen. Vielmehr ist das lediglich der letzte Schritt eines Prozesses, dessen denkintensivster Teil nicht aus dem Eintippen von Zeichenfolgen besteht. Computerprogramme werden für den Einsatz „im wirklichen Leben“ geschrieben. Das heißt, die Anweisungen und Anforderungen, die ein Programmierer erhält, sind in der Regel in menschlicher Sprache formuliert. Ein Computer ist jedoch, wie der Name schon sagt, eine Rechenmaschine. Zugegebenermaßen eine sehr schnelle, präzise und mächtige Rechenmaschine, aber dennoch nur eine Rechenmaschine. D.h. ein Computer kann arithmetische und logische Operationen ausführen und den Zustand seines Speichers verändern, aber nicht die Bedeutung (*Semantik*) menschlicher Sprache verstehen. Dass es sprachgesteuerte Maschinen gibt, ist der Software zu verdanken, die Sprachimpulse in Maschinenbefehle übersetzt.

Semantik

Ein *ausführbares* Computerprogramm ist eine Folge von *Maschinencodebefehlen*, auch *Maschinenprogramm* genannt. Ein einzelner Maschinencodebefehl ist eine Operation, die der Prozessor direkt ausführen kann (z.B. Addieren zweier Zahlen, Vergleich zweier Werte, Lesen oder Beschreiben eines Speicherregisters). Die Ausführung eines Programms im Sinne des Computers besteht darin, die Folge von Maschinencodebefehlen nacheinander abzuarbeiten und dabei den Speicher zu verändern. Häufig spricht man auch vom „Zustand“ des Rechners und meint damit die gesamte Speicherbelegung. Um ein ausführbares Computerprogramm zu erstellen, muss also letztendlich das in menschlicher Sprache formulierte Problem in Maschinensprache übersetzen werden.

ausführbares Programm

Maschinenprogramm

Ein Maschinenprogramm ist jedoch lediglich eine Folge von 0 und 1 und für einen menschlichen Programmierer schwer zu erstellen und zu verstehen. Daher gibt es sog. *höhere Programmiersprachen*, die es erlauben, für Menschen besser lesbare und verständliche Programme zu erstellen. Dieser für Menschen lesbare Text wird auch *Quelltext* oder *Quellcode* (engl.: *source code*) genannt. Damit der Computer die, in der höheren Programmiersprache geschriebenen, Anweisungen im Quelltext ausführen kann, muss entweder ein *Compiler* oder ein *Interpreter* verwendet werden. Ein Compiler ist ein Übersetzungsprogramm, welches den Quelltext in ein Maschinenprogramm übersetzt. Ein Interpreter ist ein Programm, welches die Anweisungen im Quelltext schrittweise ausführt. Es interpretiert die Anweisungen einer höheren Progammiersprache. Während der Übersetzung bzw. Ausführung, wird auch überprüft, ob es sich bei dem Quelltext um ein zulässiges, d.h. gültiges Programm handelt. Ist dies nicht der Fall, dann sollte eine entsprechende Fehlermeldung ausgegeben werden.

höhere Programmiersprachen

Quellcode

Compiler

Interpreter

Zwar sind höhere Programmiersprachen für Menschen besser verständlich als Maschinenbefehle, allerdings sind sie noch weit entfernt von menschlichen Sprachen. Begriffe, Konstrukte und Befehle in menschlicher Sprache enthalten immer Interpretationsspielraum. Ein Computer benötigt aber eindeutige Handlungsanweisungen. Programmieren bedeutet also, eine Lösung für das gestellte Problem zu finden und diese Lösung in einer höheren Programmiersprache zu implementieren. Hierfür werden folgende Schritte, häufig auch mehr als einmal, durchlaufen.

1. Zerlegung des Problems in Teilprobleme
2. Präzisierung der Teilprobleme
3. Entwickeln einer Lösungsidee für die Teilprobleme und das Gesamtproblem
4. Entwurf eines groben Programmgerüsts
5. Schrittweise Verfeinerung
6. Umsetzung in eine Programmiersprache
7. Testen und Korrigieren von Fehlern

Das, was viele gemeinhin als Programmieren bezeichnen, geschieht demnach frühestens an 6. Stelle. Also sein Sie nicht frustriert, wenn Sie nach 20 Minuten noch kein bisschen Code geschrieben haben.

1.1. Programmiersprachen

Es gibt unzählige höhere Programmiersprachen. Jenachdem welches Programmierkonzept die Struktur und Vorgaben der Sprache dabei unterstützen, unterscheidet man in *deklarative* und *imperative* Progammiersprachen.

1.1.1. Deklarative Programmiersprachen

„Deklarativ" stammt vom lateinischen Wort „declarare", was soviel heißt wie „beschreiben" oder auch „erklären". Programme in deklarativen Programmierspachen beschreiben das Ergebnis des Programms. Sie erklären vielmehr *was* berechnet werden soll, als *wie* das Ergebnis berechnet werden soll. Sie bestehen aus, häufig mathematischen, Ausdrücken, die ausgewertet werden. Die Auswertung des arithmetischen Ausdrucks $(7 \cdot 5 + 3 \cdot 8)$ beispielsweise erfolgt, wie aus der Schule bekannt, über die schrittweise Auswertung der Teilausdrücke. $(7{\cdot}5{+}3{\cdot}8) = (35{+}3{\cdot}8) = (35{+}24) = 59$.

logische Programmiersprachen

Deklarative Programmiersprachen lassen sich aufteilen in *logische* und *funktionale* Programmiersprachen. Programme in logischen Programmiersprachen bestehen aus logischen Formeln und Aussagen, aus denen mithilfe logischer Schlussfolgerungen neue Aussagen hergeleitet werden. Die wohl bekannteste logische Programmiersprache ist *Prolog*.

funktionale Programmiersprachen

Programme in funktionalen Programmiersprachen bestehen aus Funktionsdefinitionen im engeren mathematischen Sinn und, evtl. selbstdefinierten, Datentypen. Das Resultat eines Programms ist immer ein einziger Wert. So scheinen funktionale Programmiersprachen nicht mehr Möglichkeiten als ein guter Taschenrechner zu bieten. Allerdings lassen sich die Funktionen nicht nur auf Zahlen, sondern auf beliebig komplexe Datenstrukturen (Listen, Bäume, Paare, usw.) anwenden. So ist es möglich Programme zu schreiben, die z.B. einen Text einlesen, Schreibfehler erkennen und als Resultat den korrigierten Text zurückliefern. Bekannte Vertreter der funktionalen Programmiersprachen sind *Scheme*, Microsofts *F#* und *Haskell*. Letztere wird in der Veranstaltung „Grundlagen der Programmierung 2" verwendet.

1.1.2. Imperative Programmiersprachen

„Imperativ" stammt vom lateinischen Wort „imperare" = „befehlen". Tatsächlich besteht ein imperatives Programm aus einer Abfolge von Befehlen, die nacheinander ausgeführt werden und den Zustand des Speichers verändern. Das klingt zunächst sehr ähnlich wie die bereits erwähnten Maschinenprogramme. Die Befehle höherer Programmiersprachen sind jedoch komplexer als Maschinenbefehle und in der Regel wird vom tatsächlichen Speicher abstrahiert. Anstatt zu manipulierende Speicherbereiche direkt zu adressieren, kann der Programmierer beschreibende Namen

vergeben und diese im Quelltext verwenden. Daher ist der Quelltext wesentlich besser lesbar als der Maschinencode. Man kann imperative Programmiersprachen in *prozedurale* und *objektorientierte* Sprachen unterscheiden.

Prozedurale Programmiersprachen erlauben es dem Programmierer den imperativen Quelltext durch Gruppierung zu strukturieren. Je nach Programmiersprache heißen diese gruppierten Programmteile „Unterprogramme“, „Routinen“, „Prozeduren“ oder „Funktionen“. Sie können Parameter entgegennehmen und Ergebnisse zurückgeben und innerhalb des Programms mehrfach verwendet werden. Typische prozdurale Programmiersprachen sind *Fortran, COBOL* und *Pascal*.

prozedurale Programmiersprachen

Objektorientierte Programmiersprachen erlauben die Definition von Objekten, welche gewisse Eigenschaften und Fähigkeiten besitzen. Der „Bauplan“ der Objekte wird dabei in sog. *Klassen* definiert, welche *Attribute* (Eigenschaften) und *Methoden* (Fähigkeiten) besitzen. Methoden können den *Zustand* (Werte seiner Attribute) eines Objekts verändern. Methoden dienen aber auch dazu, dass Objekte untereinander Nachrichten austauschen. Ein wesentliches Konzept der objektorientierten Programmierung ist die *Vererbung*. Es können *Unterklassen* erzeugt werden, welche sämtliche Attribute und Methoden ihrer Oberklasse übernehmen und eigene hinzufügen können. Mehr zur objektorientierten Programmierung findet sich in Kapitel 5. Bekannte objektorientierte Programmiersprachen sind *Java, C++* und *C#*. Allerdings unterstützen die meisten modernen imperativen Sprachen (z.B. *Ruby, Modula-3* und Python) ebenfalls objektorientierte Programmierung.

objektorientierte Programmiersprachen

1.2. Python

Abbildung 1.1.

Python ist eine imperative Programmiersprache, die auch Konzepte der objektorientierten und funktionalen Programmierung unterstützt. Ursprünglich als Lehrsprache entwickelt, legt sie viel Wert auf Übersichtlichkeit und einfache Erlernbarkeit. Python wird üblicherweise mit einem Interpreter statt eines Compilers verwendet. Obwohl der Name eigentlich auf die englische Komikertruppe *Monty Python* zurückgeht, etablierte sich die Assoziation zur Schlange, nicht zuletzt vermutlich wegen des Logos (Abb.: 1.1).

Python läuft auf Windows, MacOS und UNIX/Linux und ist frei erhältlich und verwendbar. Zur Zeit gibt es zwei verschiedene Versionen, Python2 und Python3. Wir werden im Vorkurs das neuere Python3 verwenden. Die aktuelle Version ist Python 3.9.0, die Version auf den Rechnern der **R**echner**b**etriebsgruppe **I**nformatik (RBI) ist Python 3.7.9. Beide Versionen sind kompatibel und die meisten Programme laufen sowohl unter 3.9.0 als auch 3.7.9.

Wichtigste Informationsquelle zu Python ist die Homepage:

`http://www.python.org`

Dort findet man außer der Dokumentation, Tutorials, FAQs und HowTos auch die aktuellen Versionen zum Herunterladen.

Wo bekommt man's?

Mac OSX ab Version 10.3 und die meisten Linux-Distributionen liefern Python bereits mit. Es kann dann über den Paketmanager installiert werden. Häufig enthalten sie nicht die neuste Version, aber solange es sich um Python3 handelt, sollte es für den Anfang reichen. Ansonsten kann die aktuelle Version auf der Python Homepage[1] heruntergeladen werden.

[1] http://www.python.org/download

Python für Windows (XP und höher) kommt mit einem Installer und kann ebenfalls auf der Homepage heruntergeladen werden.

1.2.1. IDLE

Entwicklungs-umgebung

IDLE ist eine *Entwicklungsumgebung* für Python, die seit Version 1.5.2 im Python-Paket enthalten ist. Eine Entwicklungsumgebung ist eine Sammlung von nützlichen Programmen, die den Vorgang der Programmierung unterstützen. IDLE wurde speziell für Anfänger*innen entworfen. Das Hauptaugenmerk wurde auf eine einfache Bedienung und Übersichtlichkeit gelegt. Für größere Programme und Softwareprojekte bietet IDLE vermutlich nicht genug Funktionalität, aber um mit der Sprache vertraut zu werden und kleinere Programme zu schreiben, wie für den Vorkurs und die Lehrveranstaltungen des kommenden Semesters, ist IDLE sehr gut geeignet. Daher werden wir es hier verwenden[2].

Interaktive Python-Shell

Auf den Unix/Linux basierten Betriebssystemen des Instituts wird IDLE über die Kommandozeile der Shell (siehe Anhang B.1) mit dem Befehl `idle3` gestartet. In einem Fenster öffnet sich ein Python-Interpreter im interaktiven Modus. Hier können Python-Anweisungen eingegeben und mit der Return-Taste ⏎ direkt ausgeführt werden. Der Interpreter signalisiert durch drei spitze Klammern >>> seine Bereitschaft Anweisungen entgegenzunehmen. Nun lässt sich beispielsweise feststellen, ob Python mit den Grundrechenarten vertraut ist (Abb. 1.2).

Python Shell

File Edit Shell Debug Options Windows Help

```
Python 3.2.3 (default, Oct 19 2012, 20:13:42)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> 3+5
8
>>> 8-10
-2
>>> 4*7
28
>>> 5/2
2.5
>>> 
```

Ln: 12 Col: 4

Abbildung 1.2.: Rechnen mit Python

Der Versuch einer zwischenmenschlichen Kontaktaufnahme scheitert jedoch kläglich (Abb. 1.3), was uns nach dem, was wir zu Beginn dieses Kapitels gelernt haben, nicht wundern dürfte. Immerhin sendet der Interpreter eine Fehlermeldung und gibt einen Hinweis darauf, was wir falsch gemacht haben. Der Name 'hallo' ist nicht definiert.

[2] z.B. PyCharm (https://www.jetbrains.com/de-de/pycharm/)

```
*Python Shell*
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Oct 19 2012, 20:13:42)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> hallo
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    hallo
NameError: name 'hallo' is not defined
>>>
Ln: 9 Col: 4
```

Abbildung 1.3.: Versuch einer Kontaktaufnahme

Syntax-Highlighting

Zu den nützlichen Funktionen der Python-Shell gehört das *Syntax-Highlighting* . Dabei werden Teile des eingegebenen Textes je nach ihrer Bedeutung farblich hervorgehoben. Bisher sind unsere Eingaben an den Interpreter immer schwarz dargestellt worden, die Rückgaben des Interpreters werden blau (Abb. 1.2) dargestellt. Fehlermeldungen erscheinen in rot (Abb. 1.3). In den Einstellungen lässt sich die Farbwahl individuell anpassen.

Autovervollständigung

Eine weitere nützliche Funktion ist die Autovervollständigung . Gibt man den Anfang eines Befehls ein und betätigt dann die Tab-Taste ⇆ , so vervollständigt IDLE den Befehl selbstständig, solange die Eingabe eindeutig ist. Ist sie nicht eindeutig, so öffnet sich ein kleines Menü mit allen möglichen Befehlen (Abb.: 1.4).

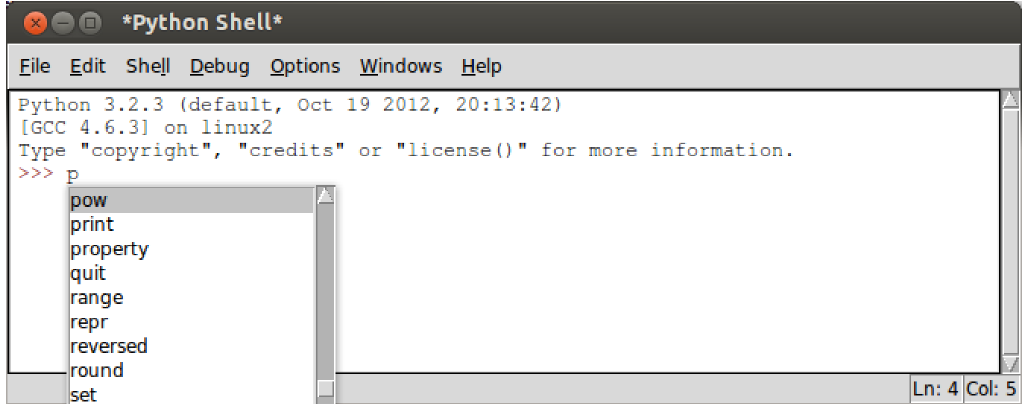


Abbildung 1.4.: Autovervollständigung

Syntax

Zusätzlich zeigt IDLE die *Syntax* des ausgewählten Befehls an (Abb. 1.5). Die Syntax beschreibt das Muster, nach dem der Befehl verwendet werden muss. Für den `print`-Befehl gilt z.B., dass zunächst der Wert (engl.: *value*), welcher ausgegeben werden soll, übergeben werden muss. `, ...,` bedeutet, dass auch mehrere Werte übergeben werden können. Der Parameter `sep` (engl.: *seperator*) gibt an, wie die Werte getrennt werden sollen, `end` gibt an, welches Zeichen am Schluss stehen soll. Der `file`-Parameter gibt an, wohin die Werte geschrieben werden sollen.

```
*Python Shell*
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Oct 19 2012, 20:13:42)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> print(
         print(value, ..., sep=' ', end='\n', file=sys.stdout)
Ln: 4 Col: 10
```

Abbildung 1.5.: Befehlssyntax

Die Parameter `sep, end` und `file` sind bereits mit Werten belegt. Wird der `print`-Befehl ohne Angabe dieser Parameter aufgerufen, dann werden die übergebenen Werte durch ein Leerzeichen getrennt und am Schluss wird ein Zeilenumbruch (`\n`) eingefügt. Gibt man beim Aufrufen des Befehls explizit Werte für einen oder mehrere der Parameter an, so werden die Defaultwerte überschrieben (Abb. 1.6). Ausgegeben wird das Ganze in der Python-Shell, denn sie ist die Standardausgabe (`sys.stdout`).

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Oct 19 2012, 20:13:42)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> print("hallo", "vorkurs")
hallo vorkurs
>>> print("hallo", "vorkurs", sep="! ")
hallo! vorkurs
>>> 
Ln: 8 Col: 4
```

Abbildung 1.6.: Verwendung des `print`-Befehls

eingebaute Hilfe-Funktion

Benötigt man mehr als nur eine Erinnerung an die Syntax eines Befehls und hätte man gerne noch Erklärungen zu den Parametern, bietet sich die eingebaute Hilfe-Funktion der Python-Shell an. Mit dem Befehl `help([Befehlsname])` kann man sie aufrufen und erhält die Ausgabe direkt in der Python-Shell (Abb.: 1.7).

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Oct 19 2012, 20:13:42)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current sys.stdout.
    sep:  string inserted between values, default a space.
    end:  string appended after the last value, default a newline.

>>>
Ln: 13 Col: 35
```

Abbildung 1.7.: Verwendung der eingebauten Hilfe-Funktion

Nun mag man berechtigterweise denken, dass das ja alles schön und gut und bestimmt auch eine Hilfe ist, aber bisher scheint man mit IDLE lediglich Programme, die aus einer Zeile bzw. einem einzigen Befehl bestehen, schreiben zu können. Außerdem muss man sie anscheinend jedes Mal wieder neu eintippen. Um Programmcode dauerhaft abzuspeichern, benötigen wir einen Editor. Der IDLE- Editor wird über die Tastenkombination Strg + N oder über den Menüpunkt „New File“ des „File“-Menüs geöffnet (siehe Anhang B.1).

IDLE Editor

Der IDLE Editor bietet, wie die Python-Shell, Syntax-Highlighting, Autovervollständigung und das Anzeigen der Syntax eines Befehls. Jedoch springt beim Betätigen der ↵ -Taste lediglich der Cursor in die nächste Zeile, die eingegebene Anweisung wird *nicht* direkt ausgeführt. So ist es uns möglich die beiden Befehle, die wir zunächst in der Python-Shell eingegeben und direkt ausgeführt hatten, hintereinander einzugeben (Abb.: 1.8).

```
*Untitled*
File Edit Format Run Options Windows Help
print("hallo", "vorkurs")
print("hallo", "vorkurs", sep="! ")

Ln: 3 Col: 0
```

Abbildung 1.8.: Eingabe mehrerer Anweisungen im Editor

Die eingegebenen Befehle können nun auch über das „File“-Menü gespeichert werden. Python-Quelltext sollte immer mit der Dateiendung *.py* gespeichert werden. Über den Menüpunkt „Run Module“ des „Run“-Menüs oder F5 (vergl. B.1) kann das Programm dann gestartet werden. Die Programmausgabe erfolgt in der Python-Shell (Abb.: 1.9).

.py

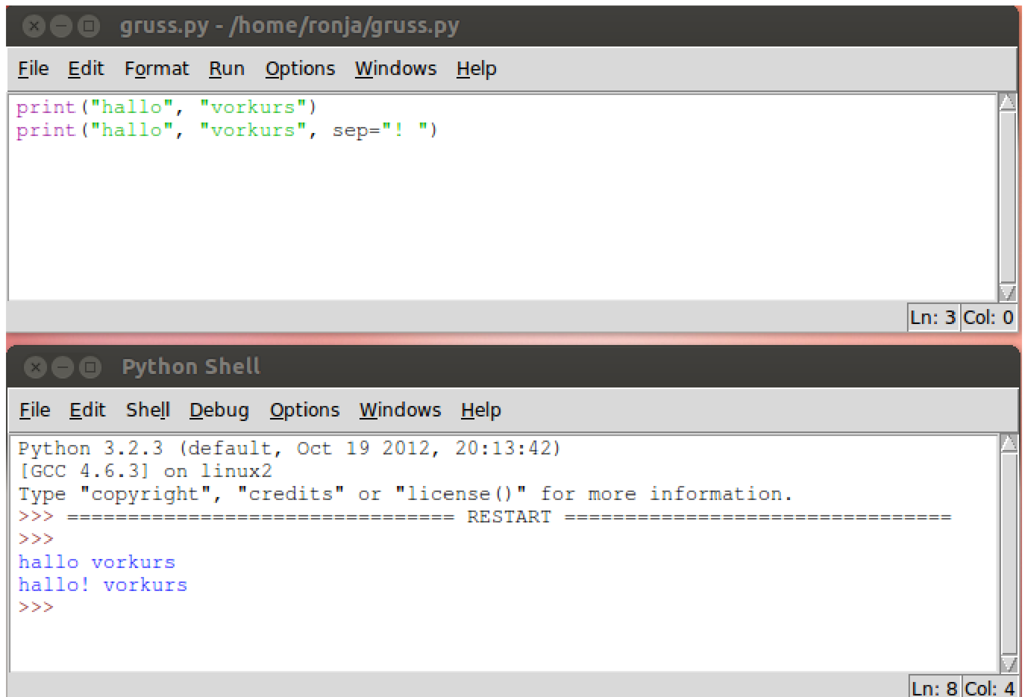


Abbildung 1.9.: Programm und Ausgabe

2. Grundlagen der Programmierung in Python

Bisher haben wir lediglich gesehen, wie man eine vorher festgelegte Nachricht, wie z.B. „hallo vorkurs“ in der Python-Shell ausgeben lassen kann. Das ist sicherlich ein bisschen wenig. Wir wollen Eingaben entgegennehmen, verarbeiten und Ausgaben produzieren können. Dies erreichen wir in Python mithilfe von Literalen und Variablen.

Literale

Literale sind Zahlen, wie 15, 7.2 oder $1.39e-8$, Zeichenketten (engl.: *strings*), wie „hallo“ oder Wahrheitswerte (vergl. Kap. 7). Sie werden auch *Direkt-Wert-Konstanten* genannt, da sie wortwörtlich den Wert bezeichnen, für den sie stehen. Die Zahl 17 steht immer für den Wert 17 und nichts anderes. Sie ist eine Konstante, ihr Wert kann nicht geändert werden kann.

2.1. Datentypen

2.1.1. Wahrheitswerte

Wahrheitswerte (vergl. Kap. 7) oder auch `bool` sind der einfachste elementare Datentyp in Python. Sie können einen von zwei möglichen Werten annehmen, nämlich `True` oder `False`. Python stellt für den Datentyp `bool` die logischen Operatoren `and`, `or` und `not` zur Verfügung (Tab. 2.1). Für eine ausführliche Erklärung der Semantik (Bedeutung) der Operatoren, verweisen wir auf Kapitel 7.2.2

Operator	Beschreibung	Beispiel
`and`	UND, Konjuktion	`True and True` ergibt `True`
`or`	ODER, Disjunktion	`True or False` ergibt `True`
`not`	NICHT, Negation	`not True` ergibt `False`

Tabelle 2.1.: Boolsche Operatoren in Python.

2.1.2. Zahlen

Für Zahlen stehen in Python vier Datentypen zur Verfügung: `int` für Ganzzahlen (engl.: *integer*), `long` für lange Ganzzahlen (engl.: *long integer*), `float` für Fließkommazahlen (engl.: *floating point numbers*) und `complex` für komplexe Zahlen (engl.: *complex numbers*). Im Vorkurs beschränken wir uns auf `int` und `float`.

`int`

Der Datentyp `int` steht für Ganzzahlen zur Verfügung. Ganzzahlen sind z.B. `3` oder `-13`, also positive und negative Zahlen ohne Nachkommastelle. Der Datentyp `int` ist durch den fehlenden Dezimalpunkt gekennzeichnet.

`float`

Beispiele für Fließkommazahlen sind `3.25, 5.0` und `1.38e-04`. Die Schreibweise mit dem `e` bezeichnet die Zehnerpotenzen. `1.38e-04` steht also für $1.38 \cdot 10^{-4}$. Der Datentyp `float` ist durch das Vorhandensein eines Dezimalpunktes gekennzeichnet. Dabei ist es unwichtig, ob die Nachkommastellen tatsächlich einen von 0 abweichenden Wert haben, oder überhaupt eingegeben werden. Beispielsweise wird `7.` als `float`-Wert 7.0 aufgenommen, da ein Dezimalpunkt eingegeben wurde.

Python unterstütze arithmetische Operationen wie Addition, Substraktion, Multiplikation und Division (Tab. 2.2).

Operator	**Beschreibung**	**Beispiel**
$+$	Addition	$2+5$ ergibt 7
$-$	Subraktion	$5-2$ ergibt 3
$*$	Multiplikation	$2*5$ ergibt 10
$/$	Division	$5/2$ ergibt 2.5
$//$	ganzzahlige Division	$5//2$ ergibt 2
$\%$	ganzzahliger Rest der Division (modulo)	$5\%2$ ergibt 1
$**$	Exponent	$5**2$ ergibt 25

Tabelle 2.2.: Arithmetische Operatoren in Python.

Dabei gehen Multiplikation und Division vor Addition und Subtraktion. Ferner gibt es die Möglichkeit, die Auswertungsreihenfolge arithmetischer Ausdrücke durch Klammersetzung zu beeinflussen.

Python richtet sich beim Ergebnis der arithmetischen Operationen in der Regel nach dem komplexesten Datenetyp der in der Eingabe vorkommt. Enthält die Eingabe beispielsweise eine `float`-Zahl, so wird auch das Ergebnis als `float` angezeigt. Enthält die Eingabe lediglich `int`-Zahlen, so wird auch das Ergebnis als `int` angezeigt (Abb. 2.1). Ausnahme ist die Division. Auch das Ergebnis einer Division zweier `int`-Zahlen wird als `float` angezeigt.

```
Python Shell
File  Edit  Shell  Debug  Options  Windows  Help
Python 3.2.3 (default, Oct 19 2012, 20:13:42)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> 3.0*2
6.0
>>> 3*2
6
>>> 4/2
2.0
>>> 3.6//2.0
1.0
>>> 3//2
1
>>>
Ln: 9 Col: 3
```

Abbildung 2.1.: Der Datentyp des Ergebnisses ist abhängig vom Datentyp der Eingabe

Ferner stellt Python Vergleichsoperatoren für Zahlen zur Verfügung. Das Ergebnis einer Vergleichsoperation ist immer ein Wahrheitswert (`bool`).

Operator	Beschreibung	Beispiel
==	Gleichheit	2 == 5 ergibt `False`
!=	Ungleichheit	5! = 2 ergibt `True`
>	größer als	2 > 5 ergibt `False`
<	kleiner als	2 < 5 ergibt `True`
>=	größer gleich	5 >= 2 ergibt `True`
<=	kleiner gleich	2 <= 2 ergibt `True`

Tabelle 2.3.: Vergleichsoperatoren in Python.

2.1.3. Zeichenketten

`string`

Für Text oder auch Folgen von Zeichen, stellt Python den Datentyp `string` zur Verfügung. Eingaben des Datentyps `string` sind durch Anführungszeichen gekennzeichnet. Dabei können einfache (‘ ’), doppelte (‘‘ ’’) oder dreifache (‘‘‘ ’’’) Anführungsstriche verwendet werden. Ein String kann immer nur mit derselben Sequenz von Anführungstrichen abgeschlossen werden, mit der er auch eingeleitet wurde.

einzelne Anführungszeichen (‘ ’) : Strings können durch Benutzung einzelner Anführungszeichen angegeben werden. Dabei können auch Ziffern eingegeben werden. Sie sind Teil des Strings und werden nicht als Zahlen erkannt. Leerräume (Leerzeichen und Tabulatoren) werden beibehalten.
Beispiel

`‘Und nun zu etwas ganz anderem...’.`
`‘Jeder nur 1 Kreuz.’`

doppelte Anführungszeichen (‘‘ ’’) : Strings in doppelten Anführungszeichen verhalten sich genauso wie Strings in einfachen Anführungszeichen. Einzelne Anführungszeichen können hier als Zeichen im String verwendet werden. Sie beenden den String nicht.
Beispiel

`‘‘Setz dich, nimm dir’n Keks, mach’s dir schön bequem...’’.`

dreifache Anführungszeichen (‘‘‘ ’’’) : Mithilfe dreifacher Anführungszeichen lassen sich mehrzeilige Strings eingeben (Abb. 2.2). Ferner kann man innerhalb dreifacher Anführungszeichen nach Belieben einfache und doppelte Anführungszeichen benutzen.
Beispiel

`‘‘‘Der Schwarze Ritter: ‘‘Aber schubsen kann ich dich noch’’.’’’`

```
Python Shell
File  Edit  Shell  Debug  Options  Windows  Help
Python 3.2.3 (default, Oct 19 2012, 20:13:42)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> 'Jeder nur 1 Kreuz'
'Jeder nur 1 Kreuz'
>>> "Setz dich, nimm dir'n Keks, mach's dir schön bequem..."
"Setz dich, nimm dir'n Keks, mach's dir schön bequem..."
>>> '''"Da ist das Untier!"
"Wo?"
"Na da"!
"Wo? Hinter dem Karnickel?"
"Es IST das Karnickel!"'''
'"Da ist das Untier!"\n"Wo?"\n"Na da"!\n"Wo? Hinter dem Karnickel?"\n"Es IST das
 Karnickel!"'
>>> 
Ln: 14 Col: 4
```

Abbildung 2.2.: Beispiele für Strings

ASCII
UTF-8

Da Computern technisch lediglich 1-en und 0-en zur Verfügung stehen, basiert die Zeichenrepräsentation auf einer Tabelle, die jedem Zeichen eine Zahl zuordnet. Der am weitesten verbreitete Zeichencode heißt *ASCII* (*A*merican *S*tandard *C*ode for *I*nformation *I*nterchange). Er verwaltet 256 Zeichen. Ein weiterer häufig verwendeter Code ist *UTF-8*. Er kann bis zu 1 114 112 Zeichen verwalten.

2.2. Built-in-Funktionen

built-in functions

Der Python-Interpreter hat eine Menge Funktionen, die fest eingebaut sind (engl.: *built-in-functions*). Diese Funktionen stehen jederzeit, unabhängig von geladenen Modulen (vergl. 2.9), zur Verfügung. Neben den bereits vorgestellten Operatoren für Zahlen, und der in Kapitel 1.2.1 bereits verwendeten print-Funktion für Strings, gibt es vielzählige andere Funktionen[1], von denen wir hier einige vorstellen möchten.

- **Für int und float**:
  - abs(x): gibt den Betrag $|x|$ der Zahl x zurück.
  - max(a,b): gibt den größeren der beiden Werte a und b zurück.
  - min(a,b): gibt den kleineren der beiden Werte a und b zurück.
  - round(x,n): gibt den, auf n Nachkommastellen gerundeten Wert von x zurück.

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Oct 19 2012, 20:13:42)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> abs(-2.5)
2.5
>>> max(7,2.46)
7
>>> min(7,2.46)
2.46
>>> round(3.3558,2)
3.36
>>> |
Ln: 12 Col: 4
```

Abbildung 2.3.: Einige Built-in Funktionen für Zahlen

- **Für string**:
  - <String>.isalpha():gibt True zurück, wenn der String nur aus Buchstaben besteht, ansonsten FALSE.
  - <String>.isdigit():gibt True zurück, wenn der String nur aus Ziffern besteht, ansonsten FALSE.
  - ord(c): gibt die Nummer des Zeichens c in der verwendeten Zeichencodetabelle zurück.
  - chr(<int>): gibt das, laut Zeichencodetabelle zur Ganzzahl i gehörige, Zeichen zurück.
  - +: fügt zwei Strings zusammen.

[1] http://docs.python.org/3/library/functions.html

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Oct 19 2012, 20:13:42)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> "hallo".isalpha()
True
>>> "hallo vorkurs".isalpha()
False
>>> "285".isdigit()
True
>>> "2.587".isdigit()
False
>>> ord('W')
87
>>> chr(64)
'@'
>>> 
Ln: 16 Col: 4
```

Abbildung 2.4.: Einige Built-in Funktionen für Strings

2.2.1. Typumwandlung

Wer aufmerksam gelesen hat, dem ist hoffentlich aufgefallen, dass es den `+` -Operator sowohl für Zahlen, als auch für Zeichenketten gibt. Allerdings ist die Funktion des `+` -Operators vom Typ der übergebenen Werte abhängig. Für Zahlen, addiert der `+` -Operator die beiden Parameter, während für Zeichenketten die beiden Parameter zusammengefügt werden (Abb. 2.5); selbst, wenn es sich bei den Zeichenketten um Folgen von Ziffern handelt. Python kann den Inhalt oder die Bedeutung von Strings nicht erkennen.

casten

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.2 (default, Sep  5 2011, 22:09:30)
[GCC 4.6.1] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> 3.5 + 7
10.5
>>> 'Sofa' + 'Hocker'
'SofaHocker'
>>> '3.5' + '7'
'3.57'
>>> 
Ln: 10 Col: 4
```

Abbildung 2.5.: `+`-Operator für Zahlen und Zeichenketten

Manchmal ist es notwendig, dass man Werte bestimmter Datentypen in andere Datentypen umwandelt (engl.: to *cast*). `int` in `float`, weil man vielleicht das Ergebnis einer Rechnung in `float` haben möchte, `float` in `int`, weil einen vielleicht lediglich der ganzzahlige Teil interessiert, und überhaupt, Zahlen in Zeichenketten, oder Zeichenketten in Zahlen, wobei letzteres nur funktioniert (und sinnvoll ist), wenn die Zeichenkette ausschließlich aus Ziffern besteht. Für solche Typumwandlungen oder *casten* gibt es in Python folgende Built-In-Funktionen:

`int()`: wandelt den übergebenen Parameter in einen `int`-Wert um. Dabei können Zeichenketten, die lediglich aus Ziffern bestehen, oder `float`-Werte übergeben werden. Eine Zeichenkette wie z.B. `''3.25''` wäre kein zulässiger Übergabewert, da der String einen `.` enthält. Bei der Umwandlung von `float-` in `int`-Werte kann es zu Informationsverlust kommen. Die Nachkommastellen werden einfach abgeschnitten es wird *nicht* gerundet (Abb. 2.6).

`int()`

`float()`: wandelt den übergebenen Parameter in einen `float`-Wert um. Es können `int`-Werte

`float()`

oder Zeichenketten, die aus Ziffern und *einem* Dezimalpunkt bestehen, übergeben werden. ``3.253.6'' ist kein zulässiger Übergabewert, da in dem String zwei „Dezimalpunkte“ vorkommen.

str()

str(): wandelt den übergebenen Parameter in einen string-Wert um. Es können int- oder float-Werte übergeben werden.

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.2 (default, Sep  5 2011, 22:09:30)
[GCC 4.6.1] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> int(2.9)
2
>>> float(5)
5.0
>>> int("5")
5
>>> float("5")
5.0
>>> str(5)
'5'
>>> str(3.97)
'3.97'
>>>
Ln: 16 Col: 4
```

Abbildung 2.6.: Typumwandlung in Python

2.2.2. Benutzereingabe und Bildschirmausgabe

Nun wissen wir zwar schon, dass es in Python unterschiedliche Datentypen gibt um Zahlen und Text darzustellen, aber bisher gibt es noch keine Möglichkeit mit einem evtl. Nutzer des Programms zu interagieren. Bisher haben wir keine Möglichkeit kennengelernt, Benutzereingaben außerhalb des interaktiven Modus der Python-Shell entgegenzunehmen oder Informationen für den Nutzer auf dem Bildschirm auszugeben.

input()

In Python werden Benutzereingaben mit der input()-Funktion eingelesen. Die input()-Funktion liest einen eingegebenen Wert, welcher mit der Return-Eingabe beendet wurde, und nimmt diesen Wert als string auf. Es ist möglich der input()-Funktion einen String zu übergeben, der dann auf dem Bildschirm erscheint (Abb. 2.7).

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Oct 19 2012, 20:13:42)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> input("Bitte geben Sie eine Zahl ein. ")
Bitte geben Sie eine Zahl ein. 17
'17'
>>> input("Wie lautet Ihr Name? ")
Wie lautet Ihr Name? Jane
'Jane'
>>>
Ln: 10 Col: 0
```

Abbildung 2.7.: Benutzereingabe in Python

print()

Für die Ausgabe von Nachrichten auf dem Bildschirm eignet sich die print()-Funktion. Diese haben wir in Kapitel 1.2.1 bereits kennengelernt. Die print()-Funktion erwartet einen oder mehrere, durch Komma (,) getrennte Zeichenketten, und gibt diese auf dem Bildschirm aus (Abb. 2.8).

```
Python 3.2.3 (default, Oct 19 2012, 20:13:42)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> print("Hallo", "Vorkurs")
Hallo Vorkurs
>>> print('''Mit der Print-Funktion werden Zeilenumbrüche \\n
auch als solche dargestellt.
Achtung, funktioniert nur bei Strings, die durch
3 Anführungszeichen gekennzeichnet sind.''')
Mit der Print-Funktion werden Zeilenumbrüche \n
auch als solche dargestellt.
Achtung, funktioniert nur bei Strings, die durch
3 Anführungszeichen gekennzeichnet sind.
>>>
```

Abbildung 2.8.: Ausgabe in Python

2.3. Variablen

Immer nur Literale zu verwenden würde unsere Möglichkeiten sehr einschränken. Wir benötigen eine Möglichkeit, beliebige Informationen zu speichern und zu verarbeiten. Dafür werden *Variablen* verwendet. Eine Variable bezeichnet einen bestimmten Speicherbereich, in dem Informationen abgelegt werden können. Über den Namen der Variable, kann dann der zugehörige Speicherbereich *gelesen* und *beschrieben* werden. Da der Speicher eines Computers lediglich 1-en und 0-en abspeichern kann, muss zusätzlich festgehalten werden, wie die Information in besagtem Speicherbereich zu interpretieren ist.

Beispiel 2.1.
Erinnern wir uns an Abb. 2.4. Ein Speicherbereich in dem `1010111` (binär für 87) steht, würde im Datentyp `int` als Zahl 87 interpretiert, im Datentyp `string` würde er als Zeichen *W* interpretiert.

Wir halten also fest, eine Variable hat immer:

einen Namen: es bietet sich an, einen Namen zu wählen, der einen Hinweis auf den Inhalt der Variablen gibt. So ist der entstehende Quelltext wesentliche leichter zu lesen und nachzuvollziehen. In Python müssen Variablennamen mit einem Unterstrich (_) oder Buchstaben beginnen und können beliebig viele weitere Buchstaben, Unterstriche und Ziffern enthalten. Leerzeichen und Interpunktionszeichen sind nicht zulässig. Python lässt zwar inzwischen Sonderzeichen, wie `ä, ü, ö` und ß zu, allerdings empfiehlt es sich diese nicht zu verwenden, da es sein kann, dass es auf anderen Systemen oder bei anderer Kodierung der Quelldatei, zu Problemen kommt. Python unterscheidet zwischen Groß - und Kleinschreibung. `varName` ist nicht dasgleiche wie `varname`. Beispiele für gültige Variablennamen sind: `a, _2_Ziegen, meine_variable` oder `Kloepse`. *Unzulässige* Variablennamen sind z.B.: `3_Finger, meine-variable, leer zeichen`. Eine nicht so günstige Wahl wäre `Klöpse`.

einen Typ: welcher festlegt, wie der Wert der Variablen interpretiert werden muss. In Python erfolgt die Festlegung des Typs *implizit* und muss im Programm nicht explizit angegeben werden.

einen Wert: der im Speicherbereich der Variablen gespeicherte Wert, bestehend aus einer Folge von 1-en und 0-en.

In Python werden Variablen angelegt, sobald einem *Namen* ein *Wert* zugewiesen wird. Die Zuweisung des *Typs* geschieht dabei automatisch anhand des Werts der zugewiesen wird. Diese Art der Typzuweisung nennt man auch *dynamische Typisierung*, im Gegensatz zur statischen Typisierung,

dynamische Typisierung

wie es z.B. bei `C++` der Fall ist. Zuweisungen erfolgen in `Python` mit einem `=` in der Form `<Name> = <Ausdruck>`. Hierbei wird *zuerst* der Ausdruck ausgewertet, *danach* wird der Wert (und Typ) dem Namen zugewiesen. So sind Zuweisungen wie in Abb. 2.10 möglich. Ist die Variable angelegt, so kann über den Namen der Variablen auf den zugehörigen Speicherbereich zugegriffen werden.

Zuweisung

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.2 (default, Sep  5 2011, 22:09:30)
[GCC 4.6.1] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> eine_Zahl=13
>>> eine_Zahl
13
>>> nocheine = 4
>>> nocheine
4
>>> eine_Zahl + nocheine
17
>>>
Ln: 10 Col: 24
```

Abbildung 2.9.: Beispiele für Variablen

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.2 (default, Sep  5 2011, 22:09:30)
[GCC 4.6.1] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> count=1
>>> count
1
>>> count=count+1
>>> count
2
>>> count=count-1
>>> count
1
>>>
Ln: 13 Col: 4
```

Abbildung 2.10.: Hoch- und Runterzählen von Variablen

Achtung! Vergibt man ein und denselben Namen zweimal, dann wird nicht eine zweite Variable mit demselben Namen angelegt. Wert und ggf. Typ der ersten Variablen werden in diesem Fall bei der zweiten Zuweisung einfach überschrieben (Abb. 2.11).

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.2 (default, Sep  5 2011, 22:09:30)
[GCC 4.6.1] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> meine_variable = 7.546
>>> meine_variable
7.546
>>> meine_variable = "ein Wert"
>>> meine_variable
'ein Wert'
>>>
Ln: 10 Col: 4
```

Abbildung 2.11.: Überschreiben von Variablen/Speicherbereichen

2.4. Kommentare und Dokumenation

Häufig vernachlässigt und später schmerzhaft vermisst, ist die Programmdokumentation. Dokumentation ist der Prozess, Informationen über das Programm und den zugehörigen Quelltext zu hinterlassen.

Die meisten Programme im Studium schreibt man nicht allein, sondern im Team. D.h. nicht nur man selbst muss den eigenen Quelltext nach 2 oder mehr Wochen noch lesen und verstehen können, auch andere müssen es können. Dafür hinterlässt man im Quelltext sog. *Kommentare*. Das sind kurze Textpassagen innerhalb des Quelltext, die eine bestimmte Stelle des Quellcodes kurz erläutern und auf Probleme, offene Aufgaben oder Ähnliches hinweisen können. Sie dienen allein als Gedankenstütze für den/die Programmierer*in und spielen für die Programmfunktion keinerlei Rolle. Sie werden vom Python-Interpreter ignoriert. Die einfachste Art einen Kommentar in Python zu verfassen, ist der *Zeilenkommentar*. Diese Art des Kommentars beginnt mit einem Rautezeichen (#) und endet mit dem Ende der Zeile (Abb. 2.12).

Kommentare

Zeilenkommentar

```
*Untitled*
File Edit Format Run Options Windows Help
###############################
# Beispiele für Kommentare
###############################

# Beispiel für einen Zeilenkommentar
numRabbits = 1 # Zählvariable für Kaninchen

'''Beispiel für einen Blockkommentar.
Dieser kann über mehrer Zeilen gehen.
Er geht so lange, bis er durch die Eingabe von
drei Anführungszeichen beendet wird'''

Ln: 9 Col: 0
```

Abbildung 2.12.: Beispiele für Kommentare

Für längere Kommentare bietet sich der *Blockkommentar* an. Ein Blockkommentar beginnt und endet mit drei Anführungszeichen (''') und kann über mehrere Zeilen gehen (Abb. 2.12). Ferner können solche Kommentare dann auch mit der `help()`-Funktion (Abschn. 1.2.1) aus der Python-Shell aufgerufen werden.

Blockkommentar

Außer Kommentaren im Quelltext gehört, vor allem zu einem größern Programm, immer auch eine *Dokumentation*. Das sind ein oder mehrere zusätzliche Dokumente, die die Funktion, den Aufbau und Bedienhinweise für das Programm enthalten. Die Dokumentation kann auch Informationen zu technischen Anforderungen und Installationsanweisungen enthalten.

Exkurs: Version 1

Nun wissen wir, wie man Benutzereingaben einliest, Werte in Variablen zwischenspeichert und verändert und wie man Nachrichten auf dem Bildschirm ausgibt. Wir können uns also daranmachen, ein erstes Programm zu schreiben. Wir könnten eine Folge von Zahlen abfragen, und überprüfen, ob die Zahlen aufsteigend sortiert sind.

Da wir ein Programm schreiben möchten, verwenden wir den IDLE-Editor. So können wir den Quelltext speichern und immer wieder verwenden. Für die Bedienung des IDLE-Editors verweisen wir auf Appendix B.1. Abbildung 2.13 zeigt den Quelltext unseres Programms.

```
sort1.py - /home/ronja/Dropbox/Skript/Code/sort1.py
File Edit Format Run Options Windows Help
#####################################
# Programm um zu ueberpruefen, ob eine
# Folge von Zahlen sortiert ist.
#
# Version 1
#####################################

#  1.Zahl erfragen
zahl1=input("Wie lautet die 1.Zahl?  ")
#  2.Zahl erfragen
zahl2=input("Wie lautet die 2.Zahl?  ")
#  3.Zahl erfragen
zahl3=input("Wie lautet die 3.Zahl?  ")
#  4.Zahl erfragen
zahl4=input("Wie lautet die 4.Zahl?  ")
#  5.Zahl erfragen
zahl5=input("Wie lautet die 5.Zahl?  ")
#Eingegebene Zahlen ausgeben
print(zahl1,zahl2,zahl3,zahl4,zahl5,sep=", ")
Ln: 2 Col: 26
```

Abbildung 2.13.: Sortierte Zahlenfolge: Version 1

Mithilfe des `input()`-Befehls fordert das Programm den Benutzer auf, 5 Zahlen nacheinander einzugeben. Diese werden in den Variablen `zahl1` bis `zahl5` zwischengespeichert. Dann wird die angegebene Folge mithilfe des `print()`-Befehls auf dem Bildschirm ausgegeben. Das war gar nicht so schwer. Aber irgendwie ist der Benutzer noch nicht glücklich. Tatsächlich, Beispiel 2.14 zeigt, dass das Programm auch unsortierte Folgen ausgibt und es dem Benutzer überlassen bleibt zu entscheiden, ob die Folge sortiert ist, oder nicht.

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.2 (default, Sep  5 2011, 22:09:30)
[GCC 4.6.1] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Wie lautet die 1.Zahl?  1
Wie lautet die 2.Zahl?  3
Wie lautet die 3.Zahl?  5
Wie lautet die 4.Zahl?  7
Wie lautet die 5.Zahl?  4
1, 3, 5, 7, 4
>>>
Ln: 12 Col: 4
```

Abbildung 2.14.: Durchlauf des Programms Version 1

Wir bräuchten eine Möglichkeit das Verhalten des Programms davon abhängig zu machen, ob bestimmte Bedingungen (in diesem Fall aufsteigende Sortierung) erfüllt sind, oder nicht.

2.5. Kontrollstrukturen

bedingte Ausführung

Anweisungsblock

Kontrollstrukturen sind Programmkonstrukte, die den Ablauf des Programms steuern. Bisher können wir nur Programme entwerfen, die eine Folge von Anweisungen von oben nach unten abarbeiten. Nun lernen wir die *bedingte Ausführung* von Anweisungsblöcken kennen. Ein *Anweisungsblock* ist eine Folge von Anweisungen, die gleich weit eingerückt sind. Die so erzwungene

einheitliche Einrückung trägt zur Übersichtlichkeit und Lesbarkeit des Programms bei. Konvention ist, dass die Einrückung jeweils vier Leerzeichen (␣␣␣␣) beträgt. Ist die Einrückung fehlerhaft, zeigt der Interpreter einen Einrückungsfehler (engl.: *indentation error*, siehe auch 4.1.1) an. Bestimmte Anweisungsblöcke, oder Programmteile, werden nur dann ausgeführt, wenn bestimmte Bedingungen erfüllt sind. Dabei muss die Bedingung eine Ausdruck sein, der als Wahrheitswerte (`bool`, siehe Kap. 2.1.1) ausgewertet werden kann. Dazu gehören z.B. alle Vergleichsoperatoren für Zahlen (Tab. 2.3). Ferner ist es in Python so, dass alle Zahlen die ungleich 0 sind, als `True` ausgewertet werden. Der Zahlenwert 0 oder `NULL` wird zu `False` ausgewertet. Mithilfe der bool'schen Operatoren `and, or` und `not` (Tab. 2.1), können Bedingungen geeignet kombiniert werden.

Indentation Fault

2.5.1. Verzweigung

if-Anweisung

Die `if`-Anweisung wird durch das Schlüsselwort `if` eingeleitet. Diesem folgt die Abfrage einer Bedingung, gefolgt von einem Doppelpunkt (`:`). In der nächsten Zeile beginnt dann, eingerückt, der Anweisungsblock der ausgeführt werden soll, falls die Bedingung erfüllt ist. Dieser Anweisungsblock darf nicht leer sein. D.h. auf eine `if`-Anweisung muss tatsächlich mindestens eine eingerückte Zeile folgen, sonst gibt es eine Fehlermeldung (siehe Abb. 4.9).

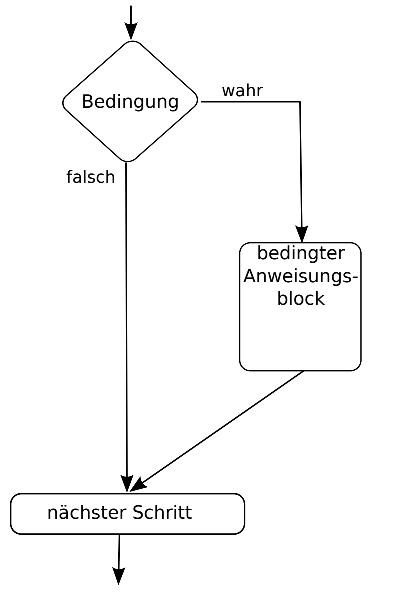


Abbildung 2.15.: Kontrollflussdiagramm der `if`-Anweisung

Syntax:

```
if <Bedingung>:
␣␣␣␣<Anweisung>
<nächster Schritt>
```

Die `if`-Anweisung verzweigt den Programmablauf. *Wenn* die Bedingung erfüllt ist, *dann* wird der eingerückte Anweisungsblock ausgeführt. Ist die Bedingung nicht erfüllt, dann wird der bedingte Anweisungsblock nicht ausgeführt (vergl. Abb. 2.15).

Beispiel 2.2 (`if`-Anweisung)**.**
Eine `if`-Abfrage in einem Programm sieht folgendermaßen aus:

```
*Untitled*
File  Edit  Format  Run  Options  Window  Help
# Beispiel für if-Anweisung

var=7 # Variable var mit Wert 7 anlegen
var2=13 # Variable var2 mit Wert 13 anlegen

if (var == 7):  # Überprüfe, ob var den Wert 7 hat
    print("Der Wert der Variablen 'var' ist 7.") # Meldung ausgeben

if (var2 == 7):  # Überprüfe, ob var2 den Wert 7 hat
    print("der Wert der Variablen 'var2' ist 7.") # Meldung ausgeben

print("Fertig") # Meldung am Ende des Programms

Ln: 13 Col: 0
```

Abbildung 2.16.: Eine `if`-Anweisung

Falls der Wert der Variablen `var` 7 ist, *dann* gib die Nachricht "Der Wert der Variablen 'var' ist 7." aus, und *falls* der Wert der Variablen `var2` 7 ist, *dann* gib die Nachricht "Der Wert der Variablen 'var2' ist 7." aus. Da zu Beginn des Programms der Variablen `var` der Wert 7 und der Variablen `var2` der Wert 13 zugewiesen wird, ist die Bedingung in der ersten Abfrage erfüllt, in der zweiten aber nicht. Dementsprechend erscheint beim Ausführen des Programms folgendes auf dem Bildschirm (Abb. 2.17).

```
Python Shell
File  Edit  Debug  Options  Windows  Help
Python 3.2.3 (default, Jun 18 2015, 21:46:42)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>>
Der Wert der Variablen 'var' ist 7.
Fertig
>>>

Ln: 8 Col: 4
```

Abbildung 2.17.: Ausgabe der `if`-Anweisung

if . . . else Anweisung

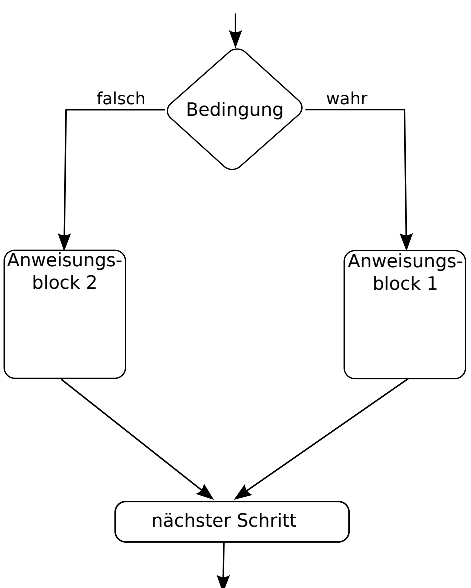


Abbildung 2.18.: Kontrollflussdiagramm der `if ...else`-Anweisung

Syntax:

```
if <Bedingung>:
␣␣␣␣<Anweisung1>
else:
␣␣␣␣<Anweisung2>
<nächster Schritt>
```

Für den Fall, dass das Programm bei Nichterfüllung der Bedingung alternative Anweisungen ausführen muss, gibt es die `if ...else`-Anweisung. Dabei wird der alternative Anweisungsblock mit dem Ausdruck `else` gefolgt von einem Doppelpunkt (`:`) eingeleitet. *Falls* die Bedingung erfüllt ist, *dann* wird Anweisungsblock 1 ausgeführt, *sonst* wird Anweisungsblock 2 ausgeführt (Abb. 2.18).

Beispiel 2.3 (`if ...else`-Anweisungen)**.**

bei_ifelse.py - /home/ronja/Dropbox/Skript/Code/bei_ifelse.py

File Edit Format Run Options Windows Help

```python
# Beispiel für if...else-Anweisung

var1=7  # Variable var1 mit Wert 7 anlegen

if (var1 == 7): #Abfrage der Bedingung
    print("Der Wert der Variablen ist 7.")  # Meldung ausgeben
    print(var1) #Wert von var1 ausgeben
else:
    print("Der Wert der Variablen ist nicht 7") # Meldung ausgeben
    print(var1)

var2=5  # Variable var2 mit Wert 5 anlegen
if (var2 == 10): #Abfrage der Bedingung.
    print("Der Wert der Variablen ist 10.") # Meldung ausgeben
    print(var2)
else:
    print("Der Wert der Variablen ist nicht 10") # Meldung ausgeben
    print(var2)
print("Fertig") # Meldung am Ende des Programms
```

Ln: 20 Col: 0

Abbildung 2.19.: Eine `if...else`-Anweisung

Falls die Bedingung erfüllt ist, *dann* wird die entsprechende Nachricht ausgegeben, *sonst* wird eine andere Nachricht ausgegeben. In der oberen `if...else`-Anweisung, ist die Bedingung erfüllt, also wird der Anweisungsblock hinter dem `if` ausgeführt. In der unteren `if...else`-Anweisung ist die Bedingung nicht erfüllt, daher wird der `else`-Anweisungsblock ausgeführt (Abb. 2.20). Am

Schluss wird die Anweisung nach dem `if...else`-Konstrukt ausgeführt und "Fertig" wird auf dem Bildschrim ausgegeben.

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Oct 19 2012, 20:13:42)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Der Wert der Variablen ist 7.
7
Der Wert der Variablen ist nicht 10
5
Fertig
>>>
Ln: 11 Col: 4
```

Abbildung 2.20.: Ausgabe der `if...else`-Anweisung

elif-Anweisung

Mithilfe der `elif`-Anweisung ist es möglich, mehrere Ausdrücke auf ihren Wahrheitswert zu überprüfen und einen Anweisungsblock auszuführen, sobald einer der Ausdrücke zu `True` ausgewertet wird. Wie die `else`-Anweisung, ist die `elif`-Anweisung optional. Während allerdings lediglich *eine* `else`-Anweisung pro `if`-Anweisung verwendet werden kann, können einer `if`-Anweisung beliebig viele `elif`-Anweisungen folgen.

Beispiel 2.4 (`if...elif`-Anweisung)**.**

```
bei_elif.py - /home/ronja/Dropbox/Skript/Code/bei_elif.py
File Edit Format Run Options Windows Help
# Beispiel für if...elif...else-Anweisung

var = 100  # Variable var mit Wert 100 anlegen

if (var == 200): #Abfrage der Bedingung
    print("1 - Bedingung erfüllt")  # Meldung ausgeben
    print(var) #Wert von var1 ausgeben
elif (var==150):
    print("2 - Bedingung erfüllt") # Meldung ausgeben
    print(var)
elif (var == 100): #Abfrage der Bedingung.
    print("3 - Bedingung erfüllt") # Meldung ausgeben
    print(var)
else:
    print("4 - keine der Bedingungen wurde erfüllt") # Meldung ausgeben
    print(var)

print("Fertig") # Meldung am Ende des Programms
Ln: 19 Col: 0
```

Abbildung 2.21.: Eine `if...elif ...else`-Anweisung

Zu Beginn wird der Wert der Variablen `var` auf 100 gesetzt. Somit ist weder die erste Bedingung, noch die zweite Bedingung erfüllt. Erst die dritte Bedingung ist erfüllt, dementsprechend wird der zugehörige Anweisungsblock ausgeführt. Abbildung 2.22 zeigt die Ausgabe des Programms.

```
Python Shell
File  Edit  Debug  Options  Windows  Help
Python 3.2.3 (default, Jun 18 2015, 21:46:42)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>>
3 - Bedingung erfüllt
100
Fertig
>>> |
Ln: 9 Col: 4
```

Abbildung 2.22.: Ausgabe der `if...elif...else`-Anweisung

Mehrfachverzweigungen

In Situationen in denen eine Bedingung überprüft werden muss nachdem sich herausgestellt hat, dass eine andere Bedingung zu `True` ausgewertet wurde, verwendet man *Mehrfachverzweigung*. Es ist möglich `if...elif ...else`-Anweisungen innerhalb von anderen `if...elif ...else`-Anweisungen zu verwenden.

Beispiel 2.5 (Mehrfachverzweigung)**.**
Das folgende Programm (Abb. 2.23) testet, ob ein Wert kleiner als 200, und wenn ja, ob er auch kleiner als 150 ist. Falls der Wert nicht kleiner als 200 ist, wird getestet, ob der Wert kleiner als 300 ist. Je nachdem welche Fälle erfüllt sind, werden entsprechende Nachrichten auf dem Bildschirm ausgegeben.

```
mehrfach.py - /home/ronja/Dropbox/Skript/Code/mehrfach.py
File  Edit  Format  Run  Options  Windows  Help
# Beispiel für Mehrfachverzweigung

var = 170  # Variable var mit Wert 170 anlegen

if (var < 200): #Abfrage ist der Wert kleiner als 200
    print("Der Wert ist kleiner als 200")  # Meldung ausgeben
    if (var<150): #Abfrage wenn er kleiner als 200 ist, ist er dann kleiner 150
        print("Der Wert ist kleiner als 150") #Meldung ausgeben.
    else: # Wert ist kleiner als 200 aber nicht < 150
        print("Der Wert liegt zwischen 150 und 200") #Meldung ausgeben
elif (var < 300): #Abfrage ist der Wert kleiner als 300 (war nicht <200)
    print("Der Wert liegt zwischen 200 und 300") #Meldung ausgeben
else: # Wert ist nicht kleiner als 300
    print('''Keine der Bedingungen wurden erfüllt.
Somit ist der Wert größer oder gleich 300''') # Meldung ausgeben

print("Fertig") # Meldung am Ende des Programms
|
Ln: 18 Col: 0
```

Abbildung 2.23.: Beispiel einer Mehrfachverzweigung

Da der Wert der Variablen `var` zu Beginn auf 170 gesetzt wird, ist die erste Bedingung (`var < 200`) erfüllt. Dementsprechend wird als nächstes überprüft, ob `var` auch kleiner als 150 ist. Diese Bedingung ist nicht erfüllt und der `else`-Anweisungsblock wird ausgeführt (Abb. 2.24).

```
Python 3.2.3 (default, Jun 18 2015, 21:46:42)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>>
Der Wert ist kleiner als 200
Der Wert liegt zwischen 150 und 200
Fertig
>>>
```

Abbildung 2.24.: Ausgabe des Beispiels einer Mehrfachverzweigung

Exkurs: Version 2

Mit dem neuen Wissen über Verzweigungen sollte es uns möglich sein, unser Programm zu verbessern. Wir sollten nun in der Lage sein, zu überprüfen, ob die Zahlen aufsteigend sortiert sind und entsprechend reagieren können. Zunächst gilt es die Frage zu klären: *woran erkennt man, dass etwas aufsteigend sortiert ist*? Nach kurzem Nachdenken kommen wir darauf, dass eine Folge von Zahlen aufsteigend sortiert ist, wenn jede Zahl größer als ihr Vorgänger ist. Demnach müssen wir also abfragen, ob das der Fall ist. Da wir eine Meldung ausgeben möchten, wenn zwei Zahlen nicht aufsteigend sortiert sind, fragen wir stattdessen ab, ob die Sortierung verletzt ist, d.h. eine Zahl größer als ihr Nachfolger ist. Wenn wir das für alle Zahlen abfragen, haben wir implizit auch die Antwort auf die Frage, ob die Zahlen aufsteigend sortiert sind. Außerdem wäre es hilfreich, wenn sofort eine Meldung ausgegeben wird, dass die eingegebenen Zahlen nicht sortiert sind und nicht erst alle Zahlen eingegeben werden müssen, nur um zu erfahren, dass bereits die ersten beiden nicht aufsteigend sortiert waren. Also wird die Eingabe der nächsten Zahl nur angefordert, wenn klar ist, dass alle bisher eingegebenen Zahlen aufsteigend sortiert sind.

Anhand dieser Überlegungen ergibt sich folgendern Programmablauf:

1. Eingabe der ersten Zahl
2. Eingabe der zweiten Zahl
3. Überprüfen, ob erste Zahl größer als zweite Zahl ist.
   3a. falls **ja**: Meldung ausgeben, und fertig.
   3b. falls **nein**: Eingabe der dritten Zahl.
   4. Überprüfen, ob zweite Zahl größer als dritte Zahl ist.
      4a. falls **ja**: Meldung ausgeben, und fertig.
      4b. falls **nein**: Eingabe der vierten Zahl.
      5. Überprüfen, ob dritte Zahl größer als vierte Zahl ist.
         5a. falls **ja**: Meldung ausgeben, und fertig.
         5b. falls **nein**: Eingabe der fünften Zahl.
         6. Überprüfen, ob vierte Zahl größer als fünfte Zahl ist.
            6a. falls **ja**: Meldung ausgeben, und fertig.
            6b. falls **nein**: Meldung ausgeben, dass Folge sortiert ist.

Dann müssen wir uns noch daran erinnern, dass wir Zahlen vergleichen wollen, die `input()`-Funktion aber nur Strings einliest (vergl.: 2.2.2). Also müssen wir, um die Zahlen vergleichen zu können, die Benutzereingabe mit der `int()`-Funktion (2.2.1) in einen Zahlentyp umwandeln. Abbildung 2.25 zeigt unser verbessertes Programm.

sort2.py - /home/ronja/Dropbox/Skript/Code/sort2.py

File Edit Format Run Options Windows Help

```
#####################################
# Programm um zu überprüfen, ob eine
# Folge von Zahlen sortiert ist.
#
# Version 2
#####################################

#  1.Zahl erfragen
zahl1=input("Wie lautet die 1.Zahl?  ")
#  2.Zahl erfragen
zahl2=input("Wie lautet die 2.Zahl?  ")
# Überprüfen, ob die Zahlen aufsteigend sortiert sind.
if (int(zahl1)>int(zahl2)): # zahl1 und zahl2 sind nicht aufsteigend sortiert
    print("Die Folge ist nicht sortiert.") # Meldung ausgeben
    print(zahl1, zahl2, sep=", ")
else: # Die ersten beiden Zahlen sind sortiert, also weiter eingeben.
    # 3.Zahl erfragen
    zahl3=input("Wie lautet die 3.Zahl?  ")
    if (int(zahl2)>int(zahl3)): # zahl2 und zahl3 sind nicht aufsteigend sortier
        print("Die Folge ist nicht sortiert.")
        print(zahl1, zahl2, zahl3, sep=", ")
    else: # Zahlen 1 bis 3 sind aufsteigend sortiert
        # 4.Zahl erfragen
        zahl4=input("Wie lautet die 4.Zahl?  ")
        if (int(zahl3)>int(zahl4)): # Zahl 3 und 4 nicht aufsteigend sortiert
            print("Die Folge ist nicht sortiert.")
            print(zahl1, zahl2, zahl3, zahl4, sep=", ")
        else: # Zahlen 1 bis 4 sind aufsteigend sortiert
            #5. Zahl erfragen
            zahl5=input("Wie lautet die 5.Zahl?  ")
            if (int(zahl4)>int(zahl5)):
                print("Die Folge ist nicht sortiert.")
                print(zahl1,zahl2,zahl3,zahl4,zahl5,sep=", ")
            else: # Folge ist sortiert
                print("Die Folge ist sortiert.")
                #Eingegebene Zahlen ausgeben
                print(zahl1,zahl2,zahl3,zahl4,zahl5,sep=", ")
```

Ln: 6 Col: 37

Abbildung 2.25.: Sortierte Zahlenfolge: Version 2

Das Programm überprüft nun nach jeder Eingabe, ob die Zahlen sortiert sind und gibt eine Meldung aus, falls die Sortierung verletzt ist. Ferner fragt das Programm dann auch nicht mehr nach weiteren Eingaben, denn daran, dass die Zahlenfolge nicht sortiert ist, werden weitere Eingaben nichts ändern (Abb. 2.26).

```
Python Shell
File  Edit  Shell  Debug  Options  Windows  Help
>>>
Wie lautet die 1.Zahl?  1
Wie lautet die 2.Zahl?  3
Wie lautet die 3.Zahl?  4
Wie lautet die 4.Zahl?  2
Die Folge ist nicht sortiert.
1, 3, 4, 2
>>> ================================ RESTART ================================
>>>
Wie lautet die 1.Zahl?  1
Wie lautet die 2.Zahl?  2
Wie lautet die 3.Zahl?  3
Wie lautet die 4.Zahl?  4
Wie lautet die 5.Zahl?  5
Die Folge ist sortiert.
1, 2, 3, 4, 5
>>>
                                                                Ln: 54 Col: 4
```

Abbildung 2.26.: Durchläufe des Programms Version 2

Dieses Programm überprüft zumindest schon einmal selbst, ob die Folge aufsteigend sortiert ist. Allerdings können nur Folgen mit exakt 5 Zahlen eingegeben werden. Wir könnten natürlich mit viel Fleißarbeit die Anzahl der Zahlen auf 10, 20 oder sogar 100 erhöhen, aber die Anzahl der Zahlen, die abgefragt würden, wäre immernoch festgelegt.

2.5.2. Schleifen

Schleifenkopf
Schleifenrumpf

Schleifen erlauben es uns, einzelne Anweisungen, oder Anweisungsblöcke mehrmals zu durchlaufen. Eine Schleife besteht aus einem *Schleifenkopf*, in dem die Ausführungsbedingungen angegeben sind, und einem *Schleifenrumpf*, der den bedingten Anweisungsblock enthält. Python stellt zwei verschiedene Schleifenkonstrukte zur Verfügung, die `while`-Schleife und die `for`-Schleife.

while-Schleife

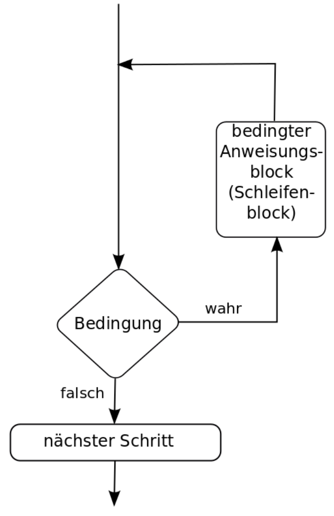


Abbildung 2.27.: Kontrollflussdiagramm für Schleifen

Syntax:

```
while <Bedingung>:
␣␣␣␣<Anweisung>
<nächster Schritt>
```

Die `while`-Schleife macht es möglich, einen Anweisungsblock zu durchlaufen, solange eine zugehörige Bedingung erfülllt ist. Dabei überprüft die `while`-Schleife *vor* jedem Durchlauf (vorprüfende Schleife), ob die Bedingung erfüllt ist. Ist die Bedingung nicht erfüllt, so geht das Programm zum nächsten Schritt nach dem Schleifenrumpf über (Abb. 2.27).

Alternativ kann auch eine Abbruchbedingung formuliert werden. In diesem Fall wird der Ausdruck im Schleifenkopf auf `True` gesetzt und im Schleifenrumpf überprüft, ob die Abbruchbedingung erfüllt ist. In diesem Fall wird der Schleifenrumpf mindestens einmal, bis zum Abfragen der Abbruchbedingung ausgeführt. Ein solches Konstrukt wird auch *nachprüfende Schleife* genannt.

Besondere Aufmerksamkeit muss beim Verwenden dieser Schleifenart darauf gelegt werden, dass die Abbruchbedingung der `while`-Schleife, bzw. die Nichterfüllung der Bedingung im Schleifenkopf, auch erreicht wird. Ist das nicht der Fall, so entsteht eine *Endlosschleife*. Das Programm durchläuft den Schleifenrumpf immer und immer wieder und endet nie.

Endlosschleife

Beispiel 2.6 (`while`-Schleife)**.**
Das Programm in Abb. 2.28 zeigt die `while`-Schleife einmal als vorprüfende und einmal als nachprüfende Schleife. Die Funktion beider Schleifen ist die Ausgabe der Ziffern 0 bis 8 auf dem Bildschirm.

bei_while.py - /home/ronja/Dropbox/Skript/Code/bei_while.py

File Edit Format Run Options Windows Help

```
# Beispiele für while-Schleife

# Beispiel für vorprüfende Schleife
print("Ausgabe vorprüfende Schleife")
count = 0 # Zählvariable count mit Wert 0 anlegen
while (count < 9): #Solange der Wert von count kleiner als 9 ist:
    print("Der Zähler lautet:", count) # gib den Wert der Zählvariablen aus.
    count = count + 1 # erhöhe count um 1. WICHTIG! da sonst Endlosschleife
print("Ende der vorprüfenden while-Schleife")

# Beispiel für eine nachprüfende Schleife
print("Ausgabe nachprüfende Schleife")
count = 0 # Zählvariable count mit Wert 9 anlegen
while True: # Führe folgende Anweisungen aus
    print("Der Zähler lautet:", count) # gib den Wert der Zählvariablen aus.
    count = count + 1 # erhöhe count um 1. WICHTIG! da sonst Endlosschleife
    if (count >= 9): # falls count größer oder gleich 9 ist:
        break # beende die Schleife
print("Ende der nachprüfenden while-Schleife")
```

Ln: 19 Col: 46

Abbildung 2.28.: Die `while`-Schleife als vor- und nachprüfende Schleife

Die vorprüfende Schleife überprüft zunächst, ob der Wert der Variablen `count` kleiner als 9 ist. Wenn das der Fall ist, wird der Wert ausgegeben, *danach* wird der Wert der Variablen `count` um 1 erhöht, und der Programmfluss springt zurück zum Schleifenkopf. Hat der Wert von `count` 9 erreicht, so springt der Programmfluss zur `print`-Anweisung nach dem Schleifenrumpf.

```
Python Shell
File  Edit  Shell  Debug  Options  Windows  Help
>>> ================================ RESTART ================================
>>>
Ausgabe vorprüfende Schleife
Der Zähler lautet: 0
Der Zähler lautet: 1
Der Zähler lautet: 2
Der Zähler lautet: 3
Der Zähler lautet: 4
Der Zähler lautet: 5
Der Zähler lautet: 6
Der Zähler lautet: 7
Der Zähler lautet: 8
Ende der vorprüfenden while-Schleife
Ausgabe nachprüfende Schleife
Der Zähler lautet: 0
Der Zähler lautet: 1
Der Zähler lautet: 2
Der Zähler lautet: 3
Der Zähler lautet: 4
Der Zähler lautet: 5
Der Zähler lautet: 6
Der Zähler lautet: 7
Der Zähler lautet: 8
Ende der nachprüfenden while-Schleife
>>> |
Ln: 28 Col: 4
```

Abbildung 2.29.: Die `while`-Schleife als vor- und nachprüfende Schleife

Die nachprüfende Schleife gibt zunächst den Wert der Variablen `count` aus und erhöht diesen dann anschließend. Erst dann wird zum ersten Mal überprüft, ob der Wert von `count` kleiner als 9 ist. Da wir aber nach dem Abbruchkriterium fragen, muss die Bedingung der vorprüfenden Schleife negiert (verneint) werden. Die Anweisung „Führe aus, *solange* `count` < 9 ist“, entspricht der Anweisung „Breche ab, *sobald* `count` ≥ 9“. Nun ist `count` mit einem entsprechend kleinen Wert initialisiert worden. Wäre der Wert aber größer als 9 gewesen, hätte die Schleife diesen trotzdem einmal ausgegeben, da erst danach festgestellt worden wäre, dass die Abbruchbedingung erfüllt ist. Der Befehl `break` beendet die Schleife sofort und der Programmfluss springt zur ersten Anweisung *nach* dem Schleifenrumpf. Beide Schleifen erzeugen die gleiche Ausgabe (Abb. 2.29)

for-Schleife

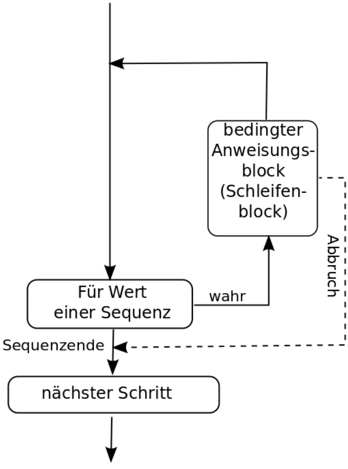


Abbildung 2.30.: Kontrollflussdiagramm für `for`-Schleifen

Die `for`-Schleife durchläuft den Schleifenrumpf für *jeden* Wert einer gegebenen Sequenz (Abb. 2.30). Diese Werte können Zahlen sein, es können aber auch Zeichen eines Strings oder andere Objekte sein. Zunächst wird der Anweisungsblock im Schleifenrumpf für das erste Element der Sequenz durchlaufen, dann für das zweite, usw. , bis die Schleife für alle Elemente in der Sequenz einmal durchlaufen wurde. Anders als bei der `while`-Schleife, muss hier also nicht darauf geachtet werden, dass die Schleife nicht endlos läuft, denn die Schleife endet auf jeden Fall, wenn das letzte Element der Sequenz erreicht ist. Meist wird die `for`-Schleife als Zählschleife (Bsp.: 2.7) verwendet. Dabei wird ein *Index* innerhalb eines bekannten Bereichs (*Startwert, Endwert*) durchlaufen. In Python wird ein solcher Index mithilfe der `range()`-Funktion (Bsp.: 2.7) erzeugt.

Beispiel 2.7 (`range()`-Funktion).
Das folgende Programm zeigt die Verwendung der `range()`-Funktion. Die `range()`-Funktion erwartet die Übergabe einer `int`-Zahl als Endwert und erzeugt eine Sequenz von `int`-Zahlen von 0 bis zum angegebenen Endwert, *ohne den Endwert einzuschließen*. Optional können zusätzlich Startwert und Schrittgröße angegeben werden. Dann startet die erzeugte Sequenz beim angegebenen Startwert und läuft in der angegebenen Schrittgröße bis zum Endwert, ohne diesen einzuschließen.

`range()`

range.py - /home/ronja/Dropbox/Skript/Code/range.py

File Edit Format Run Options Windows Help

```
# Beispiele für die Verwendung der range()-Funktion

# range(5) erzeugt die Sequenz 0, 1, 2, 3, 4
print("Ausgabe für range(5)")
for i in range(5):
    print (i)


# range(11,17,2) erzeugt die Sequenz 11, 13, 15
print("Ausgabe für range(11,17,2)")
for i in range(11,17,2):
    print(i)

# range(2,-2,-1) erzeugt die Sequenz 2, 1, 0, -1
print("Ausgabe für range(2,-2,-1)")
for i in range(2,-2,-1):
    print(i)
```

Ln: 18 Col: 0

Abbildung 2.31.: Verwendung der `range()`-Funktion

Durch die Angabe einer negativen Schrittgröße ist es auch möglich absteigende Sequenzen zu generieren (Abb. 2.32).

Python Shell

File Edit Shell Debug Options Windows Help

```
>>> ================================ RESTART ================================
>>>
Ausgabe für range(5)
0
1
2
3
4
Ausgabe für range(11,17,2)
11
13
15
Ausgabe für range(2,-2,-1)
2
1
0
-1
>>>
```

Ln: 157 Col: 4

Abbildung 2.32.: Verwendung der `range()`-Funktion

Beispiel 2.8 (`for`-Schleife).
Das folgende Programm zeigt die Verwendung der `for`-Schleife. Wie die `for`-Schleife als Zählschleife verwendet wird, haben wir im vorherigen Beispiel (2.7) bereits gesehen. Daher beschränken wir uns hier auf Strings und andere Objekte. Um das tun zu können, müssen wir ein wenig auf den kommenden Abschnitt (2.6) vorgreifen. Im kommenden Abschnitt lernen wir Listen kennen. Diese sind ebenfalls Sequenzen, daher verwenden wir in unserem jetztigen Beispiel bereits eine Liste.

for.py - /home/ronja/Dropbox/Skript/Code/for.py

File Edit Format Run Options Windows Help

```
# Beispiele für for-Schleifen

# for-Schleife über eine Sequenz von Zeichen
print("Erstes Beispiel")
for i in "Python 3":
    print ("aktuelles Zeichen:", i)


# for-Schleife über eine Liste von Früchten
print("Zweites Beispiel")
fruits = ['Apfel', 'Birne', 'Banane'] # Liste fruits mit 3 Objekten
for i in fruits:
    print ("aktuelle Frucht:", i)
```

Ln: 14 Col: 0

Abbildung 2.33.: Verwendung der `for`-Schleife auf Sequenzen

Die `for`-Schleife durchläuft jedes einzelne Zeichen des Strings und gibt dieses aus. Auch das Leerzeichen wird ausgegeben. Bei der Liste von Früchten ist es ebenso. Jede Frucht wird einzeln Ausgegeben. Nach jedem Durchlauf geht `i` zum nächsten Element der Sequenz und führt den Schleifenrumpf aus (Abb. 2.34).

```
Python Shell
File Edit Shell Debug Options Windows Help
>>>
Erstes Beispiel
aktuelles Zeichen: P
aktuelles Zeichen: y
aktuelles Zeichen: t
aktuelles Zeichen: h
aktuelles Zeichen: o
aktuelles Zeichen: n
aktuelles Zeichen:
aktuelles Zeichen: 3
Zweites Beispiel
aktuelle Frucht: Apfel
aktuelle Frucht: Birne
aktuelle Frucht: Banane
>>>
Ln: 172 Col: 4
```

Abbildung 2.34.: Ausgabe der `for`-Schleife auf Sequenzen

2.5.3. Schleifen-Kontrollanweisungen

Es gibt zusätzliche Befehle mit denen man den Programmfluss innerhalb von Schleifen kontrollieren kann. Einen dieser Befehle haben wir in Beispiel 2.6 bereits gesehen. Python unterstützt die folgenden Schleifen-Kontrollanweisungen.

`break`

`break`: Der `break`-Befehl beendet die Schleife sofort. Der Programmfluss springt direkt, ohne die Schleifenbedingung erneut zu überprüfen, zur ersten Anweisung nach dem Schleifenrumpf.

```
break.py - /home/ronja/Dropbox/Skript/Code/break.py
File Edit Format Run Options Windows Help
# Beispiele für break-Befehl

# in for-Schleife
print('for-Schleife')
for letter in 'Python': # Durchlaufe jeden Buchstaben im Wort 'Python'
    if (letter == 'h'): # falls der Buchstabe 'h' ist
        break # beende die Schleife
    print('aktueller Buchstabe:', letter) # gib den aktuellen Buchstaben aus

# in while-Schleife
print('while-Schleife')
var = 10 # variable var mit Wert 10
while var > 0: # Solange var größer als 0 ist
    print('aktueller Wert:', var) # gib den aktuellen Wert aus
    var = var-1 # veringere den Wert von var um 1
    if (var  == 7): # falls var den Wert 7 hat
        break # beende die Schleife

print('Ende des Beispiels')
Ln: 20 Col: 0
```

Abbildung 2.35.: Verwendung der `break`-Anweisung

Die `for`-Schleife in Abbildung 2.35 durchläuft jeden Buchstaben des Wortes ‘Python’ und gibt ihn aus. Trifft sie allerdings auf ein ‘h’, was nach dem ‘t’ der Fall ist, so führt die `break`-Anweisung dazu, dass die Schleife *sofort* beendet wird. Der Buchstabe ‘h’ wird nicht einmal mehr ausgegeben (Abb. 2.36). Auch die `while`-Schleife wird sofort beendet und, obwohl die Bedingung (`var`> 0) noch erfüllt ist, nicht weiter ausgeführt.

```
Python Shell
File  Edit  Shell  Debug  Options  Windows  Help
>>> ================================ RESTART ================================
>>>
for-Schleife
aktueller Buchstabe: P
aktueller Buchstabe: y
aktueller Buchstabe: t
while-Schleife
aktueller Wert: 10
aktueller Wert: 9
aktueller Wert: 8
Ende des Beispiels
>>>
Ln: 78 Col: 4
```

Abbildung 2.36.: Ausgabe des Programms zur Verwendung der `break`-Anweisung

`continue`

`continue`: Die `continue`-Anweisung führt dazu, dass für den aktuellen Schleifendurchlauf alle Anweisungen *nach* dem `continue`-Befehl ignoriert werden und der Programmfluss *sofort* wieder zurück zum Schleifenkopf springt. Die Schleifenbedingung wird erneut überprüft (Abb. 2.37).

```
continue.py - /home/ronja/Uni/Lernzentrum/Vorkurs/WS1516/Mater  /Skript/Co
File  Edit  Format  Run  Options  Windows  Help
# Beispiele für continue-Befehl

# in for-Schleife
print('for-Schleife')
for letter in 'Python': # Durchlaufe jeden Buchstaben im Wort 'Python'
    if (letter == 'h'): # falls der Buchstabe 'h' ist
        continue # beende Schleifendurchlauf
    print('aktueller Buchstabe:', letter) # gib den aktuellen Buchstaben aus

# in while-Schleife
print('while-Schleife')
var = 10 # variable var mit Wert 10
while var > 5: # Solange var größer als 0 ist
    var = var-1 # veringere den Wert von var um 1
    if (var  == 7): # falls var den Wert 7 hat
        continue # beende die Schleife
    print('aktueller Wert:', var)

print('Ende des Beispiels')

Ln: 20 Col: 0
```

Abbildung 2.37.: Verwendung der `continue`-Anweisung

Die `for`-Schleife in Abbildung 2.37 durchläuft jeden Buchstaben des Wortes 'Python' und gibt ihn aus. Trifft sie allerdings auf ein 'h', was nach dem 't' der Fall ist, so führt die `continue`-Anweisung dazu, dass dieser Schleifendurchlauf *sofort* beendet wird. Der Buchstabe 'h' wird nicht ausgegeben (Abb. 2.38). Der Programmfluss kehrt zum Schleifenkopf zurück und die Schleife läuft für den nächsten Buchstaben ('o') weiter. In der `while`-Schleife führt die `continue`-Anweisung dazu, dass alle Zahlen von 9 bis 5, außer der 7 ausgegeben werden (Abb. 2.38).

```
Python Shell
File Edit Shell Debug Options Windows Help
>>>
for-Schleife
aktueller Buchstabe: P
aktueller Buchstabe: y
aktueller Buchstabe: t
aktueller Buchstabe: o
aktueller Buchstabe: n
while-Schleife
aktueller Wert: 9
aktueller Wert: 8
aktueller Wert: 6
aktueller Wert: 5
Ende des Beispiels
>>>
Ln: 18 Col: 4
```

Abbildung 2.38.: Ausgabe des Programms zur Verwendung der `continue`-Anweisung

pass

`pass`: Der `pass`-Befehl wird verwendet, wenn Python eine Anweisung verlangt, aber das Programm keine Anweisung ausführen soll. Er ist besonders nützlich in der Programm-entwurfsphase, wenn man bereits weiß, dass man eine bestimmte Funktion oder einen be-stimmten Fall benötigt, aber noch nicht genau weiß wie. Die `pass`-Anweisung kann auch verwendet werden, um im `if ...elif...else`-Konstrukt immer den optionalen `else`-Anweisungsblock explizit zu programmieren. Damit ist die Gefahr, dass man nur vergessen hat den `else`-Anweisungsblock zu programmieren, geringer (Abb. 2.39).

```
pass.py - /home/ronja/Dropbox/Skript/Code/pass.py
File Edit Format Run Options Windows Help
# Beispiele f\"ur pass-Befehl


liste = ['tulpe', 'rose', 'magerite'] #Liste mit 3 Elementen
for item in liste: # Durchlaufe alle Elemente der Liste list
    if(item == 'rose'): # Falls das aktuelle Element 'rose' ist
        pass # etwas besonders muss passieren, aber noch nicht klar was oder wie
    else: #Falls das Element nicht 'rose' ist
        pass # tue nichts
    print('aktuelles Element:',item) # gib aktuelles Element aus
print('Ende des Beispiels')

Ln: 12 Col: 0
```

Abbildung 2.39.: Verwendung der `pass`-Anweisung

Die `pass`-Anweisung in obigem Programm hat keinerlei Funktion bezüglich des Programmablaufs. Sie macht das Programm aber erst lauffähig. Ohne die `pass`-Anweisung im bedingten Anweisungsblock, würde der Interpreter einen *Indentation fault* anzeigen. So ist es möglich schon andere Funktionen, wie z.B. die `print`-Funktion, zu testen, ohne dass man sich bereits Gedanken darüber machen muss, was nun genau mit dem Element `'rose'` geschehen muss (Abb. 2.40).

```
Python Shell
File Edit Shell Debug Options Windows Help
>>> ================================ RESTART ================================
>>>
aktuelles Element: tulpe
aktuelles Element: rose
aktuelles Element: magerite
Ende des Beispiels
>>>
Ln: 24 Col: 4
```

Abbildung 2.40.: Ausgabe zum Programm der `pass`-Anweisung

2.6. Listen

Datenstruktur

Eine Datenstruktur ist ein Konstrukt zur Speicherung und Organisation von Daten. Die Daten werden in einer bestimmten Art und Weise angeordnet und verknüpft, um den Zugriff auf sie und ihre Verwaltung effizient zu ermöglichen. Die einfachste *Datenstruktur* in Python ist die Sequenz. Im vorherigen Abschnitt (2.5.2) haben wir bereits zwei Arten vom Sequenzen gesehen. Strings und Listen. Strings haben wir bereits in Abschnitt 2.1.3 behandelt, nun wollen wir uns mit Listen befassen.

Listen in Python sind ein sehr vielseitiger Datentyp. Sie sind durch eckige Klammern (`[]`) gekennzeichnet, und können Elemente unterschiedlichen Typs enthalten. Wie bereits in Beispiel 2.8 gesehen, wird eine Liste durch eckige Klammern(`[]`) erzeugt. Zwischen den Klammern werden, durch Komma getrennt, die Elemente der Liste angegeben.

Zugriff auf Werte in einer Liste

Auf die einzelnen Elemente einer Liste kann über ihren Index zugegriffen werden. Dabei erhält das erste Element den Index 0, das zweite den Index 1, das dritte 2, usw . Der Index des gewünschten Elements einer Liste wird in eckigen Klammern hinter den Namen der Liste geschrieben (Abb. 2.41).

```
Python Shell
File Edit Shell Debug Options Windows Help
>>>
>>> # Beispiele für Listen
>>>
>>> liste_1 = ['Jane',17, 'Joe', 3.5, "John"] # Liste mit 5 Elementen
>>> liste_2 = [1,2,3,4,5,6,7] # Liste mit 7 Elementen
>>>
>>> liste_1[0] # das erste Element von liste_1
'Jane'
>>> liste_2[1:5] # zweites bis fünftes Element von liste_2
[2, 3, 4, 5]
>>>
Ln: 261 Col: 4
```

Abbildung 2.41.: Zugriff auf Elemente einer Liste

Mithilfe des Doppelpunkts (`:`) kann eine Sequenz von Indizes angegeben werden und so auf einen Ausschnitt der Liste zugegriffen werden. Achtung! der Endwert ist nicht eingeschlossen!

Werte in Listen verändern

`append()`

Mithilfe des Zuweisungsoperators (`=`) können Elemente in Listen verändert werden (Abb. 2.42). Die `append()`-Methode dient dazu Elemente an eine Liste anzuhängen (engl.: *append*).

```
Python Shell
File  Edit  Debug  Options  Windows  Help
==== No Subprocess ====
>>>
>>>
>>>
>>> liste_1 = ['Jane', 17, 'Joe', 3.5, 'John']
>>> liste_2 = [1,2,3,4,5,6,7]
>>>
>>> liste_1[2] # drittes Element von liste_1
'Joe'
>>> liste_1[2] = 'Katrin' #drittes Element von liste_1 wird geändert zu 'Katrin'
>>> liste_1[2]
'Katrin'
>>> liste_2.append('Tom')
>>> liste_2
[1, 2, 3, 4, 5, 6, 7, 'Tom']
>>>
Ln: 19 Col: 4
```

Abbildung 2.42.: Verändern und Hinzufügen von Elementen

Werte aus Liste löschen

Wenn man den Index des Elements kennt, welches man aus der Liste entfernen möchte, so kann man den `del`-Befehl verwenden (Abb. 2.43). Kennt man den Index nicht, sondern lediglich den Namen des Elements, so muss man die `remove()`-Methode verwenden, um das Element zu entfernen (engl.: *remove*).

`del`

`remove()`

```
Python Shell
File  Edit  Shell  Debug  Options  Windows  Help
>>> # Elemente aus Liste löschen
>>>
>>> liste_1 = ['Jane', 17, 'Joe', 3.5, 'John']
>>> liste_2 = [1,2,3,4,5,6,7]
>>>
>>> del liste_1[2] # lösche drittes Element von liste_1
>>> liste_1
['Jane', 17, 3.5, 'John']
>>> liste_2.remove(3) # lösche die 3 aus liste_2
>>> liste_2
[1, 2, 4, 5, 6, 7]
>>> liste_1.remove('Jane') # lösche 'Jane' aus liste_1
>>> liste_1
[17, 3.5, 'John']
>>>
Ln: 327 Col: 4
```

Abbildung 2.43.: Löschen von Elementen einer Liste

Neben den oben erwähnten Funktionen und Operatoren, gibt es noch zahlreiche weitere. Wir wollen hier nur wenige kurz vorstellen

Operator/ Funktion	**Beschreibung**	**Beispiel**
len(<list>)	Länge der Liste	len([1,2,5]) ist 3
+	Zusammenfügen	[1,2,3]+['a','b','c'] ergibt [1,2,3,'a','b','c']
*	Wiederholung	['a']*3 ergibt ['a','a','a']
in	Enthalten in	1 in [1,2,3] ergibt True
<list>.count(obj)	Häufigkeit des Vorkommens von obj in Liste list	[1,1,3].count(1) ergibt 2

Tabelle 2.4.: Operatoren und Funktionen für Listen in Python.

Exkurs: Version 3

Nun haben wir gelernt, wie wir Programmteile wiederholt durchlaufen können und wie wir eine nicht vordefinierte Anzahl von Daten verwalten können. Unternehmen wir also einen neuen Versuch unser Programm zu verbessern. Ziel ist es, dass der Benutzer beliebig viele Zahlen nacheinander eingeben kann und das Programm mitteilt, ob diese Zahlen aufsteigend sortiert sind. Nachdem was wir bereits über Sortierung wissen, muss unser Programm also etwa so ausseheh:

1. Solang der Benutzer noch Zahlen eingeben möchte:
    2. Eingabe der Zahl
    3. Überprüfe, ob vorherige Zahl größer als eingegebene Zahl ist
        3a. falls **ja**: Meldung ausgeben, und fertig.
        3b. falls **nein**: zurück zu 2.
4. Wenn Eingabe beendet, Zahlen ausgeben

Schön und gut, aber wie weiß der Computer, dass der Benutzer keine Zahlen mehr eingeben möchte? Wir benötigen eine Eingabe, die dem Programm signalisiert, dass die Eingabe beendet ist. Außerdem müssen wir die eingegeben Zahlen ja auch irgendwie zwischenspeichern und verwalten. Erstens wollen wir sie am Ende des Programms ausgeben, und für den Vergleich der letzten beiden Zahlen müssen wir zumindest Zugriff auf die vorletzte Zahl haben. Wir verbessern unseren Entwurf also zu folgendem:

1. Lege eine leere Liste an.
2. Solang der Benutzer noch Zahlen eingeben möchte:
    2. Eingabe der Zahl
    3. Überprüfe, ob die Eingabe das Ende der Eingabe signalisiert
        3a. falls **ja**: Schleifenbedingung auf False setzen.
        3b. falls **nein**: Überprüfe, ob vorherige Zahl größer als eingegebene Zahl ist
            3b1. falls **ja**: Meldung ausgeben, und Eingabe beenden.
            3b2. falls **nein**: Eingegebene Zahl an Liste anhängen; und zurück zu 2
4. Wenn Eingabe beendet, Zahlen ausgeben

Index Error

Value Error

Während der Programmierung fällt uns noch auf, dass wir vor dem Prüfen, ob die vorherige Zahl größer als die gerade eingegebene Zahl ist, noch prüfen müssen, ob es überhaupt eine vorherige Zahl gibt. Versuchen wir auf Elemente zuzugreifen, die es nicht gibt, zeigt der Python-Interpreter einen Index-Fehler (engl.: *IndexError*) an. Allerdings hat Python auch hier für den unerfahrenen Programmierer vorgesorgt. Das Element *vor* dem ersten Element einer Liste, ist das erste Element. Ferner sollten wir überprüfen, ob es sich bei der Eingabe, wenn nicht um das Signal zum Beenden der Eingabe, um eine Zahl handelt. Sonst meldet der Python-Interpreter einen Werte-Fehler (engl.: *ValueError*), wenn wir versuchen die Eingabe für den Vergleich in eine int-Zahl umzuwandeln. Also schreiben wir folgendes Programm (Abb. 2.44).

sort3.py - /home/ronja/Dropbox/Skript/Code/sort3.py

File Edit Format Run Options Windows Help

```
#####################################
# Programm um zu überprüfen, ob eine
# Folge von Zahlen sortiert ist.
#
# Version 3
#####################################

print(''' Hallo, in diesem Programm können Sie Zahlen eingeben.
Das Programm überprüft dann, ob die Zahlen in aufsteigender Reihenfolge
eingegeben wurden. Wenn Sie die Eingabe beenden möchten, geben Sie bitte 'e' ein
Eingabe_beenden = False # Variable um zu überprüfen, ob Eingabe beendet wurde
zahlen = [] # leere Liste anlegen, um Benutzereingaben zu speichern
while (Eingabe_beenden == False): # Eingabe, bis Eingabe beendet wird
    zahl = input("Geben Sie eine Zahl ein. ") # Eingabeaufforderung
    if(zahl == 'e'): # Benutzer möchte Eingabe beenden
        Eingabe_beenden = True
        print("Die Folge ist sortiert." ) # Meldung ausgeben
        continue # Rest der Schleife überspringen
    elif(zahl.isdigit()):# Überprüfen, ob eine Zahl eingegeben wurde
        zahlen.append(int(zahl)) # Eingabe an Zahlenfolge anhängen
        # Überprüfen, ob bereits mindestens 2 Zahlen eingegeben wurden
        if (len(zahlen)>1):
        # Vergleiche vorletztes und letztes Element der Liste
            if(zahlen[(len(zahlen)-2)]> int(zahl)): #Falls vorletztes größer als
                print("Die Folge ist nicht sortiert.") # Liste ist nicht sortier
                break # Eingabe abbrechen
# Eingabe ist beendet, jetzt die Liste ausgeben
print(zahlen)
```

Ln: 28 Col: 13

Abbildung 2.44.: Sortierte Zahlenfolge: Version 3

Nun ist das Programm nicht mehr selbsterklärend, denn der Benutzer muss wissen, welche Eingabe dem Programm signalisiert, dass die Eingabe von Zahlen abgeschlossen ist. Daher sorgen wir zunächst dafür, dass bei Programmstart ein kurzer Erklärungstext auf dem Bildschirm ausgegeben wird. Dann setzen wir die Kontrollvariable `Eingabe_beenden` auf `False`, denn zunächst gehen wir davon aus, dass Zahlen eingegeben werden sollen. Anschließend startet eine `while`-Schleife, die so lange läuft, bis die Kontrollvariable `Eingabe_beenden` den Wert `True` annimmt. Nachdem wir vom Benutzer eine Eingabe erfragt haben, überprüfen wir, ob es sich um das Signal zum Beenden der Eingabe handelt. Wir haben `e`, für 'Ende' gewählt. Falls die Eingabe `'e'` ist, setzen wir die Kontorllvariable `Eingabe_beenden` auf `True`, geben eine Meldung aus, dass die Folge sortiert ist und kehren mit einem `continue`-Befehl direkt zum Schleifenkopf zurück. Da die Schleifenbedingung nun nicht mehr erfüllt ist, wird die Schleife nicht erneut aufgeführt. Das Programm springt direkt zur Anweisung nach der `while`-Schleife und gibt die eingegebene Zahlenfolge aus.

Falls es sich bei der Eingabe nicht um das Signal zum Beenden der Eingabe handelt, überprüfen wir zunächst, ob es sich um eine Zahl handelt. Falls nicht, ignorieren wir die Eingabe einfach und fragen nach der nächsten Eingabe. Das passiert automatisch, wenn wir keine `else`-Anweisung geben. Falls es sich um eine Zahl handelt, so speichern wir sie in unserer Zahlenliste. Dann überprüfen wir, ob sich in der Liste mindestens 2 Zahlen befinden. Falls nein, wollen wir wieder nichts tun und nach der nächsten Eingabe fragen. Also benötigen wir wieder keine `else`-Anweisung. Falls ja, überprüfen wir, ob die vorherige Zahl (die sich als vorletztes Element in unserer Zahlenliste befindet) größer ist, als die gerade eingegebene Zahl (die sich sowohl als letztes Element in der Zahlenliste befindet, als auch noch in der Variablen `zahl`). Falls nein, ist die Folge sortiert, und wir fragen nach der nächsten Eingabe. Wir benötigen also erneut keine `else`-Anweisung. Falls ja, ist die aufsteigende Sortierung verletzt. Wir geben eine entsprechende Meldung aus und beenden die Eingabe sofort. Das Programm springt zur Anweisung nach der `while`-Schleife und gibt die eingegebene Zahlenfolge aus.

```
Python Shell
File Edit Shell Debug Options Windows Help
>>> ================================ RESTART ================================
>>>
 Hallo, in diesem Programm können Sie Zahlen eingeben.
Das Programm überprüft dann, ob die Zahlen in aufsteigender Reihenfolge
eingegeben wurden. Wenn Sie die Eingabe beenden möchten, geben Sei bitte 'e' ein
.
Geben Sie eine Zahl ein. 1
Geben Sie eine Zahl ein. 3
Geben Sie eine Zahl ein. 5
Geben Sie eine Zahl ein. 7
Geben Sie eine Zahl ein. e
Die Folge ist sortiert.
[1, 3, 5, 7]
>>> ================================ RESTART ================================
>>>
 Hallo, in diesem Programm können Sie Zahlen eingeben.
Das Programm überprüft dann, ob die Zahlen in aufsteigender Reihenfolge
eingegeben wurden. Wenn Sie die Eingabe beenden möchten, geben Sei bitte 'e' ein
.
Geben Sie eine Zahl ein. 1
Geben Sie eine Zahl ein. a
Geben Sie eine Zahl ein. b
Geben Sie eine Zahl ein. 5
Geben Sie eine Zahl ein. 3
Die Folge ist nicht sortiert.
[1, 5, 3]
>>>
                                                          Ln: 87 Col: 4
```

Abbildung 2.45.: Durchlauf des Programms Version 3

Tatsächlich funktioniert unser Programm. Wir können beliebig viele Zahlen hintereinander eingeben. Die Eingabe von anderen Zeichen wird ignoriert, und die Eingabe von `e` beendet die Eingabe und gibt eine Meldung und die Zahlenfolge aus. Sobald die Sortierung der Zahlen verletzt ist, wird ebenfalls eine Meldung ausgegeben und die Zahlenfolge ausgegeben.

2.7. Funktionen

Quelltext, der an verschiedenen Stellen eines Programms benötigt wird, sollte als *Funktion* geschrieben werden. Eine Funktion ist ein Stück wiederverwendbarer Quelltext, der eine Aufgabe ausführt. Wir haben bereits einige vordefinierte Built-In-Funktionen kennengelernt (Abschnitt 2.2), man kann aber auch seine eigenen Funktionen definieren. Der Vorteil von Funktionen ist, dass der Quelltext, der diese Aufgabe erledigt, nur an einer Stelle steht und an allen Stellen, an denen er gebraucht wird, lediglich auf diese Stelle verwiesen wird. Sollte sich also herausstellen, dass in diesem Teil ein Fehler ist, oder, dass die Ausführung der Aufgabe geändert werden muss, so muss der Quelltext nur an einer Stelle geändert werden. Das erhöht die Wartbarkeit und Lesbarkeit des Programms. Ferner ersparen einem Funktionen viel Arbeit, denn wenn man eine Funktion einmal programmiert hat, kann man sie immer wieder verwenden. Sogar in anderen Programmen (dazu mehr in Abschnitt 2.9).

`return`

Eine Funktion beginnt mit dem Schlüsselwort `def`, gefolgt von dem Funktionsnamen und Klammern `()`. Innerhalb der Klammern werden die Übergabeparameter angegeben. Der *Funktionskopf* wird mit einem Doppelpunkt (`:`) abgeschlossen. Der *Funktionsrumpf* ist eingerückt. Die erste Anweisung in einer Funktion kann ein Blockkommentar (2.4) sein. Der `return`-Befehl beendet eine Funktion und kann einen Wert zurückliefern.

Beispiel 2.9 (Funktion)**.**
Die erste Funktion in Abbildung 2.46 zeigt die einfachste Form einer Funktion in Python. Die Funktion macht nichts anderes, als die bereits vordefinierte `print()`-Funktion. Sie gibt den übergebenen

Parameter auf dem Bildschirm aus. Dafür verwendet die Funktion sogar die vordefinierte `print()`-Funktion. Letztendlich tut die Funktion `mein_print()` nichts anderes, als den erhaltenen Parameter an die `print()`-Funktion weiterzureichen.

funktion.py - /home/ronja/Dropbox/Skript/Code/funktion.py

File Edit Format Run Options Windows Help

```
# Beispiele f\"ur Funktionen

def mein_print(a):  # Funktionskopf
    '''Gibt den übergebenen String auf dem Bildschirm aus''' # Dokumentation
    print(a) # Ruft print()-Funktion mit übergebenem Parameter auf
    return # Ende der Funktion, kein Rückgabewert

def summe(n):
    '''berechnet die Summe der ersten n Ganzzahlen'''
    ergebnis = 0 # Variable für Ergebnis mit Startwert 0
    for i in range(n+1): # für jede Ganzzahl zwischen 0 und n
        ergebnis = ergebnis + i # addiere die Zahl zum Ergebnis
    return(ergebnis) # Ende der Funktion, das Ergebnis wird zurückgegeben
```

Ln: 14 Col: 0

Abbildung 2.46.: Definition von Funktionen in Python

Die zweite Funktion ist schon etwas komplexer. Sie erwartet einen ganzzahligen Übergabewert n und berechnet dann die Summe der ersten n Zahlen. Diesen Wert gibt sie zurück.

Eine Funktion zu definieren, gibt ihr lediglich einen Namen, spezifiziert die Übergabeparameter und strukturiert den Quelltext. Um die Funktion zu verwenden und ausführen zu lassen, muss man sie aufrufen. Dies geschieht durch Angabe des Funktionsnamens gefolgt von Klammern `()`, in den Klammern werden die Übergabeparameter angegeben, falls die Funktion welche erwartet.

Beispiel 2.10 (Aufruf von Funktionen)**.**
Wir können nun unsere im obigen Beispiel 2.9 definierten Funktionen aufrufen. Entweder im Quelltext in einem Editor, oder, nachdem wir einmal „Run Module“ angegeben haben, auch in der interaktiven Python-Shell. Als Parameter können sowohl Werte, als auch Variablen übergeben werden. Rückgabewerte von Funktionen können in Variablen zwischengespeichert werden. Wenn für die Funktion ein Dokumentationsstring angegeben wurde, so wird dieser beim Aufruf der `help()`-Funktion angezeigt (Abb. 2.47).

Python Shell

File Edit Shell Debug Options Windows Help

```
>>> ================================ RESTART ================================
>>>
>>> # Aufruf von mein_print()
>>> mein_print('Hallo Vorkurs!')
Hallo Vorkurs!
>>> x = 7.38
>>> mein_print(x) # Aufruf mit einer Variablen
7.38
>>> summe(5) # Aufruf von summe()
15
>>> var = summe(10) # Speicher Rückgabewert in Variable var
>>> var
55
>>> help(mein_print) # help Funktion gibt Dokumentation aus
Help on function mein_print in module __main__:

mein_print(a)
    Gibt den übergebenen Wert auf dem Bildschirm aus

>>> |
```

Ln: 126 Col: 4

Abbildung 2.47.: Aufruf von Funktionen in Python

Exkurs: Version 4

Unser Programm würde wohl noch wesentlich komfortabler sein, wenn es nicht nur überprüfen würde, ob eine eingegebene Zahlenfolge sortiert ist, sondern wenn es unsortierte Zahlenfolgen sortieren würde. Eine Liste von Zahlen zu sortieren ist sicherlich auch etwas, das in zukünftigen Programmen nützlich sein könnte, daher wollen wir den Sortiervorgang als Funktion schreiben.

Wie sortiert man eine Folge von Zahlen?
Es gibt viele verschiedene Sortieralgorithmen, denn tatsächlich müssen häufig Dinge sortiert werden. Der einfachste und intuitivste ist vermutlich *Insertionsort*. Insertionsort ist das, was die meisten von uns anwenden, wenn sie Spielkarten sortieren. Die meisten arbeiten von links nach rechts, andersherum geht es aber auch. Man geht das Blatt von links nach rechts durch und steckt dabei jede Karte an die richtige Position.

Insertionsort

Der Algorithmus lautet:
1. Für jede Karte von links nach rechts:
   2. Solange der Nachbar links größer ist:
      3. gehe einen weiter nach links
   4. Wenn Nachbar kleiner ist, sortiere Karte rechts davon ein.

Für unser Programm stellen wir uns die Karten als Liste vor. Jede Karte ist ein Element der Liste. Nun durchlaufen wir die Liste von 0 bis zum letzten Element (`liste[len(liste)]`). Für jedes Element vergleichen wir nun schrittweise nach links (in Richtung Anfang der Liste), ob das dortige Element größer ist, als das Element welches wir aktuell einsortieren möchten. Falls ja, verschieben wir das dortige Element um 1 nach rechts, denn unser aktuelles Element wird am Ende links davon stehen und wir benötigen einen Platz. Sobald wir ein Element gefunden haben, das kleiner als unser einzusortierendes Element ist, haben wir die Stelle gefunden an der unser aktuelles Element eingefügt werden muss. Da wir beim Durchgehen der Elemente alle größeren einen nach rechts geschoben haben, können wir das aktuelle Element nun rechts vom gefundenen, kleineren Element einfügen.

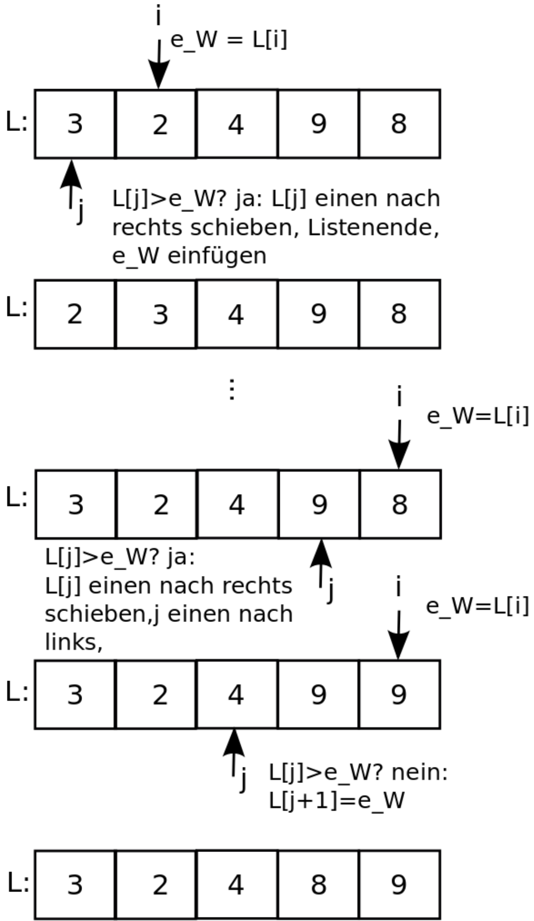


Abbildung 2.48.: Insertionsort

Die Sortierfunktion können wir jetzt für unser Programm nutzen. Wir brauchen nun nicht mehr abzufragen, ob die Eingabe sortiert ist, sondern können einfach nach Beendigung der Eingabe die Folge mithilfe der Funktion sortieren, und dann ausgeben. Unser Programm muss demnach folgende Schritte beinhalten:

1. Lege eine leere Liste an.
2. Solang der Benutzer noch Zahlen eingeben möchte:
   2. Eingabe der Zahl
   3. Überprüfe, ob die Eingabe das Ende der Eingabe signalisiert
      3a. falls **ja**: Schleifenbedingung auf `False` setzen.
      3b. falls **nein**: Eingegebene Zahl an Liste anhängen; und zurück zu 2
4. Wenn Eingabe beendet, Liste sortieren
5. Zahlen ausgeben

Bleibt noch das Problem zu lösen, wie man in einer Liste Elemente „nach rechts schiebt“. Wenn wir unser einzusortierendes Element in einer Variablen zwischenspeichern, dann können wir seinen Wert in der Liste ruhig überschreiben. Somit brauchen wir lediglich jedes Element links von der ursprünglichen Position unseres einzusortierenden Elements eine Listenposition weiter nach rechts zu schreiben. Unser Programm sieht dann wie folgt aus:

```
######################################
# Programm um eine
# Folge von Zahlen zu sortieren.
#
# Version 4
######################################


def sortieren(liste):  #sortieren mit Insertionsort
    #für jedes Element in der Liste
    for i in range(1, len(liste)):
        einzusortierender_wert = liste[i]
        j=i-1
        #bereits sortierte Liste wird durchgegangen und
        #das neue Element an der richtigen Stelle plaziert.
        # Solange die Werte größer als einzusortierender Wert:
        while j >= 0 and liste[j] > einzusortierender_wert:
            liste[j+1] = liste[j] #Wert nach rechts schieben
            j = j - 1 # zum nächsten Wert in sortierter Liste gehen
        # Kleineres Element gefunden, aktuellen Wert davor einfügen
        liste[j+1] = einzusortierender_wert
    #fertig, sortierte Liste zurückgeben
    return liste

print(''' Hallo, in diesem Programm können Sie Zahlen eingeben.
Das Programm sortiert dann, die Zahlenfolge.
Wenn Sie die Eingabe beenden möchten, geben Sie bitte 'e' ein.''')
Eingabe_beenden = False # Variable um zu überprüfen, ob Eingabe beendet wurde
zahlen = [] # leere Liste anlegen, um Benutzereingaben zu speichern
while (Eingabe_beenden == False): # Eingabe, bis Eingabe beendet wird
    zahl = input("Geben Sie eine Zahl ein. ") # Eingabeaufforderung
    if(zahl == 'e'): # Benutzer möchte Eingabe beenden
        Eingabe_beenden = True
        continue # Rest der Schleife überspringen
    elif(zahl.isdigit()):# Überprüfen, ob eine Zahl eingegeben wurde
        zahlen.append(int(zahl)) # Eingabe an Zahlenfolge anhängen
# Eingabe ist beendet, jetzt sortieren und die Liste ausgeben
zahlen = sortAlgo.sortieren(zahlen)
print(zahlen)
```

Abbildung 2.49.: Sortierte Zahlenfolge: Version 4

Die Ausgabe unseres Programms zeigt, dass es nun tatsächlich eingegebene Zahlenfolgen sortiert.

```
 Hallo, in diesem Programm können Sie Zahlen eingeben.
Das Programm sortiert dann, die Zahlenfolge.
Wenn Sie die Eingabe beenden möchten, geben Sie bitte 'e' ein.
Geben Sie eine Zahl ein. 9
Geben Sie eine Zahl ein. 3
Geben Sie eine Zahl ein. 6
Geben Sie eine Zahl ein. 7
Geben Sie eine Zahl ein. 2
Geben Sie eine Zahl ein. e
[2, 3, 6, 7, 9]
>>>
 Hallo, in diesem Programm können Sie Zahlen eingeben.
Das Programm sortiert dann, die Zahlenfolge.
Wenn Sie die Eingabe beenden möchten, geben Sie bitte 'e' ein.
Geben Sie eine Zahl ein. 7
Geben Sie eine Zahl ein. 8
Geben Sie eine Zahl ein. 1
Geben Sie eine Zahl ein. 45
Geben Sie eine Zahl ein. e
[1, 7, 8, 45]
>>>
```

Abbildung 2.50.: Durchläufe des Programms Version 4

2.8. Gültigkeitsbereiche

lokale Variable

globale Variable

Variablen, die in einer Funktion definiert werden, sind in keinster Weise mit andern Variablen des gleich Namens, die außerhalb der Funktion benutzt werden, verbunden. Man sagt, die Variablen sind *lokal* gültig. Im Gegensatz dazu gibt es *global* gültige Variablen, die außerhalb von Funktionen definiert werden und dort gültig sind. Dies nennt man den *Gültigkeitsbereich* einer Variablen. Eine

Variable ist genau in dem Anweisungsblock gültig, in dem sie definiert wird, beginnend an der Stelle wo die erste Zuweisung stattfindet.

Beispiel 2.11 (Gültigkeitsbereiche)**.**
In folgendem Beispiel definieren wir außerhalb einer Funktion eine Variable `x`, mit dem Wert 50. Außerdem definieren wir eine Funktion `eine_Funktion()` der ein Wert übergeben werden kann. Dieser Wert wird dann ausgegeben. Anschließend wird in der Funktion eine Variable mit Namen `x` und Wert `2` definiert, deren Wert ebenfalls ausgegeben wird (Abb. 2.51). Unser Programm ruft nun die Funktion `eine_Funktion` auf. Danach wird der aktuelle Wert der Variablen `x` nochmals ausgegeben.

scope.py - /home/ronja/Dropbox/Skript/Code/scope.py

File Edit Format Run Options Windows Help

```
# Beispiel für Gültigkeitsbereiche

def eine_funktion(x):  # Funktion, die einen Übergabeparameter erwartet
    print(x) # gib aktuellen Wert von x aus
    x=2 # Lokale Variable x mit Wert 2
    print(x) # gib aktuellen Wert von x aus
    return # Funktionsende

x = 50  # Variable x mit Wert 50
print('Aufruf von Funktion eine_Funktion')
eine_funktion(x) # Aufruf von eine_Funktion mit x (Wert 50)
print('Funktion abgeschlossen') # Meldung, über Beendigung der Funktion
print(x) # gib aktuellen Wert von x aus
```

Ln: 14 Col: 0

Abbildung 2.51.: Gültigkeitsbereiche

Wenn das Programm gestartet wird, verwendet Python beim ersten Zugriff auf den *Wert* mit Namen `x` in der Funktion den Wert des gleichnamigen Parameters; welcher 50 ist. Danach weisen wir der Variablen `x` den Wert 2 zu. Da wir uns in einer Funktion befinden, legt Python eine lokale Variable mit Namen `x` an. Der Wert der globalen Variablen `x` wird davon nicht beeinflusst. Innerhalb der Funktion gibt es nun einen neuen Wert mit Namen `x` und zwar 2. Daher wird bei der folgenden `print()`-Anweisung 2 ausgegeben. Mit der `print()`-Anweisung nach Beendigung der Funktion vergewissern wir uns, dass der Wert der *globalen* Variable `x` von der Zuweisung innerhalb der Funktion nicht betroffen war. Es wird 50 ausgegeben.

Python Shell

File Edit Shell Debug Options Windows Help

```
>>> ================================ RESTART ================================
>>> 
Aufruf von Funktion eine_Funktion
50
2
Funktion abgeschlossen
50
>>> 
```

Ln: 139 Col: 4

Abbildung 2.52.: Ausgabe des Programms zu Gültigkeitsbereichen

Möchte man einem außerhalb der Funktion definierten Namen in einer Funktion einen Wert zuweisen, so muss man Python mitteilen, dass der Name nicht lokal, sondern global ist. Dies geschieht mithilfe des `global`-Befehls. Ohne die `global`-Anweisung wird Python bei einer Zuweisung innerhalb einer Funktion immer eine lokale Variable anlegen.

Die Werte von Variablen, die außerhalb einer Funktion definiert wurden, können innerhalb einer Funktion zwar benutzt werden (solange innerhalb der Funktion keine gleichnamige Variable

`global`

existiert), es gehört aber nicht zum guten Stil und sollte vermieden werden, da einem Leser des Programms nicht sofort klar ist, wo diese Variable definiert wurde.

Beispiel 2.12 (`global`-Anweisung)**.**
Wir benutzen ein ganz ähnliches Programm wie in Beispiel 2.11. Allerdings nimmt die Funktion diesmal keinen Wert entgegen. Die `global`-Anweisung wird genutzt, um `x` innerhalb der Funktion als globale Variable zu definieren (Abb. 2.53).

```
global.py - /home/ronja/Dropbox/Skript/Code/global.py
File  Edit  Format  Run  Options  Windows  Help
# Beispiel für global-Anweisung

def eine_funktion():  # Funktion, die keinen Übergabeparameter erwartet
    global x # Variable x ist in Funktion global
    print(x) # gib aktuellen Wert von x aus
    x=2 # Weise x den Wert 2 zu
    print(x) # gib aktuellen Wert von x aus
    return # Funktionsende

x = 50  # Variable x mit Wert 50
print('Aufruf von Funktion eine_Funktion')
eine_funktion() # Aufruf von eine_Funktion
print('Funktion abgeschlossen') # Meldung, über Beendigung der Funktion
print(x) # gib aktuellen Wert von x aus

Ln: 15 Col: 0
```

Abbildung 2.53.: Verwendung der `global`-Anweisung

Nun wird bei der Zuweisung von 2 an die Variable `x` keine lokale Variable angelegt, sondern der Wert der globalen Variablen `x` wird überschrieben. Die Ausgabe des Wertes von `x` nach Beendigung der Funktion bestätigt das. Es wird 2 ausgegeben (Abb. 2.54).

```
Python Shell
File  Edit  Shell  Debug  Options  Windows  Help
>>> ================================ RESTART ================================
>>>
Aufruf von Funktion eine_Funktion
50
2
Funktion abgeschlossen
2
>>> 

Ln: 146 Col: 4
```

Abbildung 2.54.: Ausgabe des Programms zur `global`-Anweisung

2.9. Module

Wir haben mit Funktionen eine Möglichkeit kennengelernt, Quelltext innerhalb eines Programms mehrmals zu verwenden. Es gibt aber in Python auch die Möglichkeit, bereits vorhandene Funktionen aus einem anderen Programm wiederzuverwenden. Dies geschieht mithilfe von *Modulen*. Ein Modul ist eine Datei, die Python-Quelltext enthält. Um ein Modul in anderen Programmen wiederverwenden zu können, *muss* der Dateiname die Endung `.py` haben.

`import`-Anweisung

`import`

Mithilfe der `import`-Anweisung kann eine Modul in ein Python-Programm importiert werden.

Die Funktionen, Variablen, usw die in dem Modul definiert werden, können dann im Programm verwendet werden. Um Namenskonflikte zu vermeiden, muss eine, aus einem Modul importierte, Funktion oder Variable mit dem Modulnamen angegeben werden (Bsp. 2.13).

Beispiel 2.13 (`import`-Anweisung)**.**
Angenommen wir schreiben ein Programm und stellen fest, dass wir im Verlauf des Programms die Summe der ersten n Ganzzahlen berechnen müssen. Zum Glück erinnern wir uns, dass wir solch eine Funktion ja bereits geschrieben haben, nämlich in Beispiel 2.9. Die Funktion ist in einer Datei namens `funktion.py` gespeichert. Diese können wir nun als Modul in unser aktuelles Programm importieren und die Funktion `summe()` verwenden (Abb. 2.55).

import.py - /home/ronja/Dropbox/Skript/Code/import.py

File Edit Format Run Options Windows Help

```
# Beispiel für import-Anweisung

import funktion # importiere Modul funktion

# Nun können die in funktion definierten Funktionen hier aufgerufen werden
# Gib den Rückgabewert der Funktion summe aus Modul funktion aus
print(funktion.summe(10))
# Funktion mein_print aus Modul funktion
funktion.mein_print('Und nun zu etwas ganz anderem...')
```

Ln: 10 Col: 0

Abbildung 2.55.: Verwendung der `import`-Anweisung

Werden Module auf diese Art und Weise importiert, so kommt es nicht zu Namenskonflikten. Es kann in unserem Programm z.B. ebenfalls eine Funktion `mein_print()` definiert sein, da aber die `mein_print()`-Funktion aus Modul `funktion` als `funktion.mein_print()` aufgerufen werden muss, kann Python die beiden Funktionen auseinanderhalten (Abb. 2.56).

namen.py - /home/ronja/Dropbox/Skript/Code/namen.py

File Edit Format Run Options Windows Help

```
# Beispiel für import-Anweisung

import funktion # importiere Modul funktion

def mein_print(a):  # Funktion mein_print() erwartet einen Parameter
    print('Ausgabe aus Funktion mein_print:', a)
    return


# Nun können die in funktion definierten Funktionen hier aufgerufen werden
# Gib den Rückgabewert der Funktion summe aus Modul funktion aus
print(funktion.summe(10))

bsp = 'Und nun zu etwas ganz anderem...'
# Funktion mein_print aus Modul funktion
funktion.mein_print(bsp)
# Funktion mein_print
mein_print(bsp)
```

Ln: 19 Col: 0

Abbildung 2.56.: Verwendung der `import`-Anweisung

Beim Aufruf von `mein_print()` wird die im Programm definierte Funktion aufgerufen, beim Aufruf von `funktion.mein_print()` wird die im Modul `funktion` definierte Funktion aufgerufen (Abb. 2.57).

```
Python Shell
File Edit Shell Debug Options Windows Help
>>> ================================ RESTART ================================
>>>
55
Und nun zu etwas ganz anderem...
Ausgabe aus Funktion mein_print: Und nun zu etwas ganz anderem...
>>>
Ln: 167 Col: 4
```

Abbildung 2.57.: Ausgabe des Programms zur `import`-Anweisung

`from...import`-Anweisung

Wenn man nicht bei jedem Funktionsaufruf den Modulnamen mit angeben möchte, kann man die `from...import`-Anweisung benutzen. Mit ihr können einzelne Funktionen und Variablen aus einem Modul in ein Programm importiert werden.

Beispiel 2.14 (`from ...import`-Anweisung)**.**
Der Befehl `from funktion import summe` importiert lediglich die Funktion `summe()` aus dem Modul `funktion`. Vorteil dieser Art des Imports ist, dass nun die Funktion `summe()` verwendet werden kann, ohne dass der Modulname vorweggestellt werden muss (Abb. 2.58).

```
from.py - /home/ronja/Dropbox/Skript/Code/from.py
File Edit Format Run Options Windows Help
# Beispiel für from...import-Anweisung

from funktion import summe # importiere Funktion summe aus Modul funktion

def mein_print(a):  # Funktion mein_print() erwartet einen Parameter
    print('Ausgabe aus Funktion mein_print:', a)
    return


# Nun kann die Funktion summe aufgerufen werden
# Gib den Rückgabewert der Funktion summe aus Modul funktion aus
print(summe(10))

bsp = 'Und nun zu etwas ganz anderem...'
# Funktion mein_print
mein_print(bsp)

Ln: 17 Col: 0
```

Abbildung 2.58.: Verwendung der `from...import`-Anweisung

Die Funktion `summe()` kann problemlos verwendet werden, auf die Funktion `mein_print()` aus Modul `funktion` kann jedoch nicht mehr zugegriffen werden. Lediglich die im Programm selbst definierte Funktion `mein_print()` steht zur Verfügung.

```
Python Shell
File Edit Shell Debug Options Windows Help
>>> ================================ RESTART ================================
>>>
55
Ausgabe aus Funktion mein_print: Und nun zu etwas ganz anderem...
>>>
Ln: 178 Col: 4
```

Abbildung 2.59.: Verwendung der `from...import`-Anweisung

Der Befehl `from funktion import *` importiert alle im Modul `funktion` definierten Funktionen und Variablen. Das erscheint auf den ersten Blick die bequemste Variante zu sein, Funktionen etc. aus anderen Programmen zu importieren. Allerdings werden bei dieser Methode namensgleiche Funktionen überschrieben (Bsp 2.15). D.h. falls im Programm *vor* dem `import *`-Befehl namensgleiche Funktionen definiert wurden, werden bei Funktionsaufruf nun Funktionen aus dem Modul aufgerufen. Falls *nach* dem `import *`-Befehl namensgleiche Funktionen im Programm definiert werden, werden bei Funktionsaufruf nur noch die im Programm definierten aufgerufen. Da man vor allem bei einem großen Programm schnell den Überblick über verwendete Namen verliert und es nicht einfach ist zu überschauen welche Variablen-, Funktionsnamen etc. in dem importierten Modul existieren, kann man dabei böse Überraschungen erleben, die einen letztendlich doch mehr Mühe kosten, als vor jeden Funktionsaufruf den Modulnamen zu schreiben, bzw nur explizit die Funktionen zu importieren, die tatsächlich gebraucht werden.

Beispiel 2.15 (`from...import *`-Anweisung)**.**

stern.py - /home/ronja/Dropbox/Skript/Code/stern.py

File Edit Format Run Options Windows Help

```
# Beispiel für from...import * -Anweisung

def mein_print(a):  # Funktion mein_print() erwartet einen Parameter
    print('Ausgabe aus Funktion mein_print:', a)
    return

from funktion import * # importiere alles auf Modul funktion

# Nun kann die Funktion summe aufgerufen werden
# Gib den Rückgabewert der Funktion summe aus Modul funktion aus
print(summe(10))

bsp = 'Und nun zu etwas ganz anderem...'
# Funktion mein_print
mein_print(bsp)
```

Ln: 16 Col: 0

Abbildung 2.60.: Verwendung der `from...import *`-Anweisung

In diesem Beispiel werden nach der Funktionsdefinition von `mein_print()` alle Funktionen etc. aus Modul `funktion` importiert. Da Modul `funktion` ebenfalls eine Funktion namens `mein_print` besitzt, wird die zuvor definierte Funktion überschrieben (Abb. 2.61).

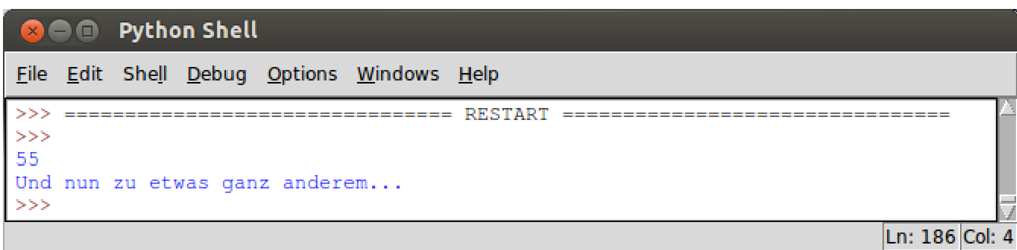


Abbildung 2.61.: Ausgabe des Programms zur Verwendung der `from...import *`-Anweisung

2.10. Datei Ein- und Ausgabe

Bisher haben wir Daten immer über den Bildschirm (die Standardein- und -ausgabe) eingelesen und ausgegeben. Bei größeren Datenmengen wollen wir die Daten aus einer Datei einlesen und die Ergebnisse auch wieder in eine Datei schreiben können. Python stellt diverse Funktionen zur Verfügung um auf Dateien zuzugreifen.

open()

open(): Der `open()`-Befehl öffnet eine Datei. Er erwartet als Übergabeparameter den Namen der zu öffnenden Datei. Falls sich diese nicht im aktuellen Verzeichnis befindet, muss zusätzlich der Dateipfad (siehe A.1.1) angegeben werden. Ferner muss angegeben werden, in welchem *Modus* die Datei geöffnet werden soll. Es gibt diverse Modi in denen die Datei geöffnet werden kann, wir betrachten hier nur die wichtigsten und verweisen auf die Python-Dokumentation[2] für die andern.

Modus	Beschreibung
r	öffnet die Datei im Lesemodus. Es kann lediglich gelesen, aber nicht geschrieben werden. Falls die Datei nicht existiert, wird eine Fehlermeldung ausgegeben.
w	öffnet die Datei im Schreibmodus. Falls die Datei nicht existiert, wird sie angelegt. ACHTUNG! Falls die Datei bereits existiert, wird sie überschrieben.
a	öffnet die Datei um Daten hinzuzuschreiben. Es kann lediglich geschrieben, aber nicht gelesen werden. Falls die Datei nicht existiert, wird sie neu angelegt. Ansonsten werden Daten am Ende der Datei hinzugefügt.

write()

write(): Die `write()`-Methode schreibt jede Zeichenkette, die ihr übergeben wird, in die geöffnete Datei. Die `write()`-Methode fügt keinen Zeilenumbruch am Ende des Strings ein (anders als es z.B. die `print`-Funktion macht). Zeilenumbrüche müssen extra übergeben werden (`'\n'`). Sie kann nur angewendet werden, wenn die Datei im Schreib- oder Hinzufügmodus geöffnet wurde.

read()

read(): Die `read()`-Methode liest eine Zeichenkette aus einer Datei. Wenn kein Parameter übergeben wird, der festlegt wie viele *Bytes* gelesen werden sollen, so liest die Methode bis zum Dateiende. Sie kann nur angewendet werden, wenn die Datei im Lesemodus geöffnet wurde.

close(): Die `close()`-Methode schließt die Datei wieder. Python schließt eine Datei zwar automatisch, sobald ihr Referenzobjekte (ähnlich wie Variable) zerstört wird, es gehört aber zum guten Stil eine geöffnete Datei, sobald sie nicht mehr gebraucht wird, mit der `close()`-Methode zu schließen.

Beispiel 2.16 (Dateizugriff)**.**
Znächst legen wir eine Datei mit Name `foo.txt` an und öffnen diese im Schreibmodus. Dann schreiben wir ein paar Zeilen in die Datei. Da die `write()`-Methode keine Zeilenumbrüche einfügt, müssen wir uns selber darum kümmern. Wir können entweder Zeilenumbrüche als Zeichenfolge `\n` direkt eingeben, oder mit drei Anführungszeichen arbeiten. Strings mit drei Anführungszeichen können mehrzeilig sein und der Zeilenumbruch wird intern eingefügt.

Nachdem wir in die Datei geschrieben haben, schließen wir sie wieder und öffnen sie im Lesemodus. Wir lesen die ersten 10 Byte der Datei und geben diese auf dem Bildschirm aus. Anschließend geben wir die Datei zeilenweise aus. Die Zeilen einer Datei können dabei wie eine Sequenz mit einer `for`-Schleife durchlaufen werden (Abb. 2.62).

[2] http://docs.python.org/3.3/library/function.html

datei.py - /home/ronja/Dropbox/Skript/Code/datei.py

File Edit Format Run Options Windows Help

```
# Dateizugriff

# öffne Datei foo.txt im Schreib- und Lesemodus. Falls sie nicht existiert,
#  lege sie an.
fo = open("/home/ronja/Documents/foo.txt","w")
#Auf die Datei kann nun über den Namen fo zugegriffen werden.
#Schreibe etwas in die Datei
fo.write("Nun zieren Sie sich doch nicht so! Sie haben einen Spenderausweis! \n")
#Schreibe noch etwas, auch mehrzeilig ohne explizite Angabe von \n
fo.write('''-Aber das ist doch nur im Todesfall!
Keine Angst. Das hat noch keiner überlebt, wenn wir ihm die Leber
rausgenommen haben''')
# Datei schließen
fo.close()
# Datei zum Lesen öffnen
ro = open("/home/ronja/Documents/foo.txt","r")
# lies die ersten 10 byte
anfang = ro.read(10)
x = len(anfang)
# gib das Gelesene auf dem Bildschirm aus
print(anfang)
# lies die Datei Zeile für Zeile und gib jede Zeile auf dem Bildschirm aus.
for line in ro:
    print(line)
# Datei schließen
ro.close()
```

Ln: 27 Col: 0

Abbildung 2.62.: Dateizugriff in Python

Bei Betrachtung der Programmausgabe fällt auf, dass die zeilenweise Ausgabe des Dateiinhalts nicht am Anfang der Datei gestartet ist (Abb. 2.64). Das liegt daran, dass beim Öffnen einer Datei im Lese- oder Schreibmodus ein Zeiger auf den Dateianfang gesetzt wird. Beim Schreiben und Lesen der Datei wird der Zeiger weiterbewegt. Da wir zuvor bereits die ersten 10 Byte gelesen hatten, steht der Zeiger zu Beginn des zeilenweisen Auslesens bereits auf dem elften Byte. Dementsprechend wird von dort gelesen.

Python Shell

File Edit Shell Debug Options Windows Help

```
>>> ================================ RESTART ================================
>>>
Nun zieren
 Sie sich doch nicht so! Sie haben einen Spenderausweis!

-Aber das ist doch nur im Todesfall!

Keine Angst. Das hat noch keiner überlebt, wenn wir ihm die Leber

rausgenommen haben
>>>
```

Ln: 289 Col: 4

Abbildung 2.63.: Ausgabe des Programms zum Dateizugriff in Python

Exkurs: Version 5

Eine weitere Verbesserung unseres Programms wäre, wenn die zu sortierenden Zahlen nicht per Hand eingegeben werden müssten, sondern aus einer Datei eingelesen würden. Hilfreich wäre es auch, wenn die sortierte Folge nicht nur ausgegeben, sondern wieder zurück in eine Datei geschrieben würde. Unser Programm muss dafür folgende Arbeitsschritte ausführen.

1. Namen der Datei, in der sich die Daten befinden erfragen
2. Datei zum lesen öffnen
3. Datei lesen und jede Zahl in Liste speichern
4. Datei schließen
5. Liste sortieren
6. Sortierte Liste ausgeben
6. Datei zum Schreiben öffnen
7. Zahlen aus Liste in Datei schreiben
8. Datei schließen

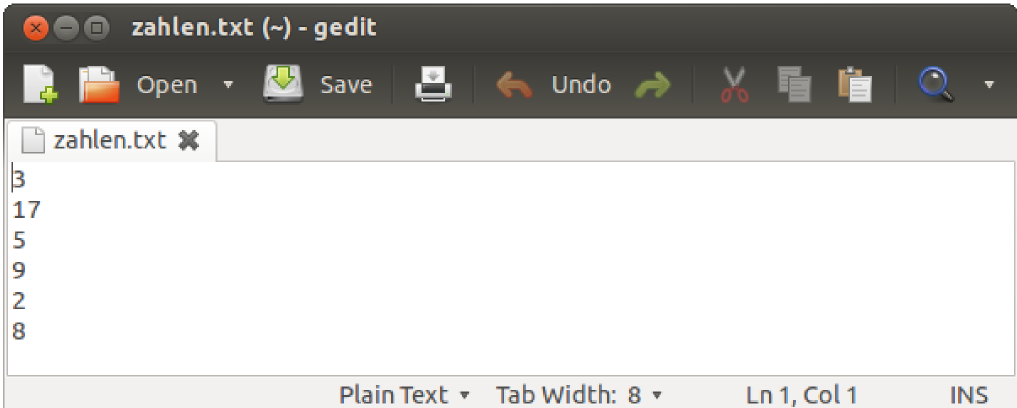


Abbildung 2.64.: Beispieldatei mit Zahlen

Nun müssen wir uns noch klarmanchen, dass die Datei, deren Inhalt für uns wie untereinander geschriebene Zahlen aussieht, für den Computer mehr Informationen enthält. Am Ende jeder Zeile steht nämlich noch ein Zeilenumbruch (`\n`). Dieser wird als Zeichen mitgelesen und wir müssen ihn erst abtrennen, bevor wir die als `string` eingelesene Zahl in einen `int`-Wert umwandeln können, um ihn in die Liste einzufügen. Außerdem haben wir unsere Sortierfunktion in ein Modul `sortAlgo` geschrieben. Dieses können wir nun in unser Programm importieren und die Sortierfunktion nutzen. Unser Programm sieht dann wie folgt aus:

```
sort5.py - /home/ronja/Uni/Lernzentrum/Vorkurs/WS1516/Material/S ipt/Code/s
File Edit Format Run Options Windows Help
#####################################
# Programm um Zahlen aus einer Datei
# einzulesen, zu sortieren und
# zurückzuschreiben.
#
# Version 5
#####################################

import sortAlgo # importiere das Modul mit den Sortierfunktionen

print('''Diese Programm liest aus einer Datei die Zahlen aus,
sortiert diese und schreibt sie sortiert in die Datei zurück.
Dabei muss jede Zahl in einer eigenen Zeile stehen''')

datei_name = input('''Geben sie den Namen der Datei ein,
deren Inhalt eingelesen werden soll: ''')

zahlen = [] # Liste für Zahlen anlegen
datei = open(datei_name, "r") # Datei öffnen
for line in datei: #für jede Zeile
    # Eintrag lesen, Zeilenumbruch abschneiden, in Zahl umwandeln, an Liste anhä
    zahlen.append(int(line.replace("\n","")))

datei.close() #Datei schließen
zahlen = sortAlgo.sortieren(zahlen) #Liste sortieren
print(zahlen) #Sortierte Folge ausgeben

datei = open(datei_name, "w") #Datei öffnen
for i in zahlen: # Für jedes Element in zahlen
    datei.write(str(i) + "\n") #in Datei schreiben, Zeilenumbruch hinzufügen
#Datei schließen
datei.close()

Ln: 33 Col: 0
```

Abbildung 2.65.: Sortierte Zahlenfolge: Version 5

Wenn wir nun eine Datei, die aus untereinander geschriebenen Zahlen besteht, anlegen, so können wir unser Programm ausprobieren.

```
Python Shell
File Edit Debug Options Windows Help
>>>
Diese Programm liest aus einer Datei die Zahlen aus,
sortiert diese und schreibt sie sortiert in die Datei zurück.
Dabei muss jede Zahl in einer eigenen Zeile stehen
Geben sie den Namen der Datei ein,
deren Inhalt eingelesen werden soll: /home/ronja/zahlen.txt
[2, 3, 5, 8, 9, 17]
>>>
Ln: 89 Col: 4
```

Abbildung 2.66.: Durchlauf des Programms, Version 5

3. Rekursive Programmierung

Alle Programme die wir bisher betrachtet haben, sind *iterative* Programme. Sie arbeiten mit Schleifen und wiederholen Anweisungen und Anweisungsfolgen. In der *rekursiven* Programmierung wird mit Selbstaufrufen gearbeitet. Eine Funktion definiert sich durch sich selbst. D.h. eine Funktion ruft sich in ihrem Funktionskörper ein- oder mehrmals selbst auf. Damit man eine rekursive Funktion in der Programmierung benutzen kann, muss man sicher gehen, dass sie irgendwann auch einmal beendet wird und sich nicht nur endlos selber aufruft. Dafür benötigt die Funktion eine *Abbruchbedingung*. Meist ist das ein Wert bei dem kein erneuter Funktionsaufruf gestartet wird, sondern ein Wert zurückgegeben wird. Eine rekursive Funktion terminiert, wenn es eine solche Abbruchbedingung gibt und sicher gestellt ist, dass diese irgendwann erreicht wird.

Abbruch-bedingung

3.1. Rekursive Berechnung der Summe

Beispiel 3.1 (rekursive Funktion)**.**
Erinnern wir uns an die Funktion `summe()` aus Beispiel 2.9. Diese Funktion lässt sich auch rekursiv programmieren, denn die Summe der ersten n ganzen Zahlen, ist nichts anderes als die Summe der ersten $n-1$ ganzen Zahlen $+n$. Die Abbruchbedingung wäre dann der Aufruf `summe(1)`, denn die Summe von 1 ist 1. In Abbildung 3.1 sind die iterativ und die rekursiv programmierte Funktion gemeinsam dargestellt.

```
rekursion.py - /home/ronja/Dropbox/Skript/Code/rekursion.py
File  Edit  Format  Run  Options  Windows  Help

# Beispiel f\"ur rekursive Funktion

#  Funktion iterativ
def summe(n):
    '''berechnet die Summe der ersten n Ganzzahlen'''
    ergebnis = 0 # Variable für Ergebnis mit Startwert 0
    for i in range(n+1): # für jede Ganzzahl zwischen 0 und n
        ergebnis = ergebnis + i # addiere die Zahl zum Ergebnis
    return(ergebnis) # Ende der Funktion, das Ergebnis wird zurückgegeben


# Funktion rekursiv
def rek_summe(n):
    '''berechnet die Summe der ersten n Ganzzahlen rekursiv'''
    if(n == 1): # Abbruchbedingung
        return(1)
    else:
        return(rek_summe(n-1)+n) # rekursiver Aufruf

Ln: 19 Col: 0
```

Abbildung 3.1.: Summation über die ersten n Ganzzahlen;
iterativ und rekursiv programmiert

Die beiden Funktionen berechnen das gleiche. Für die gleichen Eingabewerte liefern sie den gleichen Rückgabewert (Abb. 3.2).

```
Python Shell
File Edit Shell Debug Options Windows Help
>>> ================================ RESTART ================================
>>>
>>> summe(5) # iterative Funktion
15
>>> rek_summe(5) # rekursive Funktion
15
>>> summe(1)
1
>>> rek_summe(1)
1
>>> summe(137)
9453
>>> rek_summe(137)
9453
>>>
Ln: 18 Col: 4
```

Abbildung 3.2.: Programmausgabe für Summation über die ersten n Ganzzahlen; iterativ und rekursiv programmiert

Was genau bei der rekursiven Funktion passiert, sieht man erst, wenn wir noch ein paar Zwischenausgaben hinzufügen (Abb. 3.3).

```
rekursion.py - /home/ronja/Dropbox/Skript/Code/rekursion.py
File Edit Format Run Options Windows Help
# Beispiel f\"ur rekursive Funktion

#  Funktion iterativ
def summe(n):
    '''berechnet die Summe der ersten n Ganzzahlen'''
    ergebnis = 0 # Variable für Ergebnis mit Startwert 0
    for i in range(n+1): # für jede Ganzzahl zwischen 0 und n
        ergebnis = ergebnis + i # addiere die Zahl zum Ergebnis
    return(ergebnis) # Ende der Funktion, das Ergebnis wird zurückgegeben


# Funktion rekursiv
def rek_summe(n):
    '''berechnet die Summe der ersten n Ganzzahlen rekursiv'''
    print("rek_summe aufgerufen mit n =", n) # Ausgabe des Parameters
    if(n == 1): # Abbruchbedingung
        return(1)
    else:
        erg = (rek_summe(n-1)+n) # rekursiver Aufruf
        print("Zwischenergebnis für" , n, "+ rek_summe(", n-1,"):",erg)
        return(erg)
Ln: 22 Col: 0
```

Abbildung 3.3.: Summation über die ersten n Ganzzahlen; iterativ und rekursiv programmiert

Ruft man die Funktion nun mit dem Wert 5 auf, erhält man folgende Ausgabe:

```
Python Shell
File Edit Shell Debug Options Windows Help
>>> ================================ RESTART ================================
>>>
>>> rek_summe(5)
rek_summe aufgerufen mit n = 5
rek_summe aufgerufen mit n = 4
rek_summe aufgerufen mit n = 3
rek_summe aufgerufen mit n = 2
rek_summe aufgerufen mit n = 1
Zwischenergebnis für 2 + rek_summe( 1 ): 3
Zwischenergebnis für 3 + rek_summe( 2 ): 6
Zwischenergebnis für 4 + rek_summe( 3 ): 10
Zwischenergebnis für 5 + rek_summe( 4 ): 15
15
>>>
Ln: 31 Col: 4
```

Abbildung 3.4.: Ausgabe der ausführlichen rekursiven Funktion zur Summation über die ersten n Ganzzahlen

Die Funktion wird nacheinander für alle Werte von 5 bis 1 aufgerufen, und erhält die Zwischenergebnisse für die Aufrufe in umgekehrter Reihenfolge zurück. Aus den Zwischenergebnissen setzt sich dann das Endergebnis zusammen.

3.2. Fibonacci-Zahlen

Beispiel 3.2 (Fibonacci-Zahlen)**.**
Die Fibonacci-Zahlen sind eine unendliche Zahlenfolge. Daher werden sie häufig auch als Fibonacci-Folge bezeichnet. Der theoretische Hintergrund der Fibonacci-Folge wird in Beispiel 12.15 behandelt. Hier wollen wir zeigen, wie wir eine rekursive Funktion programmieren können, die die n-te Fibonacci-Zahl berechnet. Mathematisch sind die Fibonacci-Zahlen wie folgt definiert:

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

Fibonacci Folge

Sie sind bereits rekursiv definiert. Die Funktion lässt sich ziemlich leicht in eine Python-Funktion übertragen (Abb. 3.5).

```
fib.py - /home/ronja/Dropbox/Skript/Code/fib.py
File Edit Format Run Options Windows Help
# Rekursive Funktion zur Berechnung der Fibonacci-Zahlen

def fib(n):
    if(n == 1 or n == 2): # Abbruchbedingung
        return 1
    else:
        return  fib(n-1) + fib(n-2) # rekursiver Aufruf


print('fib(5)  =',fib(5))
print('fib(10)  =',fib(10))
print('fib(20)  =',fib(20))
print('fib(35)  =',fib(35))

Ln: 14 Col: 0
```

Abbildung 3.5.: Rekursive Berechnung der Fibonacci-Zahlen

Wenn wir obiges Programm ausführen, erhalten wir folgende Ausgabe:

```
Python Shell
File  Edit  Shell  Debug  Options  Windows  Help
>>> ================================ RESTART ================================
>>>
fib(5) = 5
fib(10) = 55
fib(20) = 6765
fib(35) = 9227465
>>>
                                                        Ln: 649 Col: 4
```

Abbildung 3.6.: Ausgabe des Programms zur rekursiven Berechnung der Fibonacci-Zahlen

Allerdings fällt bei der Ausführung auf, dass das Ergebnis des Funktionsaufrufs `fib(35)` sekundenlang auf sich warten lässt. Beim Aufruf von `fib(40)` wartet man bereits minutenlang auf das Ergebnis.

Die Erklärung hierfür liegt in der Rekursion. Jeder Aufruf von `fib(n)` mit $n > 2$ bewirkt zwei weitere Aufrufe der Funktion, welche erneut je zwei Aufrufe bewirken, usw.... So entsteht für den Aufruf `fib(5)` bereits ein Rekursionsbaum mit 9 Funktionsaufrufen (Abb. 3.7).

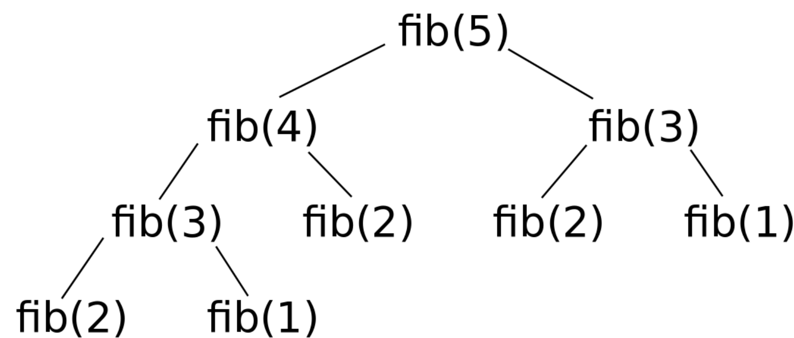


Abbildung 3.7.: Rekursionsbaum für Aufruf von `fib(5)`

Die Berechnung `fib(2)` wird gleich dreimal, die Berechnung von `fib(3)` zweimal ausgeführt. Viele Dinge werden also mehrfach ausgerechnet, das macht das Programm so langsam. Wenn man allerdings die Ergebnisse zwischenspeichert und auf bereits berechnete Zwischenergebnisse zurückgreift, ist die rekursive Implementierung (Programmierung) sehr schnell.

Rekursive Programmierung ist bei vielen Problemen der einfachere Weg zu einer Lösung, da viele Prozesse eine rekursive Struktur haben und unser Denken auf rekursiven Denkprozessen basiert. Allerdings muss man vorsichtig bei der Programmierung sein. Wie das Beispiel der Fibonacci-Zahlen zeigt, kann unüberlegte Rekursion auch sehr ineffizient sein.

4. Debugging

Fehler in einem Computerprogramm werden in der Fachsprache auch *bugs* genannt. Der Begriff wurde von amerikanischen Ingenieuren lange vor dem ersten Computer benutzt, um Fehlverhalten ihrer Erfindungen zu bezeichnen. So verwendete z.B. Thomas Edison den Begriff in einem Brief an einen Freund im Jahr 1878.

> *"The first step [in all of my inventions] is an intuition, and comes with a burst, then difficulties arise - this thing gives out and [it is] then that 'Bugs' - as such little faults and difficulties are called - show themselves [...]."*

Debugging

Debugging ist der Prozess Fehler im Programmcode zu finden und diese zu eliminieren. Programmieranfängerinnen und -anfänger machen naturgemäß besonders viele Fehler, aber auch erfahrene Programmierer sind davor nicht gefeit. Eine Studie des iX-Magazins veröffentlicht in 1/2006 ermittelte, dass Softwarefehler deutschen Mittel- und Großbetrieben jährlich rund 84,4 Mrd Euro Verlust bescheren, rund 14,4 Mrd Euro jährlich für die Beseitigung von Programmierfehlern verwendet wird und Computerausfälle aufgrund fehlerhafter Software Produktivitätsverluste von ca 70 Mrd Euro verursachen.

In diesem Kapitel werden wir über verschiedene Arten von Fehlern berichten, typische Programmierfehler beschreiben und Tipps geben, wie man Fehler finden und korrigieren kann.

So anstrengend und nervenaufreibend die Fehlersuche auch sein kann, es gibt wenig Tätigkeiten die so spannend und intellektuell anspruchsvoll sind, wie Programmierfehler zu finden und zu korrigieren. Abgesehen davon, dass die Fehlersuche sehr charakterbildend (Gelassenheit, Frustrationstoleranz, Durchhaltevermögen) sein kann, wird man dabei auch eine besserere Programmiererin bzw ein besserer Programmierer, denn häufig entstehen Fehler dadurch, dass man einen Aspekt der Programmiersprache doch noch nicht so ganz verstanden hatte.

Programmierfehler lassen sich grob in drei Gruppen aufteilen. Solche, bei denen Python nicht versteht was es tun soll, solche, bei denen Python zwar versteht was es tun soll, aber bei der Ausführung Probleme auftreten und solche, bei denen Python keinerlei Probleme mit der Ausführung hat, aber nicht das tut, was wir eigentlich wollten.

4.1. Syntaxfehler

Syntaxfehler

Die erste Art von Fehler heißen *Syntaxfehler*. Python versteht nicht was es tun soll, weil der Code kein gültiger Python Ausdruck ist. Beispiele für Syntaxfehler in der Deutschen Sprache wäre z.B. der Satz *"Bitte Oma Kuchen."*, denn es fehlen Prädikat und Subjekt. Formal betrachtet enthalten auch die Sätze , *"bitte bring Oma Kuchen."* oder *"Bitte iss Omas Kuchen"*, Syntaxfehler, denn ein Satz muss mit einem Großbuchstaben beginnen und mit einem Punkt (.) enden.

In Python treten Syntaxfehler z.B. bei der Verwendung nicht definierter Zeichen `5 ~ 6`, fehlenden Argumenten zwischen Operatoren `5+*6` oder fehlendem Doppelpunkt in Verzweigungen auf (Abb. 4.1).

```
Python 3.5.2 Shell
File  Edit  Shell  Debug  Options  Window  Help
>>>
>>> # Beispiel 1
>>> ~ # nicht definiertes Zeichen
SyntaxError: invalid syntax
>>>
>>> # Beispiel 2
>>> 5 + * 6 # fehlendes Argument zwischen Operatoren
SyntaxError: invalid syntax
>>>
>>> # Beispiel 3
>>> if (3 > 1) # fehlender Doppelpunkt
SyntaxError: invalid syntax
>>>
>>> # Beispiel 4
>>> 3 * (5 + 1)) # Klammer zu viel
SyntaxError: invalid syntax
>>>
>>> |
                                                   Ln: 97  Col: 4
```

Abbildung 4.1.: Beispiele für Syntaxfehler

In der Python-Shell zeigt der Interpreter den Fehler mit einer Meldung und einer roten Markierung im Code, an der Stelle an der der Syntaxfehler aufgetreten ist, an (Abb. 4.1). Schreibt man das Programm im Editor und lässt es mit dem Menüpunkt `Run` oder F5 laufen, erscheint die Fehlermeldung als Popup-Fenster und die Stelle an der der Syntaxfehler aufgetreten ist, ist rot hinterlegt (Abb. 4.2).

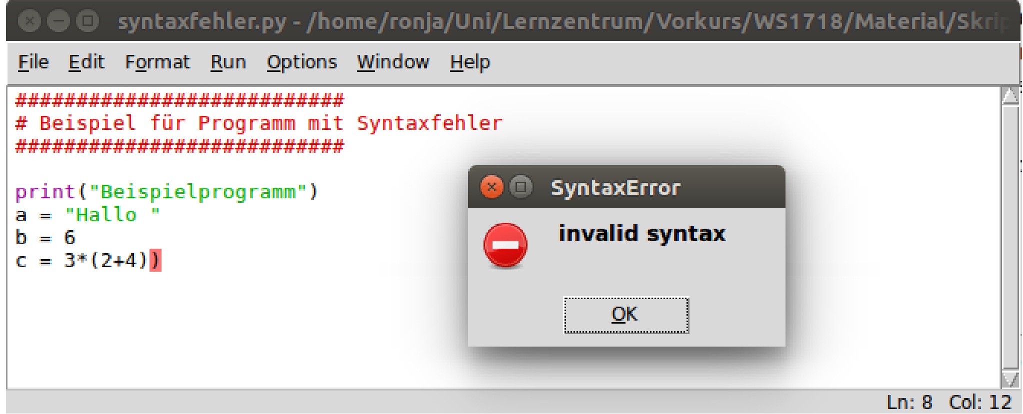


Abbildung 4.2.: Syntaxfehleranzeige im Editor

Syntaxfehler werden vor dem Programmstart vom Interpreter erkannt und angezeigt. Ein Programm mit Syntaxfehlern startet nicht.

Die meisten Syntaxfehler werden durch Schreibfehler verursacht und sind schnell behoben. Da wir Menschen aber einen Hang dazu haben, Dinge nicht so zu sehen wie sie sind, sondern wie wir denken wie sie sein sollten, kann selbst das Korrigieren dieser Fehler langwierig sein.

4.1.1. Typische Syntaxfehler in Python

Falsch gesetzte Anführungszeichen oder Klammern

Häufig passiert es, dass man vergessen hat schließende Anführungszeichen oder Klammern zu setzen. Die Fehlermeldung lautet `SyntaxError: EOL while scanning string`

literal(Abb. 4.3). Das bedeutet Python hat das Ende einer Zeile erreicht, aber keine abschließenden Anführungszeichen gefunden.

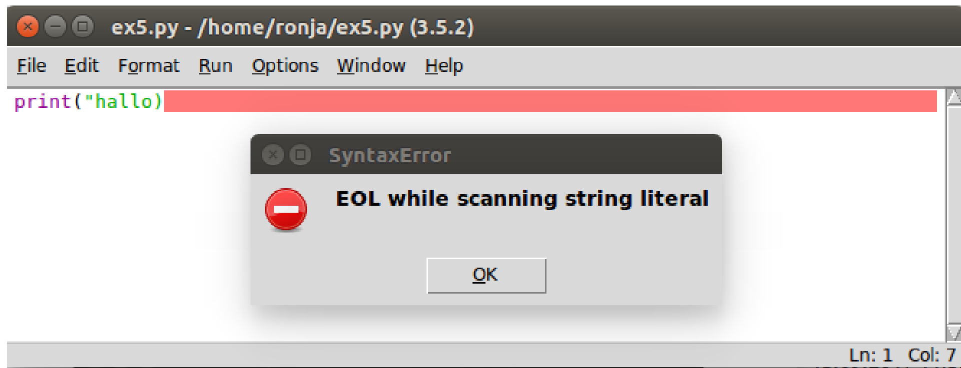


Abbildung 4.3.: Syntaxfehler schließende Anführungszeichen fehlen

Die beste Hilfe zur Vermeidung dieses Fehlers ist das Syntax-Highlighting des IDLE-Editors. Strings werden grün dargestellt. Wenn man Glück hat, fällt einem direkt auf, dass die schließende Klammer der `print`-Anweisung in obigem Beispiel grün eingefärbt ist.

Die Fehlermeldung `SyntaxError:unexpected EOF while parsing` deutet darauf hin, dass Python das Ende einer Zeile erreicht hat, aber noch Eingaben erwartet. Meist deutet das auf eine fehlende schließende Klammer hin (Abb. 4.4).

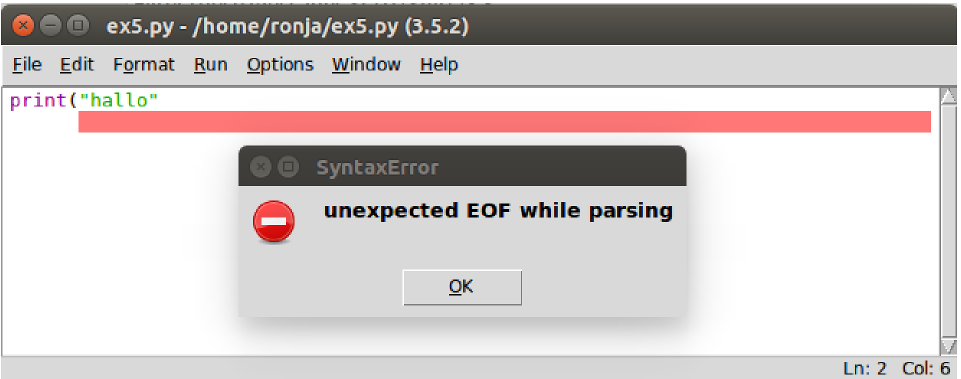


Abbildung 4.4.: Syntaxfehler schließende Klammer fehlt

Auch hier hilft das Syntax-Highlighting des IDLE-Editors, denn immer, wenn man eine schließende Klammer eintippt, wird der Bereich innerhalb der Klammern, die man gerade geschlossen hat, grau eingefärbt.

Achtung! Fehlen die Anführungszeichen komplett, so kann das zu verwirrenden Fehlermeldungen führen (Abb. 4.5).

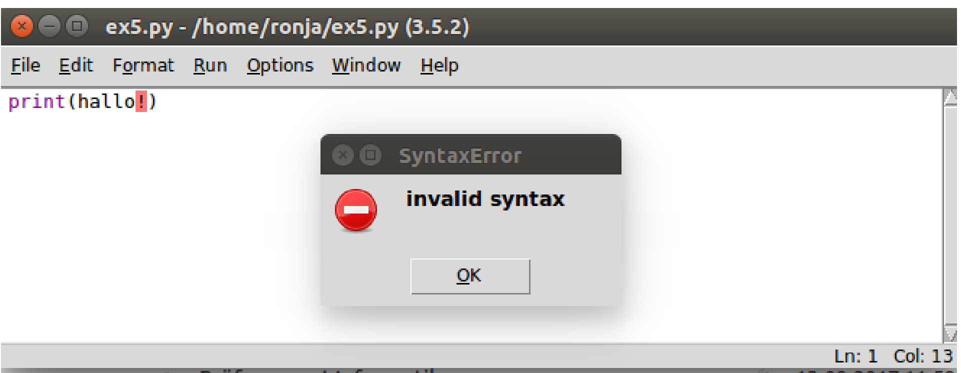


Abbildung 4.5.: Syntaxfehler fehlende Anführungszeichen

Hier hat Python ein Problem mit dem Ausrufezeichen(`!`), da das Ausrufezeichen in Python als Symbol nicht definiert ist. Innerhalb eines Strings kann das Ausrufezeichen sehr wohl verwendet werden, jedoch nicht in Namen oder Bezeichnern.

Schlüsselwörter/reservierte Bezeichner

Folgende Code-Zeile sollte eigentliche keine Probleme machen, oder?

```
1 class="Vorsemesterkurs Informatik"
```

Listing 4.1: Python Zuweisung

Versucht man diese Zeile aber ausführen zu lassen, so zeigt Python einen Syntaxfehler an (Abb. 4.6).

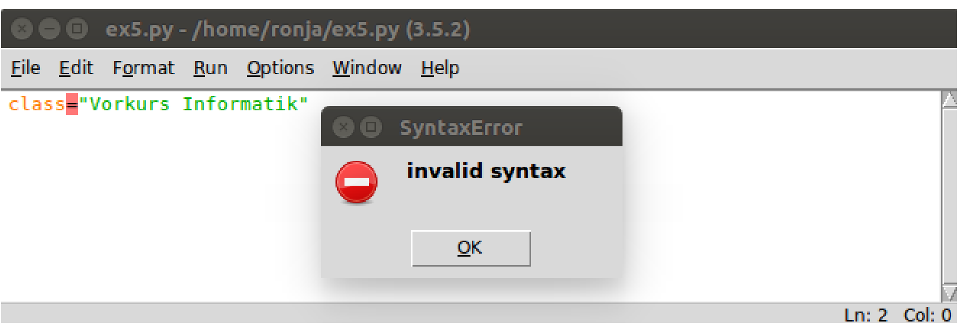


Abbildung 4.6.: Fehlerhafte Python Zuweisung

Schlüsselwort

Hier ist das Problem, dass `class` in Python ein sog. *Schlüsselwort* oder auch *reservierter Bezeichner* ist. Hätten wir `course` oder `Veranstaltung` geschrieben, wäre kein Fehler aufgetreten.

Auch hier kann Syntax-Highlighting helfen den Fehler zu vermeiden, denn Schlüsselwörter werden gelb angezeigt, während Variablennamen schwarz angezeigt werden. Es hätte uns also bereits beim Eintippen auffallen können, dass etwas nicht stimmt. Eine Liste aller Schlüsselwörter kann man sich übrigens mit dem Befehl `import keyword` und anschließendem `print(keyword.kwlist)` in der Python-Shell anzeigen lassen (Abb. 4.7).

```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
>>>
>>> import keyword
>>> print(keyword.kwlist)
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
>>> |
Ln: 73 Col: 4
```

Abbildung 4.7.: Anzeigen aller Python-Schlüsselwörter

Nicht erlaubte Bezeichner

Auch die Verwendung eines nicht erlaubten Bezeichners, wird als Syntaxfehler angezeigt (Abb. 4.8).

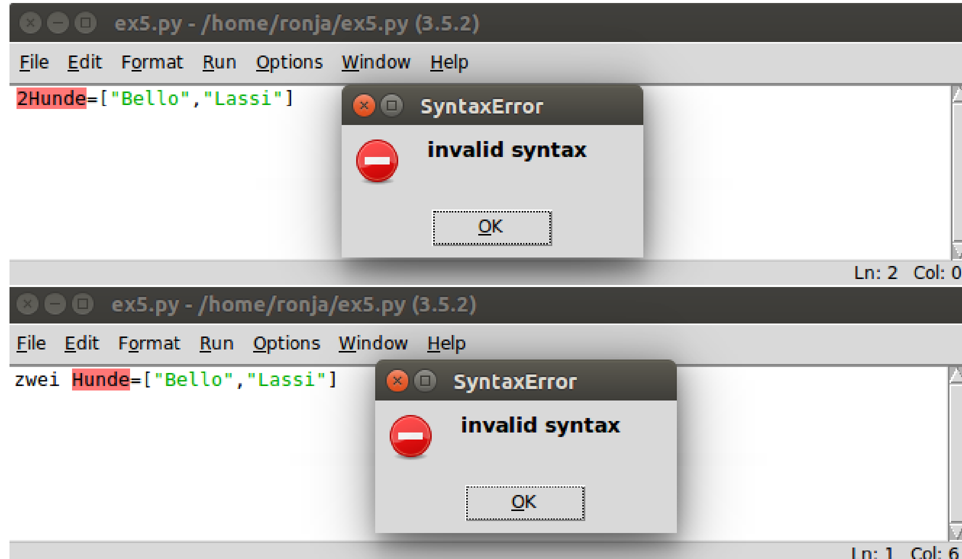


Abbildung 4.8.: Syntaxfehler bei Verwendung nicht erlaubter Bezeichner

Zur Erinnerung: Variablennamen in Python dürfen nur mit einem Unterstrich(_) oder Buchstaben anfangen und keine Leerzeichen enthalten (vergl. 2.3).

Einrückung

Python hat strenge Einrückungsregeln, die beachtet werden müssen. Geschieht dies nicht, wird ein Syntaxfehler angezeigt.

Verzweigungen, Schleifen und Funktionen fordern eingerückte Anweisungsblöcke (Abb. 4.9).

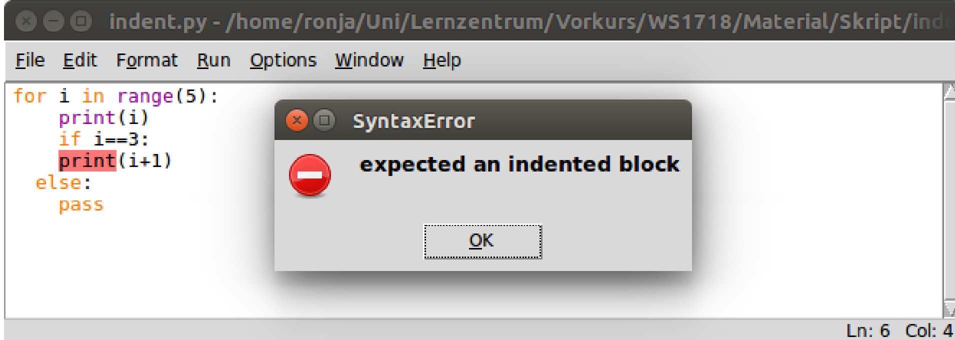


Abbildung 4.9.: Syntaxfehler durch fehlende Einrückung

Laut Konvention sollte die Einrückung 4 Leerzeichen(␣␣␣␣) betragen, muss aber nicht unbedingt. Allerdings muss die Einrückung zusammengehöriger `if` und `else` bzw. `elif` Zeilen gleich sein, sonst kann Python sie nicht zuordnen und beschwert sich (Abb. 4.10).

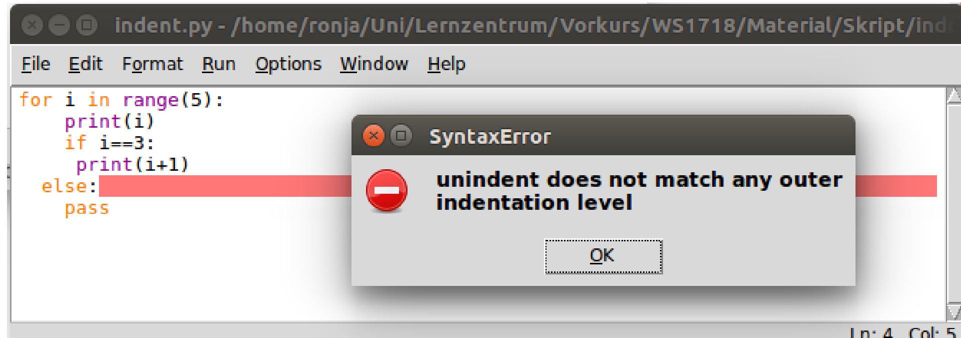


Abbildung 4.10.: Syntaxfehler durch nicht einheitliche Einrückung

Der IDLE-Editor ist in der Regel so eingestellt, dass Tabs ( ) als 4 Leerzeichen(␣␣␣␣) repräsentiert werden. Das ist aber nicht bei jedem Editor der Fall. Verwendet man mal Leerzeichen und mal Tabs zum Einrücken, kann es passieren, dass Python einen `IndentationError` anzeigt, obwohl es im Editor so aussieht, als sei alles einheitlich eingerückt (Abb. 4.11).

Abbildung 4.11.: Einrückung mit Tab und Leerzeichen

Um Fehler dieser Art zu finden und zu korrigieren hilft es, wenn man den Editor so einstellt, dass Tabs und Leerzeichen angezeit werden (Abb.: 4.12)

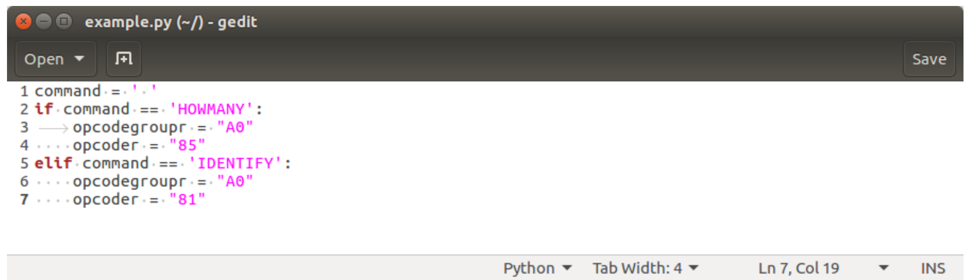


Abbildung 4.12.: Einrückung mit Tab und Leerzeichen, sichtbar

Um solche Fehler zu vermeiden, stellt man im Editor seiner Wahl am besten direkt ein, dass Tabs durch 4 Leerzeichen ersetzt werden.

4.2. Semantikfehler

Semantikfehler

Die zweite Art von Fehlern sind *semantische Fehler*. Python versteht zwar, was es tun soll, bekommt aber bei der Ausführung Probleme. Ein Beispiel in deutscher Sprache wäre der Satz *“Bitte lauf Oma Kuchen.”*. Dies ist ein grammatikalisch völlig korrekter Satz, die Bedeutung (*Semantik*) des Satzes macht aber Probleme. Wie soll man einen Kuchen laufen? Was soll das heißen?

Beispiele für Semantikfehler in Python sind die Verwendung eines vorher nicht definierten Bezeichners (`NameError`), die Anwendung eines Operators auf eine Variable, für deren Typ der Operator nicht definiert ist (`TypeError`) oder der Versuch auf nicht vorhandene Listenelemente zuzugreifen (`ValueError`) bzw auf Speicherbereiche, die nicht zur Liste gehören (`IndexError`) (Abb. 4.13).

```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
>>>
>>> # Beispiel 1
>>> print(a) # Verwendung eines nicht definierten Bezeichners
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    print(a) # Verwendung eines nicht definierten Bezeichners
NameError: name 'a' is not defined
>>>
>>> # Beispiel 2
>>> b = 'hallo'
>>> b - 6 # Anwendung des Operators '-' auf Variablen des Typs string und integer
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    b - 6 # Anwendung des Operators '-' auf Variablen des Typs string und integer
TypeError: unsupported operand type(s) for -: 'str' and 'int'
>>>
>>> # Beispiel 3
>>> listA = [3,5,6,8]
>>> listA.remove(4)
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    listA.remove(4)
ValueError: list.remove(x): x not in list
>>>
>>> # Beispiel 4
>>> del listA[4]
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    del listA[4]
IndexError: list assignment index out of range
>>>
>>> |
                                                        Ln: 47  Col: 4
```

Abbildung 4.13.: Semantikfehleranzeige im Editor

Der erste Teil der Fehlermeldung gibt an, an welcher Stelle das Problem aufgetreten ist. In diesem Fall wurden alle Befehle direkt in die Python-Shell eingegeben, deshalb erscheint als `File` lediglich der Hinweis `pyshell` und die Nummer der Eingabe. Dann folgt die Anweisung bei der der Fehler aufgetreten ist und ein Hinweis was das Problem ist.

Ist ein Programm syntaktisch korrekt, beginnt der Interpreter das Programm auszuführen. Wenn das Programm Semantikfehler enthält, so wird Python irgendwann auf ein Problem stoßen, stoppen und eine Fehlermeldung ausgeben.

Betrachten wir folgendes Programm:

```python
1 print("Dieses Programm berechnet einen Quotienten")
2 a = 1
3 b = 0
4 result = a/b
5 print(result)
```

Listing 4.2: Pythonprogramm mit Semantikfehler

Startet man das Programm, so erscheint in der Python-Shell folgende Ausgabe (Abb. 4.14).

```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
Dieses Programm berechnet einen Quotienten
Traceback (most recent call last):
  File "/home/ronja/Uni/Lernzentrum/Vorkurs/WS1718/Material/Skript/calculate.py"
, line 4, in <module>
    result = a/b
ZeroDivisionError: division by zero
>>> 
Ln: 81 Col: 4
```

Abbildung 4.14.: Fehlermeldung in der PythonShell

Der erste Teil der Fehlermeldung gibt an, an welcher Stelle das Problem aufgetreten ist. In diesem Fall ist der Fehler in der 4. Zeile der Datei `calculate.py` aufgetreten, und zwar bei der Anweisung `result = a/b`. Zusätzlich enthält die Fehlermeldung noch einen Hinweis auf die Art des Fehlers. Es handelt sich um einen `ZeroDivisionError`, d.h. es wurde versucht durch `0` zu teilen. Tatsächlich ist der Wert der Variablen `b`, `0` (siehe Listing 4.2).

Häufig sind Semantikfehler auf einen Fehler im Programmdesign zurückzuführen. Die Fehlermeldung ist also ein Hinweis darauf, wo man anfangen sollte zu suchen. Der eigentliche Fehler liegt aber häufig irgendwo vor dem tatsächlich auftretenden Fehler, wie das nächste Beispiel zeigt.

Beispiel 4.1.
Betrachten wir folgendes Programm zur Berechnung des Alters des Benutzers:

```
1  ##############################
2  # Beispiel für Programm mit Semantikfehler
3  ##############################
4
5  # Dieses Programm berechnet das Alter des Benutzers,
6  #  nachdem dieser sein Geburtsjahr eingegeben hat.
7
8  heute = 2017
9
10 geboren = input("In welchem Jahr sind Sie geboren? ")
11
12 alter = heute - geboren
13
14 print("Sie sind oder werden in diesem Jahr", alter, "Jahre alt.")
```

Listing 4.3: Pythonprogramm zur Berechnung des Alters

Die Ausgabe des Programms in der Python-Shell ist in Abb. 4.15 zu sehen.

```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
 RESTART: /home/ronja/Uni/Lernzentrum/Vorkurs/WS1718/Material/Skript/alter.py
In welchem Jahr sind Sie geboren? 1985
Traceback (most recent call last):
  File "/home/ronja/Uni/Lernzentrum/Vorkurs/WS1718/Material/Skript/alter.py", li
ne 12, in <module>
    alter = heute - geboren
TypeError: unsupported operand type(s) for -: 'int' and 'str'
>>>
Ln: 42 Col: 4
```

Abbildung 4.15.: Fehlermeldung in der PythonShell

In diesem Fall ist ein Typfehler in Zeile 12 der Datei `alter.py` aufgetreten, und zwar bei der Anweisung `alter = heute - geboren`. (Abb. 4.14). Es wurde versucht den `-`Operator auf eine Variable des Typs `integer` und eine Variable des Typs `string` anzuwenden.

Schauen wir uns also den Code an (Listing 4.3). Die Variable `heute` enthält die Zahl 2017, `geboren` enthält die Jahreszahleingabe des Benutzers (vergl. Listing 4.3). Die ist allerdings vom Typ `string`, denn die `input`-Funktion gibt alle Eingaben als `string` zurück[1]. Der eigentliche Fehler ist also früher gemacht worden, denn die Benutzereingabe hätte mit `int()` in einen `integer`-Wert umgewandelt werden sollen.

```python
# Dieses Programm berechnet das Alter des Benutzers,
#  nachdem dieser sein Geburtsjahr eingegeben hat.

heute = 2017

geboren = int(input("In welchem Jahr sind Sie geboren? "))

alter = heute - geboren

print("Sie sind oder werden in diesem Jahr", alter, "Jahre alt.")
```

Listing 4.4: Pythonprogramm zur Berechnung des Alters

```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
>>>
 RESTART: /home/ronja/Uni/Lernzentrum/Vorkurs/WS1718/Material/Skript/alter.py
In welchem Jahr sind Sie geboren? 1985
Sie sind oder werden in diesem Jahr 32 Jahre alt.
>>>
>>>
>>> |
                                                    Ln: 87  Col: 4
```

Abbildung 4.16.: Ausgabe des Programms in Listing 4.4

4.2.1. Typische Semantikfehler

Viele Fehler führen zu Fehlermeldungen. Wir geben im folgenden einige Beispiele und Hinweise wie man bei der Fehlersuche vorgehen kann.

TypeError

Typfehler treten auf, wenn man versucht Operatoren oder Funktionen auf Variablen eines Typs anzuwenden für den sie (die Operatoren und Funktionen) nicht definiert sind (siehe Bsp 4.1).

Häufige Ursachen

- vergessen die Benutzereingabe in eine Zahl umzuwandeln
- bei der Parameterübergabe an eine Funktion mit mehreren Parametern die Reihenfolge nicht beachtet

Fehlersuche: Die built-in-Funktion `type()` gibt den Typ der Variablen aus. Eine Möglichkeit Typfehler zu finden ist, `print()`-Anweisungen in den Programmcode einzufügen und sich Wert und Typ von Variablen im Programmverlauf ausgeben zu lassen. In Kapitel 4.4 wird auf diese Methode ausführlicher eingegangen.

Tipps
Wenn man eine Funktion mit einer langen (längeren) Parameterliste hat, ist es sinnvoll, die Parameter bei der Übergabe explizit zu benennen. Damit ist klar festgelegt, welcher Wert für welchen

[1] https://docs.python.org/3/library/functions.html#input

Parameter steht. Gibt man die Namen nicht an, so ordnet Python die übergebenen Parameter der Reihenfolge nach zu.

Beispiel 4.2.
Betrachten wir folgendes Programm:

```python

def func(artikel, lager, verkauft):
    bestand = lager - verkauft
    print(artikel, bestand)
    return

# Funktionsaufruf 1:
func("Nägel", 7000, 2975)

# Funktionsaufruf 2:
func(lager=7000, verkauft=2975, artikel="Nägel")

#Funktionsaufruf 3:
func(7000, 2975, "Nägel")
```

Listing 4.5: Parameterübergabe

Funktionsaufruf 1 und 2 liefern dasselbe Ergebnis. Funktionsaufruf 3 führt aber zu einer Fehlermeldung, denn Python ordnet den `string`-Wert `''Nägel''` der Funktionsvariablen `verkauft` zu. Dementsprechend kommt es zu einem Typfehler, wenn in Zeile 3 versucht wird die `string`-Variable zu subtrahieren (Abb. 4.17).

```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
=============================== RESTART: Shell ===============================
>>>
>>>
 RESTART: /home/ronja/Uni/Lernzentrum/Vorkurs/WS1718/Material/Skript/para.py
Nägel 4025
Nägel 4025
Traceback (most recent call last):
  File "/home/ronja/Uni/Lernzentrum/Vorkurs/WS1718/Material/Skript/para.py", lin
e 14, in <module>
    func(7000, 2975, "Nägel")
  File "/home/ronja/Uni/Lernzentrum/Vorkurs/WS1718/Material/Skript/para.py", lin
e 3, in func
    bestand = lager - verkauft
TypeError: unsupported operand type(s) for -: 'int' and 'str'
>>>
Ln: 142 Col: 4
```

Abbildung 4.17.: Ausgabe des Programms in Listing 4.5

NameError

Ein NameError bedeutet in den meisten Fällen, dass man eine Variable verwendet hat, bevor sie einen Wert hat.

Häufige Ursachen

- die allerhäufigste Ursache ist ein Tippfehler
- man erinnert sich nicht korrekt an den Namen einer Variablen oder Funktion
- man hat vergessen ein Modul zu importieren

- ein Variablenname wird außerhalb seines Gültigkeitsbereichs benutzt
- im `print()`-Befehl die Anführungszeichen vergessen
- eine Funktion wird aufgerufen bevor sie definiert wurde

Fehlersuche: Wenn man nicht besonders gut darin ist Tippfehler zu finden, kann man die Suchfunktion des Editors nutzen. In den meisten Fällen passiert eins von zwei Dingen:

1. Das Wort nach dem man sucht, taucht nur ein einziges Mal auf im Code auf, meist auf der rechten Seite einer Zuweisung oder als Parameter in einem Funktionsaufruf. Falls der Name so ist wie er sein soll, dann ist der Tippfehler irgendwo im Code davor, und zwar auf der linken Seite einer Zuweisung. Es sei denn, man hat überhaupt vergessen der Variablen einen Wert zuzuweisen.
2. Man sieht die Zeile in der der NameError aufgetaucht ist und in der man den gesuchten String vermutet, aber die Suche hat nichts gefunden. Dann ist dort genau der Tippfehler.

Tipp
Es hilft, Variablen und Funktionen nach einem einheitlichen Schema zu benennen, denn Python unterscheidet zwischen Groß- und Kleinschreibung und `myVar` ist nicht das gleiche wie `MyVar` oder `myvar`. Ferner ist die Autovervollständigung von IDLE hilfreich. Bei der Verwendung einer Variablen oder Funktion von der man meint, dass sie bereits angelegt ist, sollte sich der Name in der Liste der Vorschläge befinden. Falls nicht, kann man sich direkt auf Ursachenforschung begeben, falls ja, verwendet man den Vorschlag und kann in dem Teil schonmal keinen Tippfehler mehr machen.

ValueError

Ein ValueError tritt auf, wenn ein Wert an eine Funktion übergeben wird und die Funktion Werte innerhalb bestimmter Parameter erwartet, der übergebene Wert aber außerhalb dieser Parameter liegt.

Beispiel 4.3.
Betrachten wir folgenden Code:

```

artikel = input("Welcher Artikel wird bearbeitet? ")
lager = int(input("Wie viele sind auf Lager? "))
verkauft = int(input("Wie viele wurden verkauft? "))

bestand = lager - verkauft

print(artikel, bestand)
```

Listing 4.6: Interaktives Programm zur Bestandsbestimmung

Führt man ihn aus und gibt bei der Frage wie viel verkauft wurde nichts ein (drückt ↵), da kein Artikel verkauft wurde, erhält man einen ValueError (Abb. 4.18), denn die Funktion `int()` erwartet eine Eingabe die aus Zahlen besteht.

```
Python 3.5.2 Shell
File  Edit  Shell  Debug  Options  Window  Help
 RESTART: /home/ronja/Uni/Lernzentrum/Vorkurs/WS1718/Material/Skript/valueError.
py
Welcher Artikel wird bearbeitet? Töpfe
Wie viele sind auf Lager? 387
Wie viele wurden verkauft?
Traceback (most recent call last):
  File "/home/ronja/Uni/Lernzentrum/Vorkurs/WS1718/Material/Skript/valueError.py
", line 4, in <module>
    verkauft = int(input("Wie viele wurden verkauft? "))
ValueError: invalid literal for int() with base 10: ''
>>>
>>> |
                                                                Ln: 14  Col: 4
```

Abbildung 4.18.: Ausgabe des Programms in Listing 4.6

Zwar ist der Fehler hier durch einen Eingabefehler des Nutzers entstanden, aber nicht alle ValueErrors werden durch falsche Nutzereingaben hervorgerufen. Es bleibt zu betonen, dass man sich bewußt machen sollte, welche Einschränkungen gelten und ob diese von allen möglichen Übergabeparametern eingehalten werden.

4.3. Logische Fehler

Die dritte Art der Fehler ist am schwierigsten zu entdecken, denn das Programm läuft ohne Fehlermeldungen durch. Trotzdem arbeitet es nicht so, wie wir uns das vorgestellt haben.

Der Satz *"Trink einen Schnaps, das wärmt."* ist ein Beispiel für einen logischen Fehler in deutscher Sprache. Alkohol erweitert die Blutgefäße der Haut und führt dadurch zu Wärmeverlust.

Ein Beispiel für einen logischen Fehler in Python wäre folgende Berechnung des Mittelwerts:

```python
1 def mittelwert(a,b):
2     '''Berechnet den Mittelwert der übergebenen Parameter a und b '''
3     mittelwert=a+b/2
4     return(mittelwert)
```

Listing 4.7: Funktion mit logischem Fehler

Die Funktion gibt 5.5 als Mittelwert der Werte 3 und 5 zurück (Abb. 4.19). Der tatsächliche Mittelwert von 3 und 5 ist jedoch $\frac{3+5}{2} = \frac{8}{2} = 4$.

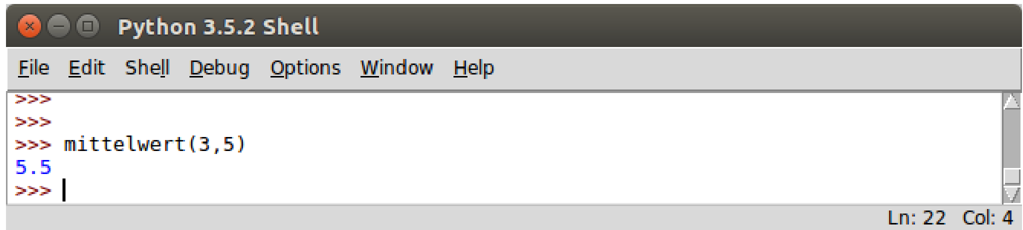


Abbildung 4.19.: Funktionsaufruf `mittelwert(3,5)`

Was ist passiert?
Python beachtet die allgemeinen Rechenregeln und rechnet Punkt- vor Strichrechnung. Somit berechnet die Funktion `mittelwert(a,b)` nicht $\frac{a+b}{2}$ sondern $a + \frac{b}{2}$ (vergl. Listing 4.7). Für $a = 3$ und $b = 5$ ergibt sich $3 + \frac{5}{2} = 3 + 2.5 = 5.5$. Die Funktion rechnet also einwandfrei, ohne Fehlermeldung, allerdings nicht das, was eigentlich gemeint war. Hätten wir den Mittelwert zweier

Zahlen nicht selbst schnell im Kopf ausrechnen können, wer weiß, ob uns dieser Fehler überhaupt aufgefallen wäre.

Logische Fehler lassen sich schwer kategorisieren, da die Fehlerquellen vielfältig sind. Manchmal ist es “nur” das “Vergessen” von Klammern, häufig gibt es aber auch grundlegende Fehler im Programmdesign oder im Algorithmus selbst. Im Folgenden sind ein paar Fehler die vor allem von Programmieranfängerinnen und -anfängern häufig gemacht werden.

4.3.1. Häufige Anfängerfehler

Falsche Verwendung logischer Operatoren : Logische Operatoren können schwierig sein. `and` muss verwendet werden, wenn beide Bedingungen erfüllt sein müssen, `or`, wenn mindestens eine der Bedingungen erfüllt sein muss. Bei komplexeren Ausdrücken, hilft es auch Klammern zu setzen.

Schleifendurchlauf endet vor dem letzten Element : Die `range()`-Funktion stoppt *vor* dem angegebenen Endwert. `range(1:11)` enthält lediglich Zahlen von 1 bis 10!

Schleifendurchlauf lässt erstes Element aus : Listen beginnen mit dem Index `0`. Um auf das erste Listenelement zuzugreifen, muss die Laufvariable bei 0 anfangen.

Wenn man Glück hat, dann führt ein logischer Fehler zu einem Semantikfehler und man findet den ursprünglichen Fehler bei dem Versuch den Semantikfehler zu beseitigen. Allerdings kann man sich darauf nicht verlassen, daher sollte man sein Programm testen.

4.4. Debugging mit `print()`

Es gibt viele Hilfsmittel zum Debuggen. Das einfachste ist die `print()`-Anweisung. Mit `print()` können Variablenwerte und Nachrichten in der Python-Shell ausgegeben werden. Informative Nachrichten im Programmablauf können uns helfen nachzuvollziehen, was genau passiert und an welcher Stelle im Code Python nicht das tut, was wir uns gedacht hatten.

Tut also ein Programm nicht das, was es eigentlich tun soll, muss man sich den Code anschauen und überlegen, an welcher Stelle der Fehler sein kann. Eine Fehlermeldung gibt meist einen guten Hinweis darauf, wo man anfangen sollte zu suchen. Eine unerwartetes Verhalten sollte dem Programmierer ebenfalls eine Idee geben an welcher Stelle im Code der Fehler stecken könnte. Hier hilft auch eine gute Dokumentation und Kommentare im Code um später leichter rekonstruieren zu können, was bestimmte Teile eigentlich tun sollen. Bei umfangreicheren Programmen ist dies unerlässlich.

Zunächst kann man `print()`-Anweisungen in den Code einfügen und so den Bereich in dem der Fehler steckt, einengen. Es ist hilfreich Zeilennummern und ggf Dateinamen mit in die Ausgabe zu schreiben, um festzuhalten wo man sich im Code befindet. Auch ist es ratsam, die eingefügten `print()`-Anweisungen zu kennzeichnen, damit man sie von regulären `print()`-Anweisungen, die zur Programmfunktionalität notwendig sind, unterscheiden kann und sie später schneller wieder entfernen kann.

Beispiel 4.4.
Betrachten wir folgendes Beispiel:

```
1 def func():
2     return 6
3
4 x = func()
5 for i in range(7):
```

```
    x += i


if x > 23:
    print("Welcome!")

else:
    print("Good bye!")
```

Listing 4.8: Beispielprogramm zur Fehlersuche

Dieses Beispiel ist sehr einfach und konstruiert, aber nehmen wir an `func()` sei eine sehr komplizerte Funktion und wir könnten uns nicht sicher sein, welchen Wert sie zurück gibt und nehmen wir an, dass wir nicht die Ausgabe `Welcome!` erwartet haben, sondern `Good bye!`. Wenn `Welcome!` und `Good bye!` noch an anderen Stellen im Code ausgegeben würde, könnten wir an alle Stellen im Code gehen und vor jeder Ausgabe eine `print()`-Anweisung einfügen (Listing 4.9)

```
def func():
    return 6

x = func()
for i in range(7):
    x += i

if x > 23:
    print("Debugging: File1.py, line 10")
    print("Welcome!")

else:
    print("Debugging: File1.py, line 14")
    print("Good bye!")
```

Listing 4.9: Programm mit `print()`-Anweisungen zur Fehlersuche

Lassen wir das Progamm jetzt laufen, sieht die Ausgabe wie folgt aus:

```
Python 3.5.2 Shell
File  Edit  Shell  Debug  Options  Window  Help
Debugging: File1.py, line 10
Welcome!
>>> |
                                        Ln: 21  Col: 4
```

Abbildung 4.20.: Ausgabe des Programms in Listing 4.9

Jetzt haben wir Gewissheit, dass das Fehlverhalten tatsächlich an dieser Stelle auftritt. Nun arbeiten wir uns von dort aus zurück. Direkt vor der Ausgabe wird eine Bedingung abgefragt. Die Ausgabe ist abhängig davon, welchen Wert `x` hat. Also würden wir als nächstes eine `print()`-Anweisung direkt hinter den Teil im Code setzen in dem `x` verändert wird (Listing 4.10).

```
def func():
    return 6

x = func()
for i in range(7):
    x += i
print("Debugging: x =", x)
if x > 23:
    print("Debugging: File1.py, line 10")
    print("Welcome!")
```

```
11
12 else:
13     print("Debugging: File1.py, line 14")
14     print("Good bye!")
```

Listing 4.10: Programm mit `print()`-Anweisungen zur Fehlersuche

Die Ausgabe ändert sich entsprechend zu:

```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
Debugging: x = 27
Debugging: File1.py, line 10
Welcome!
>>>
                                        Ln: 31 Col: 4
```

Abbildung 4.21.: Ausgabe des Programms in Listing 4.10

`x` hat also den Wert 27 und es wundert uns nicht mehr, dass `Welcome!` ausgegeben wird. Die Frage ist, warum `x` den Wert 27 hat. Wir betrachten den Code und sehen, dass Zeilen 5 und 6 21 zu dem vorherigen Wert von `x` addieren. Wir könnten nun eine `print()`-Anweisung in Zeile 4 einfügen und stellen fest, dass die Funktion `func()` `x` einen Wert von 6 zuweist. Davon ausgehend dass `func()` eine komplizierte Funktion ist, könnten wir nun `print()`-Anweisungen innerhalb der Funktion einfügen um nachzuvollziehen, was genau passiert und warum die Funktion den Wert `6` zurückgibt.

Allgemeines Vorgehen bei der Fehlersuche mit `print()`

1. Finden des ungefähren Bereichs im Code, der das Fehlverhalten zu verursachen scheint, indem man Zeilennummer (und Dateinamen) in Teilen des Programms in denen man den Fehler vermutet, ausgeben lässt.
2. Wenn man den Bereich gefunden hat, den Code lesen um eine Vorstellung davon zu bekommen, welche Variablen für den Programmfluss und das Programmverhalten wichtig sind. Die Werte dieser Variablen ausgeben lassen.
3. im Code rückwärts arbeiten um festzustellen, woher die Werte der Variablen kommen.
4. Schritte 1-3 wiederholen, bis man die Ursache des Fehlverhaltens gefunden hat.

5. Objektorientierte Programmierung

Objektorientierte Programmierung ist ein Programmierstil. Es ist eine Art und Weise Software (Programme) zu entwerfen, die sich an die Prinzipien der Objektorientierung hält. Objektorientierung bedeutet, dass ein System durch das Zusammenspiel kooperierender Objekte beschrieben wird. Dabei ist der Objektbegriff unscharf gefasst. Es muss keine Einheit sein, die man sehen und anfassen kann. Alles dem bestimmte Eigenschaften und Verhaltensweisen (*Methoden*) zugeordnet werden können und das in der Lage ist, mit anderen Objekten zu interagieren, ist ein Objekt.

Objektorientierung liegt in der Natur des Menschen. Schon früh erkennen Kinder, dass Roller, Dreirad, Fahrrad, Auto oder Lastwagen alle etwas gemeinsam haben. Es sind alles Fortbewegungsmittel. Täglich ordnen wir Gegenstände in Gruppen. Äpfel, Birnen oder Bananen sind Obst, Tomaten, Gurken oder Möhren sind Gemüse. Programmiert man ein Programm zur Lagerverwaltung, so arbeitet man genauso mit Objekten. Waren, Regale, Transportmittel, und viele mehr. Das Leben ist voller Objekte, und Personen in der Softwareentwicklung haben erkannt, dass komplexe Probleme einfacher oder nachhaltiger zu lösen sind, wenn für objektorientierte Probleme, objektorientierte Lösungen gefunden werden können.

Bei der objektorientierten Programmierung können Objekte direkt im Programm abgebildet werden. Dabei beschränkt man sich auf die Eigenschaften und Verhaltensweisen des Objekts, die für die Anwendung des Programms relevant sind. Eigenschaften eines Objekts werden *Attribute* genannt. Typische Eigenschaften des Objekts Mensch wären z.B. Name, Geburtsdatum, Größe, Gewicht oder Haarfarbe, während die Eigenschaften eines Autos Geschwindigkeit, Kraftstoffstand oder Farbe sein könnten. Das Verhalten eines Objekts wird durch seine *Methoden* definiert. Ein Mensch nimmt zu oder wird älter, ein Auto kann beschleunigen oder bremsen. Objekte haben oft Beziehungen untereinder, was ebenfalls als Methode anzusehen ist. Ein Mensch kann z.B. ein Auto fahren. Somit hat der Mensch eine Methode `fahren(Auto)` die auf ein Auto zugreift.

Attribut

Methode

Um nicht für jedes einzelne Objekt immer wieder seine Eigenschaften und sein Verhalten festzulegen, werden gleichartige Objekte zu Gruppen zusammengefasst. Diese Gruppen nennt man *Klassen*. Klassen sind der Bauplan für Objekte. In ihnen werden alle Attribute und Methoden, welche die Objekte der Klasse ausmachen, definiert. Die konkreten Objekte werden über die Klasse erzeugt. Häufig wird ein konkretes Objekt auch als *Instanz* (von engl.: *instance*) der Klasse bezeichnet (Abb. 5.1).

Klasse

Instanz

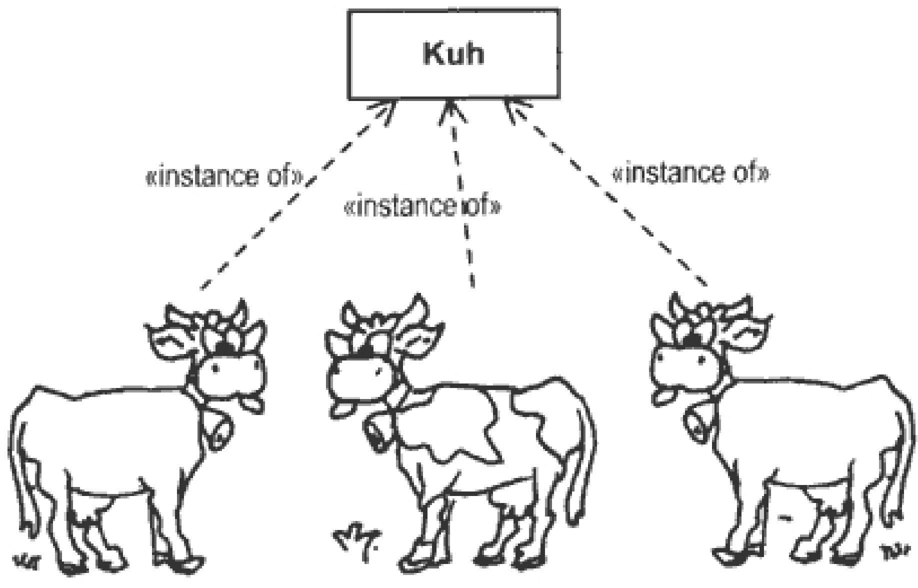


Abbildung 5.1.: Instanzen einer Klasse, v.l.n.r.: Elsa Euter, Vera Vollmilch und Anja v.d. Alm
Quelle: www.hki.uni-koeln.de

Elsa Euter, Vera Vollmilch und Anja v.d. Alm beispielsweise sind Objekte, oder Instanzen der Klasse Kuh. Sie haben natürlich unterschiedliche Attributwerte. Elsa hat ihr eigenes Geburtsdatum, und gibt im Durchschnitt mehr Milch als ihre Kollegin Anja v.d. Alm und hat einen eigenen Namen. Die Werte seiner Attribute definieren den *Zustand* eines Objekts.

Zustand

Klassen werden in einer hierarchischen Beziehung aufgebaut. Z.B. könnte es die Klasse “Nutztier” geben. Diese Klasse hat die Kindklassen “Kuh” und “Schwein”. Die Kindklassen erben die Eigenschaften und das Verhalten der allgemeineren Elternklasse Nutztier und haben zusätzlich noch andere Eigenschaften und Methoden, die in der allgemeineren Elternklasse nicht enthalten sind. So benötigt die Klasse “Kuh” eine Methode `gemolken_werden()`. Zwischen Kind- und Elternklasse besteht eine “ist ein” Beziehung. Eine Kuh *ist ein* Nutztier.

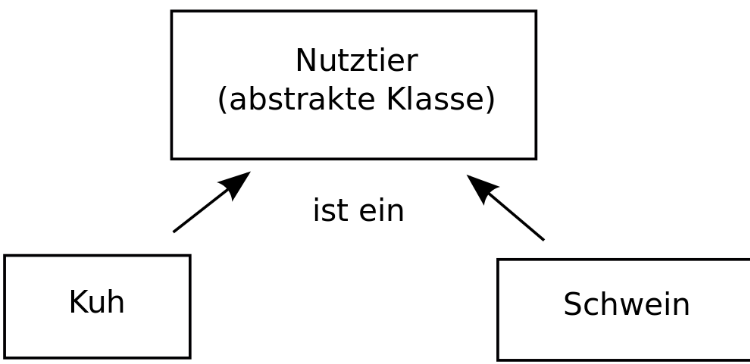


Abbildung 5.2.: Eltern- und Kindklassen

Die Klasse “Nutztier” könnte in diesem Fall eine *abstrakte Klasse* sein. Abstrakte Klassen sind ein gängiges Konzept in der objektorientierten Programmierung. Sie enthalten selbst nur leere Methoden. Dementsprechend kann keine Instanz von ihnen erzeugt werden. Sie dienen dazu ähnliche Klassen unter einem Oberbegriff zusammenzufassen und gemeinsame Methoden- und Attributnamen zu definieren. Sie müssen abgeleitet werden, um sinnvoll verwendet zu werden. Dann garantieren sie das Vorhandensein der in ihnen definierten Attribute und Methoden in allen ihren Kindklassen.

abstrakte Klasse

Vorteile der objektorientierten Programmierung sind:

Abstraktion: Jedes Objekt im System kann als ein abstraktes Modell eines Akteurs betrachtet werden. Bei der Systementwicklung konzentriert sich der Entwickler zunächst darauf, welche Objekte benötigt werden und welche Objekteigenschaften und -fähigkeiten für die Softwarelösung wichtig sind, ohne direkt festzulegen, wie dieses *implementiert* (programmiert) werden müssen. Das verhindert eine frühe Festlegung auf Datenstrukturen etc. bevor das Problem vollständig verstanden wurde.

Abstraktion

Kapselung: auch *information hiding* genannt. Objekte können auf Attribute anderer Objekte nur über vordefinierte Methoden zugreifen. Nach außen ist somit lediglich sichtbar *was* implementiert ist, aber nicht *wie*. Dadurch kann die Implementierung eines Objektes geändert werden, ohne die Zusammenarbeit mit anderen Objekten zu beeinträchtigen, solange sich die *Schnittstelle* (nach außen sichtbare Methoden) nicht ändert.

Kapselung

Schnittstelle

Vererbung: durch die Vererbungsbeziehung zwischen Klassen ist es möglich, dass ähnliche Kindklassen eine gemeinsame Struktur nutzen. Dadurch werden Redundanzen vermieden und die konzeptuelle Klarheit wird erhöht.

Wiederverwendbarkeit: Abstraktion, Kapselung und Vererbung erhöhen die Wiederverwendbarkeit objektorientierte Software enorm. Nicht nur können entworfene Klassen und Strukturen in anderen Programmen wiederverwendet werden, auch die Weiterentwicklung von Software wird durch die Modularität deutlich erleichtert. Da im System lediglich die Interaktion von Objekten wichtig ist, ist objektorientierte Software sehr viel übersichtlicher und damit leichter testbar, stabiler und änderbar.

Beispiel 5.1 (Bankkonto)**.**
Stellen wir uns eine Klasse “Bankkonto” vor. Ein Bankkonto hat einen Kontoinhaber, eine Kontonummer und einen Kontostand. Typische Methoden für ein Bankkonto wären `einzahlen(betrag)` und `auszahlen(betrag)`. Diese Methoden verändern den Wert des Attributs Kontostand. Ein Kontoinhaber hat einen Namen, Anschrift, und Geburtsdatum. Durch das Verhalten “umziehen” ändert sich die Anschrift (Abb. 5.3)

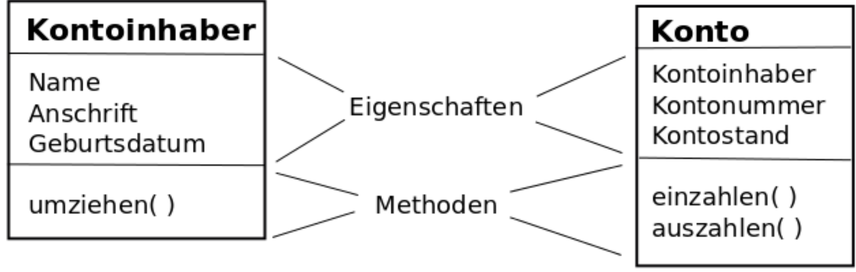


Abbildung 5.3.: Klassendiagramme der Klassen Kontoinhaber und Konto

Die Klasse “Konto” genügt einer realen Bank nicht. Es gibt verschiedene Arten von Konten. Sparkonten, die nicht überzogen werden dürfen und Girokonten, die bis zu einem gewissen Rahmen sehr wohl überzogen werden dürfen. Die Eigenschaften und Methoden unserer Klasse “Konto”, sind aber beiden Kontoarten gemeinsam. Dies sind Attribute und Methoden, die sie von der Klasse “Konto” erben (Abb. 5.4)

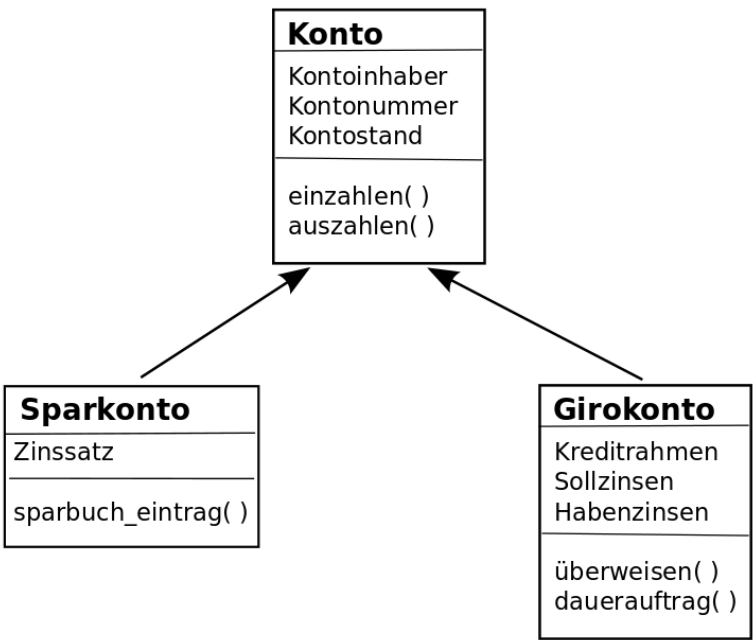


Abbildung 5.4.: Klassenhierarchie Konto

Implementierung

In Python beginnt eine Klasse mit dem Schlüsselwort `class`, gefolgt vom Klassennamen und Doppelpunkt (`:`). Klassennamen beginnen mit einem Großbuchstaben. Unsere Klasse “Konto” könnte wie folgt implementiert werden.

konto.py - /home/ronja/Dropbox/Skript/Code/konto.py

File Edit Format Run Options Windows Help

```
# Implementierung der Klasse Konto mit private Attributen

class Konto:
    'Klasse, die ein generische Bankkonto repräsentiert'
    anzKont = 0  # Klassenvariable die zählt, wie viele Konten erzeugt wurden

    def __init__(self, name, nummer, betrag):  # Konstruktor
        ' initialisiere neues Exemplar der Klasse '
        self.inhaber = name
        self.kontonummer = nummer
        self.saldo = betrag
        Konto.anzKont += 1 # erh\"ohe die Klassenvariable um 1

    def einzahlen(self, betrag): # auf das Konto einzahlen
        'addiere zum Saldo hinzu'
        self.saldo = self.saldo + betrag

    def auszahlen(self, betrag): # vom Konto abheben
        'subtrahiere vom Saldo'
        self.saldo = self.saldo - betrag

    def abfragen(self): # Saldo abfragen
        'gib aktuellen Kontostand zurück'
        return self.saldo

    def __del__(self): # Destruktor
        Konto.anzKont -= 1 # veringere Klassenvariable um 1

```

Ln: 29 Col: 0

Abbildung 5.5.: Implementierung der Klasse Konto

Innerhalb der Klasse werden dann die Attribute und Methoden definiert. Eine Methode unterscheidet sich von einer Funktion darin, dass:

- sie innerhalb einer Klasse definiert ist.
- der erste Parameter immer eine Referenz auf die Instanz der Klasse (`self`) ist.

Da `self` immer der erste Parameter ist, erscheint er nur bei der Definition der Methode. Beim Aufruf wird er nicht angegeben.

Konstruktor

Die Implementierung der Klasse "Konto" erzeugt noch kein Konto an sich. Es wird lediglich definiert, welche Eigenschaften ein Konto hat und was man mit ihm machen kann. Um eine Instanz der Klasse "Konto" anzulegen, muss der *Konstruktor* aufgerufen werden. Der Konstruktor ist eine spezielle Methode. In Python wird der Konstruktor durch die Methode `__init__` aufgerufen. Im Konstruktor werden Initialisierungen bei der Erstellung einer Instanz der Klasse vorgenommen. In unserem Beispiel muss jedes Konto einen Kontoinhaber, eine Kontonummer und einen Kontostand (Saldo) haben. Dementsprechend müssen beim Anlegen eines Kontos (Exemplar der Klasse Konto) Werte für diese Attribute übergeben werden. Diese werden dann den entsprechenden Attributen (`inhaber, kontonummer, saldo`) zugewiesen. Ferner haben wir für unsere Konto-Klasse eine Klassenvariable `anzKont` definiert. Eine Klassenvariable wird von allen Instanzen der Klasse geteilt. Mit dem Befehl `<Klassenname>.<Variablenname>` kann von innerhalb und außerhalb der Klasse auf Klassenvariablen zugegriffen werden. In unserer Klasse soll die Klassenvariable angeben wie viele Konten es aktuell gibt. Dementsprechend muss beim Aufruf des Konstruktors jedesmal der Wert von `Konto.anzKont` erhöht werden. Ferner haben wir die Methoden `einzahlen, auszahlen` und `abfragen` definiert.

Klassenvariable

Destruktor

Es ist auch möglich einen *Destruktor* für eine Klasse zu entwerfen. Mit dem Destruktor kann eine Instanz der Klasse gelöscht werden. In Python lautet der Name der Methode die den Destuktor aufruft `__del__`. In unserem Fall wollen wir, dass die Klassenvariable `Konto.anzKont` um 1 re-

duziert wird, wenn ein Konto gelöscht wird. Destruktoren werden selten benutzt, da man sich normalerweise nicht um das Aufräumen im Speicher kümmern muss.

Hat man den obigen Quelltext unter `konto.py` abgespeichert, so kann man die Klasse mit dem Befehl `from konto import Konto` in den Python-Interpreter oder jedes andere Programm laden und mit der Klasse arbeiten (Abb. 5.6).

```
k1 = Konto("Joe", 176358, 1000) # Konto für Joe mit 1000 € anlegen
k2 = Konto("Jane", 938674, 10000.00) # Konto für Jane mit 10000 € anlegen

k1.einzahlen(50.79) # 50,79 € auf Joes Konto einzahlen
# kontrollieren, ob das Geld angekommen ist
print("Joes Kontostand ist:",k1.abfragen())

k2.auszahlen(6000) # 6000 € von Janes Konto abheben

print("Anzahl der Konten:", Konto.anzKont) # Schauen, wie viele Konten es gibt

del k1 # Joes Konto löschen
print("Anzahl der Konten:", Konto.anzKont) # Schauen, wie viele Konten es gibt

print("Janes Saldo beträgt:",k2.saldo) # Jane's Kontostand
```

Abbildung 5.6.: Verwendung der Klasse Konto

Ausgabe des Programms:

```
>>> ================================ RESTART ================================
>>>
Joes Kontostand ist: 1050.79
Anzahl der Konten: 2
Anzahl der Konten: 1
Janes Saldo beträgt: 4000.0
>>>
```

Abbildung 5.7.: Ausgabe des Programms zur Klasse Konto

Datenkapselung

Im Augenblick sind noch alle Attribute unserer Klasse "Konto" direkt von außen zugänglich. Mit dem Aufruf `k2.saldo` kann man direkt Janes Kontostand abrufen, ohne über die `abfrage`-Methode gehen zu müssen (Abb. 5.7). Das widerspricht aber dem Prinzip der Kapselung, welches in der objektorientierten Programmierung so vorteilhaft ist. Daher gibt es Möglichkeiten den direkten Zugriff zu unterbinden. In Python erfolgt dies über die Anzahl der Unterstriche (_) vor dem Namen der Attribute und Methoden (Tab. 5.1).

Name	Bezeichnung	Bedeutung
`name`	public	Attribute ohne führenden Unterstrich sind sowohl innerhalb einer Klasse, als auch von außen les- und schreibbar.
`_name`	protected	Attribute mit *einem* führenden Unterstrich sind zwar auch von außen les- und schreibbar, aber der Entwickler macht damit klar, dass man diese Attribute und Methoden nicht benutzen sollte.
`__name`	private	Attribute mit *zwei* führenden Unterstrichen sind von außen werder sichtbar, noch nutzbar.

Tabelle 5.1.: Datenkapselung in Python.

Nun verbessern wir unsere Kontoklasse und schützen sie vor ungewolltem Zugriff von außen. Ferner fügen wir `get`- und `put`-Methoden hinzu, um den Zugriff von außen zu regeln (Abb. 5.8).

```
impKonto.py - /home/ronja/Dropbox/Skript/Code/impKonto.py
File Edit Format Run Options Windows Help

# Implementierung der Klasse Konto mit private Attributen

class Konto:
    'Klasse, die ein generische Bankkonto repräsentiert'
    anzKont = 0  # Klassenvariable die zählt, wie viele Konten erzeugt wurden

    def __init__(self, name, nummer, betrag):  # Konstruktor
        ' initialisiere neues Exemplar der Klasse '
        self.__inhaber = name
        self.__kontonummer = nummer
        self.__saldo = betrag
        Konto.anzKont += 1 # erh\"ohe die Klassenvariable um 1

    def einzahlen(self, betrag): # auf das Konto einzahlen
        'addiere zum Saldo hinzu'
        self.__saldo = self.__saldo + betrag

    def auszahlen(self, betrag): # vom Konto abheben
        'subtrahiere vom Saldo'
        self.__saldo = self.__saldo - betrag

    def getSaldo(self): # Saldo abfragen
        'gib aktuellen Kontostand zurück'
        return self.__saldo

    def putSaldo(self, kontostand):# neuen Saldo eintragen
        'aktualisiere den Kontostand'
        self.__saldo = kontostand

    def __del__(self): # Destruktor
        Konto.anzKont -= 1 # veringere Klassenvariable um 1

Ln: 32 Col: 0
```

Abbildung 5.8.: Die verbesserte Klasse Konto

Attribute Error

Tatsächlich liefert der Interpreter nun beim Durchlauf des Programms aus Abb. 5.6 einen *Attribute Error* und teilt uns mit, dass die Klasse "Konto" kein Attribut `saldo` besitzt (Abb. 5.9).

```
Python Shell
File Edit Shell Debug Options Windows Help

>>> ================================ RESTART ================================
>>>
Joes Kontostand ist: 1050.79
Anzahl der Konten: 2
Anzahl der Konten: 1
Traceback (most recent call last):
  File "/home/ronja/Dropbox/Skript/Code/konto.py", line 49, in <module>
    print("Janes Saldo beträgt:",k2.saldo) # Jane's Kontostand
AttributeError: 'Konto' object has no attribute 'saldo'
>>>

Ln: 71 Col: 4
```

Abbildung 5.9.: Ausgabe des Programms zur verbesserten Klasse Konto

Vererbung

Nun hatten wir schon zu Beginn des Beispiels festgestellt, dass eine reale Bank verschiedene Kontoarten unterscheiden können muss. Dementsprechend legen wir nun eine Kindklasse "Sparkonto" an. Der Name der Elternklasse wird bei der Definition der Kindklasse in Klammern hinter den Klassennamen geschrieben (Abb. 5.10).

```
sparkonto.py - /home/ronja/Dropbox/Skript/Code/sparkonto.py
File Edit Format Run Options Windows Help

class  Sparkonto (Konto): # Klasse Sparkonto, erbt von Konto
    'Klasse, die ein Sparkonto repräsentiert'

    def __init__(self, name, nummer, betrag, zins):
        Konto.__init__(self,name, nummer, betrag)
        self.__zinssatz = zins

    def _sparbuch_eintrag(self, betrag):
        print(betrag, 'neuer Saldo:' , self.getSaldo())

    def einzahlen(self, betrag):
        'addiere zum Saldo hinzu und schreibe in Sparbuch'
        self.putSaldo(self.getSaldo() + betrag)
        self._sparbuch_eintrag(betrag)

    def auszahlen(self, betrag):
        if((self.getSaldo() - betrag)>=0):
            self.putSaldo(self.getSaldo() - betrag)
            self._sparbuch_eintrag(betrag)
        else: print("Das Guthaben ist nicht ausreichend")

    def getZinssatz(self): # liefert Wert des Attributs zinssatz
        return self.__zinssatz

    def putZinssatz(self, zinsNeu): # weist Attribut zinssatz neuen Wert zu
        self.__zinssatz = zinsNeu
Ln: 90 Col: 78
```

Abbildung 5.10.: Die Klasse Sparkonto, Kindklasse von Konto

Im Konstruktor der Kindklasse rufen wir den Konstruktor der Elternklasse auf. Zusätzlich zu den Attributen der Elternklasse "Konto" benötigt das Sparkonto noch eine Attribut `zinssatz`. Dementsprechend hat der Konstruktor von "Sparkonto" einen weiteren Übergabeparameter `zins`, welcher im Konstruktor dem Attribut `zinssatz` zugewiesen wird. Für den Zugriff auf `zinssatz` programmieren wir die beiden Methoden `getZinssatz()` und `putZinssatz()`. Ferner implementieren wir die Methode `_sparbuch_eintrag(betrag)`, um Ein- und Auszahlungen im Sparbuch zu dokumentieren. Da diese Funktion in engem Zusammenhang mit Ein- und Auszahlungen steht und verlässlich bei jeder Ein- und Auszahlung ausgeführt werden muss, definieren wir die Methoden `einzahlen(betrag)` und `auszahlen(betrag)` der Elternklasse "Konto" für die Kindklasse "Sparkonto" neu. Diesen Vorgang nennt man auch *überschreiben* von Methoden. Beide Methoden rufen nun am Ende die `_sparbuch_eintrag(betrag)`-Methode auf. Die `auszahlen(betrag)`-Methode überprüft zusätzlich, ob der Saldo für die Auszahlung ausreichend ist.

überschreiben von Methoden

Nun können, wie zuvor bei der Klasse "Konto", Instanzen der Klasse "Sparkonto" erzeugt werden. Beim Aufruf der `einzahlen(betrag)` und `auszahlen(betrag)`-Methoden, werden die Methoden der Klasse "Sparkonto" ausgeführt. Da "Sparkonto" von "Konto" erbt, ist aber auch die Klassenvariable `Konto.anzKont` verwendbar (Abb. 5.11).

```
*sparkonto.py - /home/ronja/Dropbox/Skript/Code/sparkonto.py*
File Edit Format Run Options Windows Help

k1 = Sparkonto("Joe", 176358, 1000, 1.5) # Konto für Joe mit 1000 € anlegen
k2 = Sparkonto("Jane", 938674, 10000.00, 1.5) # Konto für Jane anlegen

k1.einzahlen(50.79) # 50,79 € auf Joes Konto einzahlen

k2.auszahlen(6000) # 6000 € von Janes Konto abheben

k1.auszahlen(2000) # 2000 € von Joes Konto abheben

print("Zinssatz für Janes Konto", k2.getZinssatz())
k2.putZinssatz(0.8) #Zinssatz für Janes ändern
print("Zinssatz für Janes Konto", k2.getZinssatz())

print("Anzahl der Konten:", Konto.anzKont) # Schauen, wie viele Konten es gibt
del k1 # Joes Konto löschen
print("Anzahl der Konten:", Konto.anzKont) # Schauen, wie viele Konten es gibt
Ln: 93 Col: 0
```

Abbildung 5.11.: Verwendung der Klasse Sparkonto

Aufgrund der Implementierung, dass nämlich jede Instanz der Klasse "Sparkonto" auch eine Instanz der Klasse "Konto" enthält, arbeitet der Zähler korrekt.

```
>>> ================================ RESTART ================================
>>>
50.79 neuer Saldo: 1050.79
6000 neuer Saldo: 4000.0
Das Guthaben ist nicht ausreichend
Zinssatz für Janes Konto 1.5
Zinssatz für Janes Konto 0.8
Anzahl der Konten: 2
Anzahl der Konten: 1
>>>
```

Abbildung 5.12.: Ausgabe des Programms zur Verwendung der Klasse Sparkonto

Mehrfachvererbung

Klassen können auch von mehreren Klassen erben. Dies bezeichnet man als *Mehrfachvererbung*. Statt nur einer Klasse innerhalb der Klammern hinter dem Klassennamen, wird eine durch Komma getrennte Liste aller Elternklassen angegeben von denen geerbt werden soll.

```
class Kindklasse (Elternklasse1, Elternklasse2, Elternklasse3,...)
```

Nun implementieren wir eine weitere Kindklasse der Klasse "Konto", die Klasse "Girokonto". Diese hat zusätzlich zu den Attributen der Elternklasse, die Attribute `kreditrahmen, sollzins` und `habenzins`. Während Haben- und Sollzins als Parameter übergeben werden, wird das Attribut `kreditrahmen` im Konstruktor mit einem Zehntel des Saldo initialisiert. Um auf `kreditrahmen` für spätere Änderungen noch zugreifen zu können, programmieren wir wieder zwei Zugriffsmethoden `getKreditrahmen()` und `putKreditrahmen()`. Ferner implementieren wir die Methode `ueberweisen()`, die als Übergabeparameter eine Instanz der Klasse Konto, und einen Betrag erwartet. Zunächst wird überprüft, ob der Kreditrahmen für die Transaktion ausreicht. Falls ja, wird der Betrag vom Konto abgezogen und dem angegebenen Konto gutgeschrieben. Die Gutschreibung geschieht über den Aufruf der `einzahlen()`-Methode des angegebenen Kontos (Abb. 5.13).

```
class Girokonto (Konto): # Klasse Girokonto, erbt von Konto
    'Klasse, die ein Girokonto repräsentiert'

    def __init__(self, name, nummer, betrag, sollzins, habenzins):
        Konto.__init__(self,name, nummer, betrag)
        self.__sollzins = sollzins
        self.__habenzins = habenzins
        self.__kreditrahmen = self.getSaldo()/10.0

    def getKreditrahmen(self):
        return self.__kreditrahmen

    def putKreditrahmen(self, val):
        self.__kreditrahmen = val

    def ueberweisen(konto, betrag):
        #überprüfe, ob Kreditrahmen ausreicht
        if((self.getSaldo()-betrag) >= -self._kreditrahmen):
            self.putSaldo(self.getSaldo()-betrag) # ziehe Betrag von diesem Konto ab
            konto.einzahlen(betrag) # Schreibe Betrag auf anderm Konto gut
        else: print('Der Kreditrahmen ist nicht ausreichend für die Überweisung')
```

Abbildung 5.13.: Die Klasse Girokonto, Kindklasse von Konto

Nun ist es uns möglich Giro- und Sparkonten anzulegen und auf diese Geld einzuzahlen, bzw. von diesen Geld auszuzahlen. Außerdem können wir Geld von Girokonten auf andere Girokonten, aber auch Sparkonten überweisen (Abb. 5.14).

girokonto.py - /home/ronja/Dropbox/Skript/Code/girokonto.py

File Edit Format Run Options Windows Help

```
g1 = Girokonto("Joe", 192386, 1000, 15.6, 0.5)
g2 = Girokonto("John", 395867, 20000, 15.6, 0.5)

k1 = Sparkonto("Joe", 176358, 1000, 1.5) # Konto für Joe mit 1000 € anlegen
k2 = Sparkonto("Jane", 938674, 10000.00, 1.5) # Konto für Jane anlegen

print("Kontostand Girokonto Joe:", g1.getSaldo())
print("Kontostand Girokonto John:", g2.getSaldo())
print("Kontostand Sparkonto Joe:", k1.getSaldo())
print("Kontostand Sparkonto Jane:", k2.getSaldo())

print("Überweisung von John an Joe")
g2.ueberweisen(g1,1000) # 1000 € von John an Joe überweisen

print("Kontostand Girokonto Joe:", g1.getSaldo())
print("Kontostand Girokonto John:", g2.getSaldo())

print("Überweisung von Joe, Girokonto auf Sparkonto")
g1.ueberweisen(k1,200) # 200 € vom Girokonto auf Sparkonto überweisen

print("Kontostand Girokonto Joe:", g1.getSaldo())
print("Kontostand Sparkonto Joe:", k1.getSaldo())
```

Ln: 112 Col: 0

Abbildung 5.14.: Verwendung der Klasse Girokonto

Dabei wird in der `ueberweisen()`-Methode der "Girokonto"-Klasse die `einzahlen()`-Methode der Klasse, der das Konto angehört, aufgerufen. Je nachdem, ob es sich dabei um ein Girokonto oder ein Sparkonto handelt, wird die `einzahlen()`-Methode der Klasse "Sparkonto" oder die `einzahlen()`-Methode der Klasse "Konto" (denn "Girokonto" hat keine eigene Methode `einzahlen()`) ausgeführt (Abb. 5.15). Die Methode `ueberweisen()` arbeitet also mit Instanzen der Klasse "Konto", aber auch mit Instanzen aller, von "Konto" abgeleiteten Klassen. Die Ausführung der Verhaltensweise `einzahlen()` ist abhängig von der Klasse der jeweiligen Instanz und wird erst zur Laufzeit des Programms aufgelöst. Dies nennt man auch *Polymorphie* (Vielgestaltigkeit)

Polymorphie

Python Shell

File Edit Shell Debug Options Windows Help

```
>>> ================================ RESTART ================================
>>>
Kontostand Girokonto Joe: 1000
Kontostand Girokonto John: 20000
Kontostand Sparkonto Joe: 1000
Kontostand Sparkonto Jane: 10000.0
Überweisung von John an Joe
Kontostand Girokonto Joe: 2000
Kontostand Girokonto John: 19000
Überweisung von Joe, Girokonto auf Sparkonto
200 neuer Saldo: 1200
Kontostand Girokonto Joe: 1800
Kontostand Sparkonto Joe: 1200
>>> 
```

Ln: 485 Col: 4

Abbildung 5.15.: Ausgabe des Programms zur Verwendung der Klasse Girokonto

Die objektorientiert Programmierung bietet viele Möglichkeiten und kann den Softwareentwurf, vor allem in der frühen Phase, deutlich vereinfachen. Allerdings trifft man immer wieder auf Probleme, die keine saubere objektorientierte Lösung bieten. Ferner bilden sich Kontrollflüsse in objektorientierten Programmen nicht mehr direkt in den Codestrukturen ab und sind daher für den Entwickler weniger transparent. Mit der wachsenden Bedeutung von paralleler Hardware und nebenläufiger Programmierung ist aber gerade eine bessere Kontrolle und Entwickler-Transparenz der komplexer werdenden Kontrollflüsse immer wichtiger.

Objektorientierte Programmierung ist *eine* Art und Weise Software zu entwerfen und zu implementieren. Sie hat ihre Vorteile, aber auch Nachteile. Ob sie sich für das zu lösende Problem eignet, ist der Softwareentwicklerin bzw -entwickler überlassen.

6. Einführung

"In der Informatik geht es genau so wenig um Computer wie in der Astronomie um Teleskope"
Edsger W. Dijkstra (1930-2002), Niederländischer Informatiker

Vielmehr ist der Computer lediglich ein Hilfsmittel dessen sich die Informatik bedient. Anstatt selbst zu rechnen, lässt man die Rechenmaschine (Computer) rechnen. Daher befasst sich die Informatik auch nicht mit der Bedienung von Software, sondern u.a. mit dem Entwurf derselbigen.

"Informatik ist die Wissenschaft von der systematischen Verarbeitung von Informationen, besonders der automatischen Verarbeitung mithilfe von Computern". (Duden Informatik).

Da wir es in der Informatik mit Computern zu tun haben, die Anweisungen nicht interpretieren können, sondern lediglich ausführen, ist es notwendig präzise zu Arbeiten und zu Formulieren und ganz sicher zu sein, dass eine entworfene Lösung in jeder auftretenden Situation funktioniert. Fehler in Software und Hardware haben in der Vergangenheit immer wieder zu großem wirtschaftlichen Schaden und auch zu Todesfällen geführt.

Hier einige Beispiele:

Der Pentium Prozessor Divisions-Fehler 1994
Für $x = 4195835.0$ und $y = 3145727.0$ und $z = x - \frac{x}{y} \times y$ liefert ein fehlerfreier Prozessor bei exakter Rechnung $z = 0$. Der Pentium Prozessor lieferte als Ergebnis $z = 256$.
Erklärung: INTEL verwendete einen speziellen Divisions-Algorithmus, der den Vorteil hatte, dass pro Takt 2 Bits des Quotienten bestimmt werden konnten und den Chip somit schneller machte. Allerdings sollte die Schätzung für die nächste Stelle aus einer Tabelle gelesen werden, die 1066 Einträge haben sollte. Durch eine fehlerhafte `for`-Schleife wurden nur 1061 Einträge auf den Chip aufgenommen, dadurch kam es zu falschen Ergebnissen bei der Gleitkomma-Division.
Laut INTEL trete der Fehler lediglich alle 27 000 Jahre einmal auf, IBM, die einen eigenen Chip herstellten, verkündete der Fehler trete ca alle 24 Tage auf. Da der Fehler erst nach einem Jahr entdeckt wurde, obwohl der Chip in vielen Geräten verbaut war, ist IBMs Behauptung schwer zu glauben. Letztendlich kostete dieser Fehler INTEL über 400 Millionen Dollar.

Therac-25 Bestrahlungsunfälle
Zwischen 1985 und 1987 kam es zu mehreren Unfällen mit dem medizinischen Bestrahlungsgerät *Therac-25*. Infolge einer Überdosis mussten mehreren Patienten Organe entfernt werden und drei Patienten starben sogar. Mehrere Fehler im Kontrollprogramm des Geräts führten zu den Unfällen.

Ariane 5
Die Ariane 5 Rakete der ESA musste am 4.Juni 1996 eine Minute nach dem Start gesprengt werden, da sie vom Kurs abgekommen war und sich nicht mehr steuern ließ.
Erklärung: Für die Steuerung der Ariane 5 hatte man die Steuerungssoftware der Ariane 4 übernommen. Die Ariane 5 war aber größer und erreichte eine höhere Geschwindigkeit als die Ariane 4. Dadurch kam es in der Kursberechnung zu Zahlen, die nicht vorgesehen waren und es kam zu einem Speicherüberlauf, die Zahlen konnten von der Software nicht mehr korrekt interpretiert werden. Die Steuerungssoftware korrigierte die vermeintlich falsche Flugbahn. Der Schaden betrug etwa 370 Millionen Dollar.

Flughafen Denver, Koffertransportsystem

Im Oktober 1993 sollte der neue Flughafen, ausgestattet mit einem vollautomatischen Koffertransportsystem, eröffnet werden. Wegen Schwierigkeiten mit dem vollautomatischen Gepäcktransportsystem verzögerte sich die Eröffnung um 16 Monate. Statt einer großen Gesamtlösung hatte jeder Flugsteig sein eignes, unterschiedlich stark automatisiertes Gepäcktransportsystem. Lediglich die größte Airline am Flughafen, United, benutzte das automatisierte Gepäcksystem für Abflüge und zahlte ca 1 Mio Dollar Instandhaltungskosten pro Monat. 2005 entschied United das automatisierte System aufzugeben.
Im Nachhinein stellte sich heraus, dass u.a. das Gesamtproblem zu komplex war und zu viele Nachrichten über das Netz verschickt werden mussten, sodass viele Sortier-Anweisungen nicht rechtzeitig ankamen.
Der Schaden wird auf 3,2 Milliarden Dollar geschätzt.

Wir wollen also Probleme und Aufgaben unseres täglichen Lebens mit Hilfe von Computern bearbeiten und lösen. Dafür müssen wir die, meist in Alltagssprache formulierten Wünsche und Vorstellungen eines Kunden modellieren und formalisieren, um dann eine Lösung zu entwerfen, die tatsächlich immer funktioniert, alle Eventualitäten berücksichtigt und im Idealfall verifizierbar ist.

Also benötigen wir formale Konstrukte und Konzepte, sodass Aussagen bewiesen und die Korrektheit von Systemen verifiziert werden kann. Unsere Alltagssprache eignet sich dafür nicht, da sie zu ungenau, und häufig mehrdeutig ist. Wir benötigen also die Möglichkeit Objekte und Konzepte anhand der, für die Problemstellung relevanten Eigenschaften, zusammenzufassen, sodass wir Aussagen über diese machen können und diese beweisen können.

7. Aussagenlogik

In der Informatik ist Logik ein zentraler Bestandteil hinsichtlich der korrekten Argumentation und des korrekten Beweisens von Aussagen. Beispielsweise findet sich in Programmiersprachen ihre Unentbehrlichkeit bei der Benutzung von Fallunterscheidungen (`if-then-else`-Konstrukte) wieder. Neben dieser *Kontrollflusssteuerung in Computerprogrammen* gibt es zahlreiche weitere wichtige und nützliche Anwendungen der Aussagenlogik:

- Modellierung von Wissen (z.B. künstliche Intelligenz)
- Auswertung von Datenbankanfragen
- Logikbauteile in der technischen Informatik (Hardware)
- Automatische Verifikation (automatisches Testen eines Systems auf dessen Funktionstüchtigkeit)
- Mathematische Beweise
- Korrektes Argumentieren

Im Folgenden möchten wir die fundamentalen Bestandteile der Aussagenlogik kurz vorstellen und erläutern, wie man Wissen formal mit dem Werkzeug der Logik beschreiben kann.

7.1. Einführung

Bevor wir die Formalismen einführen, müssen wir uns zunächst klar darüber werden, **was** genau eigentlich Aussagen sind, mit denen wir arbeiten können.

Definition 7.1.
Eine *logische Aussage* besteht aus einer oder mehreren Teilaussagen, denen die Wahrheitswerte **0** (für `falsch`) oder **1** (für `wahr`) zugeordnet werden können. Eine Teilaussage nimmt dabei niemals gleichzeitig mehrere Werte an.

Aussage
Wahrheitswert

Mit welchem Wert man dabei eine Teilaussage belegt, hat keinen Einfluss darauf, ob man etwas „Aussage“ nennt oder nicht - lediglich die Eigenschaft, dass man es **kann**, ist der Indikator einer korrekten logischen Aussage.

Beispiel 7.2.
Betrachte folgende Ausdrücke, welche eine *logische Aussage* repräsentieren:

- Die Sonne scheint.
- Es ist hell.
- Der Tisch ist blau.

Offensichtlich können diese Ausdrücke **entweder** wahr **oder** falsch sein und erfüllen damit Definiton 7.1.

Vergleiche obige nun mit folgenden Ausdrücken, welche keine logische Aussage darstellen:

- „$5+5$“ , „$\frac{5}{7}$“ , „$3 \cdot 2$“ , usw.
- 15 ist eine schöne Zahl („schön“ ist für Zahlen nicht definiert)
- Fragen („Wie spät ist es?“) und Aufforderungen („Lach mal!“)

Ihnen können keine konkreten Wahrheitswerte (0 oder 1) zugeordnet werden, sodass sie Definition 7.1 nicht entsprechen.

Im Weiteren möchten wir die fundamentalen Werkzeuge der Aussagenlogik vorstellen, wozu wir die **atomaren Aussagen**, die **Negation** $\neg$, die **Konjunktion** $\wedge$ (Ver*und*ung), die **Disjunktion** $\vee$ (Ver*oder*ung), die **Implikation** $\rightarrow$ und die **Biimplikation** $\leftrightarrow$ zählen.

Beispiel 7.3 (Atomare Aussagen).

atomare Aussage

Die einfachsten *atomaren Aussagen* sind **0** (ein Ausdruck, der immer falsch ist) und **1** (ein Ausdruck, der immer wahr ist). Allgemeiner ist das eine Aussage mit der Eigenschaft, nicht weiter zerlegt werden zu können. Weiter ist es bei der Formalisierung eines sprachlichen Satzes notwendig zu erkennen, welches die Teilaussagen sind.
Der Satz „**Wenn** ich Hunger habe **und** der Supermarkt geöffnet hat, **dann** gehe ich einkaufen.“ , den wir als φ bezeichnen, besitzt drei Teilaussagen:

- $A :=$ „Ich habe Hunger.“
- $B :=$ „Der Supermarkt hat geöffnet.“
- $C :=$ „Ich gehe einkaufen.“

Notation 7.4.
Atomare Aussagen kürzen wir immer mit den Buchstaben A, B, $\ldots$, Z ab.

Rein formal hat unser obiger Satz die Form

$$\varphi = ((A \wedge B) \rightarrow C)$$

Lies: Wenn A und B wahr sind, dann ist auch C wahr.

Wie lässt sich nun erkennen, wann φ eine wahre Aussage und wann φ eine falsche Aussage darstellt? Dazu muss man sich die möglichen Belegungen von A, B und C ansehen. Das macht man meist mit einer Wahrheitstabelle (Wahrheitstafel). Wir werden später hierauf nochmal genauer eingehen, wenn wir die Bedeutung (Semantik) einer Konjunktion und einer Implikation behandelt haben.

Anmerkung 7.5.

Belegung

Eine *Belegung* für eine aussagenlogische Variable A (bzw. atomare Aussage) ist entweder 0 oder 1, d.h. A kann den Wert 0 (falsch) oder den Wert 1 (wahr) annehmen.

Beispiel 7.6 (Negation / Verneinung $\neg$).

Negation $\neq$

Jede logische Aussage A lässt sich auch in ihrer negierten Form $\neg A$ darstellen. Dabei gilt, dass $\neg A$ **genau dann** wahr ist, **wenn** A falsch ist. Betrachten wir hierzu folgenden Satz φ: „Alle Kinder spielen gern.“
Dann ist

$$\neg\varphi = \text{„Nicht alle Kinder spielen gern.“}$$

Umgangssprachlich bedeutet $\neg\varphi$, dass es (mindestens) ein Kind gibt, das nicht gerne spielt.
Des Weiteren ist bei der Verneinung von Aussagen zu beachten, an welcher Stelle negiert werden muss. Es wäre falsch zu sagen, dass $\neg\varphi$ der Aussage „Alle Kinder spielen ungern.“ entspricht.

Beispiel 7.7 (Konjunktion $\wedge$).
Es ist möglich mehrere Teilaussagen miteinander in Form einer Ver*und*ung zu verknüpfen, sodass die dadurch entstehende komplexere Aussage genau dann wahr ist, wenn beide Teilaussagen gleichzeitig wahr sind.
Der Satz φ „Die Sonne scheint **und** der Wind weht stark.“ ist nur dann wahr, wenn sowohl die Aussage A „Die Sonne scheint.“ als auch die Aussage B „Der Wind weht stark.“ wahr ist.

Konjunktion $\wedge$

Formal entspricht obiger Satz der Konjunktion von A und B, also

$$\varphi = A \wedge B$$

Beispiel 7.8 (Disjunktion $\vee$).
Analog zur in Beispiel 7.7 eingeführten Konjunktion ist auch eine Ver*oder*ung zweier Teilaussagen A und B möglich. Die dadurch entstehende komplexere Aussage ist genau dann wahr, wenn mindestens eine der Teilaussagen wahr ist.
Betrachten wir den Satz φ „Der Vorhang ist rot oder der Koffer ist blau.“, welcher dann wahr ist, wenn der Vorhang rot ist (A) **oder** der Koffer blau ist (B) **oder** beides der Fall ist.

Disjunktion $\vee$

Formal entspricht obiger Satz der Disjunktion von A und B, also

$$\varphi = A \vee B$$

Hinweis 7.9.
Eine Disjunktion ist **keine** ausschließende Veroderung im Sinne von „entweder ... oder ..., aber nicht beides zusammen“. Hierzu verwendet man in der Aussagenlogik das `exklusive Oder` (formal: $A \oplus B$ bzw. $A \dot{\vee} B$), welches wir im Vorkurs nicht ausführlich behandeln, sondern nur kurz ansprechen.

Beispiel 7.10 (Implikation $\rightarrow$).
Unter einer Implikation versteht man die Folgerung des Sachverhalts B aus einem Sachverhalt A heraus (wie in Beispiel 7.3, in dem wir eine Implikation bereits behandelt haben). Man hat also eine Voraussetzung, die eine Konsequenz zur Folge hat.

Implikation $\rightarrow$

Der Satz φ „**Wenn** es regnet, **dann** ist die Straße nass.“ wird wahr, wenn beide Teilaussagen wahr sind oder die Voraussetzung falsch ist.

Zur Verdeutlichung: Wenn es regnet und die Straße nass ist, dann ist φ logischerweise wahr. Allerdings kann die Straße auch nass sein, wenn es nicht regnet - dann ist unsere Folgerung trotzdem richtig. Umgekehrt wird φ falsch, falls es regnet, die Straße aber nicht nass ist - denn dann ist die Voraussetzung zwar erfüllt, aber die Konsequenz tritt nicht ein.

Als Leitsatz kann man sich Folgendes merken: *Aus Wahrem kann man nur Wahres folgern - aus Falschem kann man alles folgern.*
Entsprechend sind auch die Formalismen einer Implikation φ „Aus A folgt B“ bzw. „Wenn A, dann B“ klar:

$$\varphi = (A \rightarrow B) \quad \text{oder auch} \quad \varphi = (\neg A \vee B)$$

Beispiel 7.11 (Biimplikation $\leftrightarrow$).
Eine Biimplikation ist, wie sich aus dem Namen schließen lässt, eine Implikation in beide Richtungen. Sprachlich wird das durch das Konstrukt **„... genau dann, wenn ...“** deutlich.
Der Satz φ „Der Schornstein raucht **genau dann, wenn** die Heizung an ist.“ wird wahr, wenn sowohl die Aussage „Der Schornstein raucht, wenn die Heizung an ist.“ ($A \rightarrow B$) als auch die Aussage „Die Heizung ist an, wenn der Schornstein raucht.“ ($B \rightarrow A$) wahr ist.
Formal entspricht φ also einer Biimplikation, das heißt:

Biimplikation $\leftrightarrow$

$$\varphi = (A \leftrightarrow B) \quad \text{oder auch} \quad \varphi = ((A \rightarrow B) \wedge (B \rightarrow A))$$

Notation 7.12.
Die Zeichen $\neg$, $\wedge$, $\vee$, $\rightarrow$ und $\leftrightarrow$ heißen *Junktoren*, mit denen wir Teilaussagen zu komplexen Aussagen verknüpfen können.

Junktoren

Nachdem wir nun die wichtigsten Konstrukte der Aussagenlogik eingeführt haben, können wir einige größere Aussagen formalisieren.

Aufgabe 7.1.
Finde die Teilaussagen des folgenden Satzes und stelle eine äquivalente aussagenlogische Formel auf: „Ich fahre an der Ampel genau dann los, wenn die Ampel grün und die Straße frei ist.“

Lösung:
Die Grundidee bei der Lösungsfindung ist prinzipiell immer gleich: Man sucht nach bestimmten Schlüsselwörtern: [**und**], [**oder**], [**entweder** ... **oder**], [**wenn** ..., **dann** ...], [... **genau dann, wenn** ...] und [**nicht**]. Hat man solche Wörter gefunden, so muss man die Teilaussagen links bzw. rechts davon nur noch aufsplitten und anschließend entsprechend des verknüpfenden Schlüsselworts mit dem richtigen Junktor verbinden.

Wir haben drei Teilaussagen. Setze

$$A := \text{„Ich fahre an der Ampel los.“}$$
$$B := \text{„Die Ampel ist grün.“}$$
$$C := \text{„Die Straße ist frei.“}$$

Die aussagenlogische Formel lautet: $(A \leftrightarrow (B \wedge C))$

Aufgabe 7.2.
Finde die Teilaussagen des folgenden Satzes und stelle eine äquivalente aussagenlogische Formel auf: „Wenn ich keinen Spicker habe und den Multiple-Choice-Test bestehen möchte, dann muss ich die Antworten wissen oder Glück beim Raten haben.“

Lösung:
Wir haben vier Teilaussagen. Setze

$$A := \text{„Ich habe einen Spicker.“}$$
$$B := \text{„Ich möchte den Multiple-Choice-Test bestehen.“}$$
$$C := \text{„Ich muss die Antworten wissen.“}$$
$$D := \text{„Ich muss Glück beim Raten haben.“}$$

Die aussagenlogische Formel lautet: $((\neg A \wedge B) \rightarrow (C \vee D))$

Hinweis: Man hätte auch $A :=$ „Ich habe keinen Spicker.“ setzen können, sodass man als aussagenlogische Formel $((A \wedge B) \rightarrow (C \vee D))$ erhält, was auch eine korrekte Lösung darstellt.

7.2. Aussagenlogik

In diesem Abschnitt möchten wir die in der Einführung erläuterten Begriffe, die wir zunächst anhand von Beispielen behandelt haben, exakt definieren. Dazu legen wir die Syntax (was genau sind eigentlich gültige aussagenlogische Formeln?) und die Semantik (welche Bedeutung haben bestimmte Konstrukte?) fest.

7.2.1. Syntax der Aussagenlogik

Syntax

Die Syntax beschreibt **gültige** (aussagenlogische) Formeln und schließt implizit ungültige aus. Zum besseren Verständnis kann man die Syntax mit dem korrekten Satzbau in der deutschen Sprache vergleichen: „Subjekt, Prädikat, Objekt“ ist erlaubt (z.B. „Alfons mäht den Rasen.“), wohingegen „Objekt, Subjekt, Prädikat“ nicht erlaubt ist („Den Rasen Alfons mäht.“).

In unserem Kontext möchten wir beispielsweise $(A \rightarrow B)$ zulassen und $(\rightarrow A)B$ verbieten.

Definition 7.13.
Gegeben ist eine unendliche Menge aussagenlogischer Variablen $A, \ldots, Z, A_1, \ldots$, wobei eine aussagenlogische Variable alleine eine gültige Formel darstellt. Komplexere Formeln werden aus den Zeichen (und) sowie den in Notation 7.12 aufgezählten Junktoren $\neg$, $\wedge$, $\vee$, $\rightarrow$ und $\leftrightarrow$ nach folgenden Konstruktionsregeln erstellt:

Basisfall:
Sowohl **aussagenlogische Variablen**, als auch **0** und **1**, sind gültige Formeln.

Rekursionsfälle:

1. Ist A eine gültige Formel, so ist auch $\neg A$ eine gültige Formel.
2. Sind A und B gültige Formeln, so ist auch $(A \wedge B)$ eine gültige Formel.
3. Sind A und B gültige Formeln, so ist auch $(A \vee B)$ eine gültige Formel.
4. Sind A und B gültige Formeln, so ist auch $(A \rightarrow B)$ eine gültige Formel.
5. Sind A und B gültige Formeln, so ist auch $(A \leftrightarrow B)$ eine gültige Formel.

Beispiel 7.14.
Gemäß Definition 7.13 ist $(A \leftrightarrow (B \wedge C))$ eine gültige Formel, wobei $(A \vee B \vee C)$ (Klammern fehlen) und $(A \leftarrow B)$ („$\leftarrow$“ nicht definiert) keine gültigen Formeln sind.

7.2.2. Semantik der Aussagenlogik

In der Semantik möchten wir die Bedeutung einzelner Konstrukte festlegen, was prinzipiell nichts Anderes ist, als Regeln zu definieren, die uns sagen, wann eine aussagenlogische Formel in Abhängigkeit der Belegungen ihrer Variablen wahr wird. Wie bereits oben angesprochen werden wir das anhand von **Wahrheitstabellen** (Wahrheitstafeln) machen. In den Spalten links neben dem Doppelstrich stehen alle aussagenlogischen Variablen, während rechts neben dem Doppelstrich die zu untersuchende Formel auftritt. In den Zeilen werden dann alle möglichen Belegungen der Variablen aufgeführt und in Abhängigkeit davon die Formel ausgewertet. Die sprachlichen Begründungen finden sich im Abschnitt 7.1, sodass wir im Folgenden nur noch die exakten Wahrheitswerte auflisten.

Definition 7.15.
Gegeben sind zwei aussagenlogische Variablen A und B. Folgende Wahrheitstabellen zeigen abhängig von den Belegungen von A und B die Auswertung für die rechts neben dem Doppelstrich stehenden Formeln:

A	$\neg A$
0	1
1	0

A	B	$(A \wedge B)$	$(A \vee B)$	$(A \rightarrow B)$	$(A \leftrightarrow B)$	$(A \oplus B)$
0	0	0	0	1	1	0
0	1	0	1	1	0	1
1	0	0	1	0	0	1
1	1	1	1	1	1	0

Anmerkung 7.16.
Das exklusive Oder (XOR) $A \oplus B$ lässt sich auch mit $((\neg A \wedge B) \vee (A \wedge \neg B))$ oder auch mit $((A \vee B) \wedge \neg(A \wedge B))$ darstellen.

Wie geht man nun an die Berechnung einer größeren Formel heran? Als nützlich und vor allem übersichtlich erweist es sich immer, die komplexe Formel in ihre Teilformeln zu zerlegen und schrittweise weiterzurechnen. Wie genau das vonstatten geht, wollen wir anhand von Beispiel 7.3 sowie Aufgabe 7.1 und Aufgabe 7.2 zeigen.

Beispiel 7.17.
Wir fragen uns, mit welchen Belegungen die in Beispiel 7.3 angeführte Formel $\varphi = ((A \wedge B) \to C)$ („**Wenn** ich Hunger habe **und** der Supermarkt geöffnet hat, **dann** gehe ich einkaufen.“) wahr wird. Dazu stellen wir die Wahrheitstabelle auf:

A	B	C	$(A \wedge B)$	$((A \wedge B) \to C)$
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	1	0
1	1	1	1	1

Offensichtlich ist unsere Formel φ **bis auf die Belegung** $A = 1$, $B = 1$, $C = 0$ immer wahr.

Dies lässt sich leicht aus der links stehenden Wahrheitstabelle ablesen.

Die Aufspaltung der Formel hat die Berechnung erleichtert.

Beispiel 7.18.
Wir fragen uns, mit welchen Belegungen die in Aufgabe 7.1 angeführte Formel $\varphi = (A \leftrightarrow (B \wedge C))$ wahr wird. Dazu stellen wir die Wahrheitstabelle auf:

A	B	C	$(B \wedge C)$	$(A \leftrightarrow (B \wedge C))$
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Betrachte die Zeilen, bei denen $A = 1$ („Ich fahre an der Ampel los“): Die Formel wird nur dann war, wenn die Ampel grün und die Straße frei ist.

Wenn $A = 0$ („Ich fahre nicht an der Ampel los“), dann stimmt die Formel logischerweise unabhängig von B und C, es sei denn, $B = C = 1$ (also Ampel ist grün und Straße ist frei), denn genau dann, wenn ich dann nicht losfahre ($A = 0$), stimmt unsere Formel nicht mehr.

Beispiel 7.19.
Wir fragen uns, mit welchen Belegungen die in Aufgabe 7.2 angeführte Formel $\varphi = ((\neg A \wedge B) \to (C \vee D))$ wahr wird. Dazu stellen wir die Wahrheitstabelle auf:

A	B	C	D	$\neg A$	$(\neg A \wedge B)$	$(C \vee D)$	$((\neg A \wedge B) \to (C \vee D))$
0	0	0	0	1	0	0	1
0	0	0	1	1	0	1	1
0	0	1	0	1	0	1	1
0	0	1	1	1	0	1	1
0	1	0	0	1	1	0	0
0	1	0	1	1	1	1	1
0	1	1	0	1	1	1	1
0	1	1	1	1	1	1	1
1	0	0	0	0	0	0	1
1	0	0	1	0	0	1	1
1	0	1	0	0	0	1	1
1	0	1	1	0	0	1	1
1	1	0	0	0	0	0	1
1	1	0	1	0	0	1	1
1	1	1	0	0	0	1	1
1	1	1	1	0	0	1	1

Die richtige Interpretation der Werte überlassen wir dem Leser.

Satz 7.20.
Die Wahrheitstabelle für eine aussagenlogische Formel mit insgesamt n Variablen hat genau 2^n Zeilen.

7.2.3. Erfüllbarkeit, Allgemeingültigkeit und Äquivalenz

In diesem Abschnitt möchten wir aussagenlogische Formeln genauer klassifizieren und miteinander vergleichen. Dazu führen wir folgende Begriffe ein:

Definition 7.21.
Gegeben ist eine aussagenlogische Formel φ.

(*a*) φ heißt *erfüllbar*, wenn es (mindestens) eine Belegung der Variablen gibt, sodass die Formel den Wahrheitswert 1 hat. (Erfüllbarkeit)

(*b*) φ heißt *Kontradiktion* (oder *unerfüllbar*), wenn alle Belegungen der Variablen dazu führen, dass die Formel den Wahrheitswert 0 hat. (Kontradiktion)

(*c*) φ heißt *Tautologie* (oder *allgemeingültig*), wenn jede Belegung der Variablen dazu führt, dass die Formel den Wahrheitswert 1 hat. (Tautologie)

Beispiel 7.22. (*a*) Die Formel $((A \wedge B) \to C)$ ist nach Beispiel 7.17 erfüllbar (aber nicht allgemeingültig).

(*b*) Die Formel $(A \wedge \neg A)$ ist eine Kontradiktion (unerfüllbar), genauso wie die Formel $((A \to B) \wedge (A \wedge \neg B))$.

(*c*) Die Formel $(A \vee \neg A)$ ist eine Tautologie (allgemeingültig), genauso wie die Formel $((A \vee B) \vee (\neg A \wedge \neg B))$.

Wie erkennt man, welcher „Klasse“ eine gegebene aussagenlogische Formel angehört? Einer komplexeren Formel sieht man das meistens nicht direkt an! Letztendlich bleibt nichts anderes übrig, als alle möglichen Belegungen durchzurechnen.

Anmerkung 7.23. (a) Eine gegebene Formel φ ist genau dann **erfüllbar**, wenn in der Wahrheitstabelle für φ mindestens eine 1 steht.

(b) Eine gegebene Formel φ ist genau dann eine **Kontradiktion** (unerfüllbar), wenn in der Wahrheitstabelle für φ ausschließlich 0 stehen.

(c) Eine gegebene Formel φ ist genau dann eine **Tautologie** (allgemeingültig), wenn in der Wahrheitstabelle für φ ausschließlich 1 stehen.
Eine gegebene Formel φ ist genau dann eine **Tautologie** (allgemeingültig), wenn $\neg\varphi$ eine Kontradiktion (unerfüllbar) ist.

Mittlerweile ist es uns möglich, aussagenlogische Formeln in gewisse „Klassen“ einzuteilen. Folgend möchten wir nun zwei gegebene Formeln α und β miteinander vergleichen und feststellen, ob diese äquivalent sind.

Definition 7.24.
Gegeben sind zwei aussagenlogische Formeln α und β, wobei M die Menge der Variablen in α und N die Menge der Variablen in β ist. Wir sagen, dass α **äquivalent zu** β ist („$\alpha \equiv \beta$“), wenn für alle Belegungen der Variablen in $M \cup N$ der Wahrheitswert von α mit dem Wahrheitswert von β übereinstimmt.

Äquivalenz

Es stellt sich nach dieser Definition nun die Frage, wie man die Äquivalenz überprüfen kann. Der relativ aufwändige Ansatz besteht wieder darin, die Wahrheitstabellen aufzustellen und die Spalten von α und β auf Gleichheit zu überprüfen.

Beispiel 7.25.
Wir betrachten die beiden Formeln $\alpha = (A \wedge (A \vee B))$ mit $M = \{A, B\}$ und $\beta = A$ mit $N = \{A\}$. Dann ist $M \cup N = \{A, B\}$. Die Wahrheitstabelle sieht wie folgt aus:

A	B	$(A \vee B)$	α	β
0	0	0	0	0
0	1	1	0	0
1	0	1	1	1
1	1	1	1	1

Offensichtlich stimmen die Werte in den Spalten von α und β überein, sodass wir nach Definition 7.24 sagen können, dass $\alpha \equiv \beta$ gilt.

7.2.4. Fundamentale Rechenregeln

Abschließend stellen wir noch einige wichtige Regeln im Umgang mit aussagenlogischen Formeln vor.

Satz 7.26.
Gegeben seien aussagenlogische Formeln A, B und C.

(a). ***Doppelte Negation***

1. $\neg\neg A \equiv A$

(b). ***Kommutativgesetze***

1. $(A \wedge B) \equiv (B \wedge A)$
2. $(A \vee B) \equiv (B \vee A)$

(c). ***Assoziativgesetze***

1. $((A \wedge B) \wedge C) \equiv (A \wedge (B \wedge C))$
2. $((A \vee B) \vee C) \equiv (A \vee (B \vee C))$

(d). ***Distributivgesetze***

1. $((A \wedge B) \vee C) \equiv ((A \vee C) \wedge (B \vee C))$
2. $((A \vee B) \wedge C) \equiv ((A \wedge C) \vee (B \wedge C))$

***Wahrheitstabelle zu* 1.**

A	B	C	$(A \wedge B)$	$(A \vee C)$	$(B \vee C)$	$((A \wedge B) \vee C)$	$((A \vee C) \wedge (B \vee C))$
0	0	0	0	0	0	**0**	**0**
0	0	1	0	1	1	**1**	**1**
0	1	0	0	0	1	**0**	**0**
0	1	1	0	1	1	**1**	**1**
1	0	0	0	1	0	**0**	**0**
1	0	1	0	1	1	**1**	**1**
1	1	0	1	1	1	**1**	**1**
1	1	1	1	1	1	**1**	**1**

***Wahrheitstabelle zu* 2.**

A	B	C	$(A \vee B)$	$(A \wedge C)$	$(B \wedge C)$	$((A \vee B) \wedge C)$	$((A \wedge C) \vee (B \wedge C))$
0	0	0	0	0	0	**0**	**0**
0	0	1	0	0	0	**0**	**0**
0	1	0	1	0	0	**0**	**0**
0	1	1	1	0	1	**1**	**1**
1	0	0	1	0	0	**0**	**0**
1	0	1	1	1	0	**1**	**1**
1	1	0	1	0	0	**0**	**0**
1	1	1	1	1	1	**1**	**1**

(e). ***De Morgan'sche Gesetze***

1. $\neg(A \wedge B) \equiv (\neg A \vee \neg B)$
2. $\neg(A \vee B) \equiv (\neg A \wedge \neg B)$

(f). ***Absorptionsgesetze***

1. $(A \vee (A \wedge B)) \equiv A$
2. $(A \wedge (A \vee B)) \equiv A$

(g). ***Tertium non Datur***

1. $(A \wedge 1) \equiv A$
2. $(A \vee 1) \equiv 1$
3. $(A \wedge 0) \equiv 0$
4. $(A \vee 0) \equiv A$
5. $(A \wedge A) \equiv A$
6. $(A \vee A) \equiv A$

(h). ***Elimination der Implikation***

- $(A \to B) \equiv (\neg A \vee B)$

(i). ***Elimination der Biimplikation***

- $(A \leftrightarrow B) \equiv ((A \to B) \wedge (B \to A)) \equiv ((\neg A \vee B) \wedge (\neg B \vee A))$

Anmerkung 7.27.
Die Rechenregeln aus Satz 7.26 lassen sich wie in 7.26(d) durch Aufstellen der Wahrheitstabellen einfach nachprüfen. Entsprechende Aufgaben für 7.26(e) und 7.26(f) finden sich auf dem Übungsblatt.

8. Mengen

In diesem Kapitel stellen wir Mengen und einige ihrer grundlegenden Eigenschaften vor. Zunächst werden wir genau definieren, was wir unter einer *Menge* verstehen wollen. Zum besseren Verständnis geben wir oft Beispiele an. Dann können wir, nur unter Zugrundelegen dieser Definition, einige Eigenschaften von Mengen zeigen. Wir formulieren sie erst als konkrete Behauptung, auch *Satz* genannt. Weniger umfangreiche bzw. leichter zu beweisende Sätze werden hin und wieder auch als *Lemma* oder *Proposition* bezeichnet. Den Satz beweisen wir anschließend mit Hilfe der Methoden, die wir im Aussagenlogik-Kapitel (Kap. 7) dieses Skripts bereits kennen gelernt haben, und unter Rückgriff auf die eingeführten Definitionen. Außerdem dürfen wir alle Voraussetzungen in unserer Argumentation benutzen, die im Satz auftauchen. Lautet dieser beispielsweise

„Jede endliche Menge ist in einer anderen endlichen Menge enthalten“, so können wir uns im Beweis auf endliche Mengen beschränken und deren Endlichkeit auch für Schlussfolgerungen benutzen.

Definition 8.1.
Eine *Menge* ist eine Zusammenfassung von wohlunterschiedenen Objekten. Letztere nennen wir *Elemente* der Menge. Hierbei schreiben wir $m \in M$, um auszudrücken, dass m ein Element der Menge M ist. Gleichermaßen drücken wir mit $m \notin M$ aus, dass m kein Element der Menge M ist. Für $a \in M$ und $b \in M$ schreiben wir kurz $a, b \in M$.

Menge
Element
$\in$
$\notin$

Der Begriff *wohlunterschieden* in der obigen Definition besagt, dass die Elemente verschieden sind und von uns auch als voneinander verschieden angesehen werden. Zum Beispiel sind die Farben „Dunkelrot“ und „Hellrot“ an sich verschieden, könnten aber, wenn wir nur gröber unterscheiden wollten, als gleich angesehen werden, da sie ja beide Rottöne sind. Wenn beide in derselben Menge auftreten, wollen wir sie aber *nicht* als gleich bezeichnen. Zum Beispiel könnten wir eine Menge *Rottöne* definieren, darin würden etwa *Hellrot* und *Dunkelrot* als (wohlunterschiedene) Elemente auftreten.

Die Elemente einer Menge müssen aber nicht gleichartig sein. So können Personen, Zahlen, Namen, Farben allesamt gemeinsam in einer Menge auftreten, wenn wir diese so definieren. Mengen können *implizit* oder *explizit* angegeben werden. Letzteres geschieht durch Aufzählung aller Elemente, dabei öffnet man vor dem ersten genannten Element eine geschweifte Klammer $\{$ und schließt nach dem letzten eine $\}$. Um Mengen implizit anzugeben, kann man sie mit Worten oder wie in der Definition für $\mathbb{Q}$ im folgenden Beispiel beschreiben.

explizit

implizit

Beispiel 8.2.

Implizit angegebene Mengen:

- die Menge aller Buchstaben des deutschen Alphabets
- die Menge aller Primzahlen
- die Menge aller von Frankfurt am Main aus angeflogenenen Ziele
- die Menge aller Einwohner Deutschlands am 24.08.2013
- die Menge der rationalen Zahlen $\mathbb{Q} := \{\frac{a}{b} \mid a, b \in \mathbb{Z}, b \neq 0\}$

- die Menge der reellen Zahlen $\mathbb{R}$
- $M := \{z \in \mathbb{Z} \mid \text{es gibt ein } k \in \mathbb{N} \text{ mit } z = 5 \cdot k\}$

Explizit angegebene Mengen:

- die leere Menge $\emptyset$ (Menge, die keine Elemente enthält. Andere Schreibweise: $\{\}$)
- $P := \{\text{ich, du, er, sie, es, wir, ihr, Sie}\}$
- $\mathbb{N} := \{0, 1, 2, 3, \dots\}$ (Menge der natürlichen Zahlen)
- $\mathbb{Z} := \{0, 1, -1, 2, -2, 3, -3, \dots\}$ (Menge der ganzen Zahlen)

Der einzelne vertikale Strich in einer Menge wird als „sodass“ gelesen: Beispielsweise ist die oben angegebene Menge M die „Menge aller ganzen Zahlen z, *sodass* es eine natürliche Zahl k mit $z = 5 \cdot k$ gibt“, d.h. M ist die Menge aller durch 5 teilbaren Zahlen.

Aufgabe 8.1.
Gib die Menge aller Primzahlen unter Verwendung des vertikalen Strichs an. Versuche, eine ähnliche Darstellung für die Menge aller geraden Zahlen zu finden.

Wenn wir Sätze über Mengen formulieren wollen, müssen wir die betreffenden Mengen miteinander vergleichen können. Dafür brauchen wir die folgende Definition.

Definition 8.3.
Seien M und N zwei Mengen.

Gleichheit

(a) Wir bezeichnen M und N als *gleich* (in Zeichen $M = N$), wenn M und N dieselben Elemente enthalten.

Teilmenge

(b) M ist eine *Teilmenge* von N (in Zeichen $M \subseteq N$), wenn jedes Element von M auch ein Element von N ist.

(c) M ist eine *echte Teilmenge* von N (in Zeichen $M \subsetneq N$), wenn $M \subseteq N$, aber nicht $M = N$ gilt.

Obermenge

(d) M ist eine *Obermenge* von N (in Zeichen $M \supseteq N$), wenn $N \subseteq M$ gilt.

(e) M ist eine *echte Obermenge* von N (in Zeichen $M \supsetneq N$), wenn $N \subsetneq M$ gilt.

Gilt für zwei Mengen M und N Gleichheit, so identifizieren wir sie miteinander. Wir sagen dann, es handelt sich um *dieselbe* Menge, und können sie je nach Kontext und Belieben mit M oder N bezeichnen. Genauso identifizieren wir Elemente von Mengen miteinander: Gibt es ein x, was Element einer Menge M und auch Element einer Menge N ist, so sprechen wir nicht von „dem x aus M“ einerseits und „dem x aus N“ andererseits, sondern von „dem x, was Element von M und auch von N ist“ bzw. „dem x, was gemeinsames Element von M und N ist“.

Anstelle von $M \subsetneq N$ bzw. $M \supsetneq N$ findet man auch oft $M \subset N$ bzw. $M \supset N$ in der Literatur. Manchmal ist damit aber auch $M \subseteq N$ bzw. $M \supseteq N$ gemeint. Um Missverständnissen vorzubeugen, wählen wir die eindeutige Schreibweise aus Definition 8.3.

Beispiel 8.4.
Wir können uns schnell überzeugen, dass die folgenden Beziehungen wahr sind:

- $M = M$ und $\emptyset \subseteq M$ für jede Menge M

- $M \subseteq M$ (und $M \supseteq M$) für jede Menge M
- {ich, du} $\subset$ {ich, du, er, sie, es, wir, ihr, Sie}
- $\{1, 2\} = \{2, 1\} = \{1, 2, 2\}$
- $\mathbb{N} \subseteq \mathbb{Z}$, sogar $\mathbb{N} \subsetneq \mathbb{Z}$
- $\mathbb{Z} \subseteq \mathbb{Q}$, sogar $\mathbb{Z} \subsetneq \mathbb{Q}$
- $\mathbb{Q} \subseteq \mathbb{R}$, sogar $\mathbb{Q} \subsetneq \mathbb{R}$

Reihenfolge

Das vierte Beispiel oben betont, dass die Gleichheit von Mengen nicht von der Reihenfolge ihrer Elemente abhängt. Das heißt, eine Menge ist durch ihre Elemente eindeutig identifiziert – es gibt nur *eine einzige* Menge, die genau diese Elemente und keine weiteren beinhaltet. Außerdem wird in der expliziten Schreibweise normalerweise jedes Element nur einmal erfasst. Wir schreiben also $\{1, 2\}$ und nicht $\{1, 2, 2\}$. Dies liegt daran, dass die Elemente einer Menge nach Definition 8.1 wohlunterschieden sind. Die beiden Zweier in $\{1, 2, 2\}$ können wir aber nur durch ihre Position in der Menge unterscheiden – weil aber die Reihenfolge keine Rolle spielt, geht auch das nicht. Also ist $\{1, 2, 2\}$ in Einklang mit Definition 8.3 die (eindeutig definierte) Menge, die genau die 1 und die 2 enthält, nämlich $\{1, 2\}$.

Aufgabe 8.2.
Gibt es zwei Mengen M und N, sodass $M \subsetneq N$ und $N \subsetneq M$ gilt? Wenn ja, welche? Wenn nein, warum nicht?

Wie sieht es mit $M \subsetneq N$ und $N \subseteq M$ aus? Kannst du hierfür Beispiele finden?

Nun haben wir alle Werkzeuge, um unseren ersten Satz zu beweisen. Er besagt umgangssprachlich: Zwei Mengen sind identisch, wenn die erste in der zweiten und die zweite in der ersten enthalten ist. Wenn nicht, sind die Mengen auch nicht gleich.

Genau-dann-wenn-Aussagen

Hierbei handelt es sich um eine Genau-dann-wenn-Aussage: *Wenn* die erste in der zweiten und die zweite in der ersten enthalten ist, sind die Mengen gleich. Aber auch *immer wenn* zwei Mengen gleich sind, gelten die beiden Teilmengenbeziehungen. Gelten sie nicht, so können die Mengen nicht gleich sein. D.h. die Voraussetzung der beiden Teilmengenbeziehungen reicht nicht nur aus, um die Gleichheit der Mengen zu folgern, sondern sie ist auch notwendig dafür. Zum Beweis müssen wir zwei Schritte vollziehen: Wir müssen einmal aus den Teilmengenbeziehungen die Gleichheit folgern und einmal aus der Gleichheit die Teilmengenbeziehungen. Die Teilbeweise heben wir voneinander durch „$\Rightarrow$“ (für „*Wenn* ..., dann ...“) und „$\Leftarrow$“ (für „*Nur wenn* ..., dann ...“) ab.

Satz 8.5.
Seien M und N Mengen. M und N sind gleich, genau dann, wenn $M \subseteq N$ und $M \supseteq N$ gilt.

Beweis. Wir wollen eine Pauschalaussage über Mengen erhalten. D.h. wir dürfen uns nicht Mengen aussuchen, für die wir den Satz zeigen, sondern wir müssen ihn *für alle* Mengen M und N beweisen. Seien also M und N zwei *beliebige* Mengen.

Wir müssen beweisen, dass aus $M \subseteq N$ und $M \supseteq N$ die Gleichheit von M und N folgt *und* dass sich aus der Voraussetzung $M = N$ die beiden Teilmengenbeziehungen $M \subseteq N$ und $N \subseteq M$ ergeben.

„$\Leftarrow$“: Wir zeigen jetzt: Wenn $M \subseteq N$ und $M \supseteq N$, dann auch $M = N$.

Nach Definition 8.3 bedeutet $M \subseteq N$, dass jedes Element von M auch eines von N ist. Umgekehrt bedeutet $M \supseteq N$, dass $N \subseteq M$, d.h. jedes Element von N ist auch Element von M. Beides

zusammengenommen heißt, dass M und N genau dieselben Elemente enthalten und wiederum nach Definition 8.3 schreiben wir dafür $M = N$ und das wollten wir zeigen.

„$\Rightarrow$“: Wir müssen noch beweisen: Wenn $M = N$, dann gilt auch $M \subseteq N$ und $M \supseteq N$.

Wir können also annehmen, dass M und N gleich sind, nach Definition 8.3 enthalten sie also genau dieselben Elemente. Dann ist natürlich jedes Element von M auch eines von N, also $M \subseteq N$, denn so haben wir es in Definition 8.3 festgelegt. Umgekehrt ist auch $N \subseteq M$, denn jedes Element von N ist auch in M enthalten, die Mengen enthalten ja dieselben Elemente! Aber $N \subseteq M$ ist gleichbedeutend mit $M \supseteq N$ und damit haben wir beide Teilmengenbeziehungen gezeigt. □

direkter Beweis

Die beiden Beweisteile für Satz 8.5 sind Beispiele für *direkte Beweise*: Aus den Voraussetzungen und Definitionen wurde sozusagen auf direktem Wege durch korrektes, exaktes Argumentieren die Behauptung gezeigt. Im nächsten Beweis werden wir noch eine andere Technik kennenlernen. Der zugehörige Satz besagt umgangssprachlich: Wenn eine Menge in einer anderen, „größeren“ und diese in einer dritten enthalten ist, so ist die erste auch in der dritten enthalten – und nicht mit dieser identisch. Zum Beispiel ist die Menge der Frankfurter Einwohner eine echte Teilmenge der Einwohner Hessens und diese wiederum eine Teilmenge der Einwohner der BRD. Auch ohne geographische Kenntnisse kann man daraus direkt folgern, dass alle Frankfurter Einwohner auch in der BRD sesshaft sein müssen und es in der BRD aber noch Einwohner gibt, die nicht in Frankfurt leben – nämlich mindestens die Hessen, die nicht in Frankfurt leben. Und die muss es ja geben, sonst wäre die Menge aller Frankfurter keine *echte* Teilmenge der Hessen. Aber haben wir die Definitionen so gewählt, dass sie mit unserer Intuition übereinstimmen? Es könnte ja sein, dass Definition 8.3 unvollständig ist bzw. so unpassend gewählt wurde, dass die beschriebene Schlussfolgerung damit nicht möglich ist. Wir werden jetzt *beweisen*, dass dem nicht so ist.

Satz 8.6.
Seien L, M und N Mengen. Wenn $L \subsetneq M$ und $M \subseteq N$ gilt, dann gilt auch $L \subsetneq N$.

Beweis. Hier müssen wir wieder zweierlei zeigen: Nach Definition 8.3 bedeuetet $L \subsetneq N$, dass $L \subseteq N$ und $L \neq N$, d.h. L und N sind verschieden.

Wir wollen Satz 8.6 wiederum *für alle* Mengen L, M, N, die den Voraussetzungen des Satzes genügen, beweisen. Also seien L, M und N beliebige Mengen, für die $L \subsetneq M$ und $M \subseteq N$ gilt.

Um daraus $L \subseteq N$ zu folgern, wollen wir uns davon überzeugen, dass jedes Element von L auch eines von N ist. Sei also $x \in L$ beliebig. Wegen $L \subsetneq M$ wissen wir mit Definition 8.3, dass $L \subseteq M$, d.h. jedes Element von L ist auch eines von M. Folglich ist $x \in M$. Genauso können wir aber wegen $M \subseteq N$ folgern, dass $x \in N$. Weil x beliebig war, muss also für *jedes* $x \in L$ gelten, dass $x \in N$. D.h. jedes Element von L ist auch eines von N und das ist nach Definition 8.3 gleichbedeutend mit $L \subseteq N$.

Wie können wir jetzt noch $L \neq N$ beweisen? Wir verwenden einen *Beweis durch Widerspruch*. Angenommen, L und N wären gleich. Dann hätten wir mit den Voraussetzungen die Beziehungen $L \subseteq M$ und $M \subseteq N = L$, d.h. $L \subseteq M$ und $M \subseteq L$ und mit Satz 8.5 wären dann L und M gleich. Das ist aber ein Widerspruch zur Voraussetzung, dass $L \subsetneq M$, insbesondere $L \neq M$ ist. Also muss unsere Annahme falsch gewesen sein, L und N sind nicht gleich! Damit sind wir fertig, denn beide Ergebnisse zusammen beweisen $L \subsetneq N$. □

Beweis durch Widerspruch

Beweise durch Widerspruch wie den obigen benutzt man oft, wenn man keine Idee für einen direkten Beweis hat: Man geht vom Gegenteil des zu Zeigenden aus und leitet daraus dann unter Verwendung der Voraussetzungen, Definitionen und bekannter Resultate einen eindeutigen Widerspruch her. Wenn das gelingt, weiß man, dass die Annahme falsch gewesen sein muss, denn

alles andere am Beweis ist ja korrekt. Damit hat man also das Gewünschte bewiesen, denn sein Gegenteil hat sich als falsch erwiesen. Wir haben eben noch eine wichtige Technik für Mengenbeweise kennengelernt: Um Teilmengenbeziehungen zu zeigen, bietet es sich oft an, für ein *beliebiges* Element der einen Menge seine Zugehörigkeit zur anderen zu zeigen, anstatt nur die Definitionen zu erläutern.

Aufgabe 8.3.
Versuche, auf ähnliche Art für alle Mengen L, M, N die folgenden Behauptungen zu beweisen:

- *Aus $L \subseteq M$ und $M \subsetneq N$ folgt $L \subsetneq N$.*
- *Aus $L \supseteq M$ und $M \supseteq N$ folgt $L \supseteq N$.*

Aus zwei Mengen kann man auf verschiedene Weise eine neue gewinnen. Einige Möglichkeiten, wie dies geschehen kann, beschreibt die folgende Definition.

Definition 8.7.
Seien M und N Teilmengen einer gemeinsamen Obermenge U.

(a) Der *Schnitt* von M und N ist die Menge — Schnitt

$$M \cap N := \{x \mid x \in M \text{ und } x \in N\},$$

also die Menge aller Elemente, die sowohl in M als auch in N auftreten. Wir bezeichnen M und N als *disjunkt*, wenn $M \cap N = \emptyset$, d.h. wenn sie keine Elemente gemeinsam haben.

(b) Die *Vereinigung* von M und N ist die Menge — Vereinigung

$$M \cup N := \{x \mid x \in M \text{ oder } x \in N\},$$

also die Menge aller Elemente, die in mindestens einer der beiden Mengen auftreten.

(c) Die *Differenz* von M und N ist die Menge — Differenz

$$M \backslash N := \{x \mid x \in M \text{ und } x \notin N\},$$

also die Menge aller Elemente von M, die nicht in N auftreten.

(d) Die *symmetrische Differenz* von M und N ist die Menge — symm. Differenz

$$M \Delta N := (M \backslash N) \cup (N \backslash M),$$

also die Menge aller Elemente, die entweder in M oder in N, aber nicht in beiden gleichzeitig sind.

(e) Das *Komplement* von M (bzgl. U) ist die Menge — Komplement

$$\overline{M} := U \backslash M,$$

also die Menge aller Elemente aus U, die nicht in M sind.

(f) Die *Potenzmenge* von M ist die Menge — Potenzmenge

$$\mathcal{P}(M) := \{N \mid N \subseteq M\},$$

also die Menge aller Teilmengen von M.

Beachte, dass das Komplement einer Menge nur definiert ist, wenn die Obermenge, auf die wir uns beziehen, klar ist! Da wir es unmissverständlich auch immer mit Hilfe einer Differenz ausdrücken können, verzichten wir im Folgenden auf seine explizite Verwendung.

Die Elemente einer Potenzmenge sind wiederum Mengen. Weil die leere Menge Teilmenge jeder beliebigen Menge M ist, gilt immer $\emptyset \in \mathcal{P}(M)$. Auch $M \in \mathcal{P}(M)$ ist immer wahr.

Um die soeben definierten Begriffe zu verstehen, geben wir den Schnitt, die Vereinigung, die Differenz, die symmetrische Differenz und die Potenzmenge für Beispielmengen an.

Beispiel 8.8.
Seien $L := \{1, 2\}, M := \{2, 3\}, N := \{3, 4\}$.

- $L \cap M = \{2\}$, denn die 2 ist das einzige Element, was L und M gemein haben.
- $L \cup M = \{1, 2, 3\}$, denn jedes dieser Elemente kommt in mindestens einer der beiden Mengen vor.
- L und N sind disjunkt, denn sie haben kein Element gemein. Daraus folgt auch $L \backslash N = L$ und $N \backslash L = N$.
- $L \backslash M = \{1\}$, denn die 1 ist das einzige Element aus L, das in M nicht auftritt.
- $L \Delta M = \{1, 3\}$, denn die 1 und die 3 sind die einzigen Elemente, die in genau einer der beiden Mengen L und M auftreten (und nicht in beiden).
- $\mathcal{P}(L) = \big\{\emptyset, \{1\}, \{2\}, \{1, 2\}\big\}$, denn dies sind alle Teilmengen von L.

Das Konzept der Potenzmenge werden wir in einem späteren Abschnitt nochmal aufgreifen. Du kannst hier nochmal überprüfen, ob du es verstanden hast.

Aufgabe 8.4.
Du hast Salz, Pfeffer, Paprika und Basilikum im Haushalt. Auf wieviele verschiedene Arten kannst du deinen Salat würzen? Gib alle möglichen Kombinationen an.

Venn-Diagramm

Die Grafiken in Abbildung 8.1 veranschaulichen die in Definition 8.7 beschriebenen Mengen. Man nennt diese Art der Darstellung von Mengen *Venn-Diagramm*. Jeder Kreis symbolisiert die Menge, die mit dem Buchstaben in der Kreismitte gekennzeichnet ist. Die farbig hervorgehobene Fläche steht für die neu definierte Operation bzw. die Menge, die sich daraus ergibt.

Manchmal möchte man auch mehr als zwei Mengen vereinigen oder schneiden. Man kann sich beispielsweise den Schnitt von drei Mengen L, M, N als Schnitt des Schnitts von L und M mit N vorstellen. Genauso gut könnte man damit aber auch den Schnitt des Schnitts von M und N mit L meinen! Dass die resultierenden Mengen identisch sind, besagt die erste Behauptung in der nächsten Aufgabe. Deshalb können wir der Einfachheit halber bei Mehrfachvereinigungen und -schnitten die Klammern weglassen und definieren:

Notation 8.9.
Für drei Mengen L, M, N bezeichnen wir mit $L \cap M \cap N$ die Menge $(L \cap M) \cap N = L \cap (M \cap N)$ und mit $L \cup M \cup N$ die Menge $(L \cup M) \cup N = L \cup (M \cup N)$. Entsprechend gehen wir bei k Mengen $M_1, M_2, \ldots, M_k$ vor.

Schnitt $M \cap N$

M N

Vereinigung $M \cup N$

M N

Differenz $M \backslash N$

M N

symmetrische Differenz $M \Delta N$

M N

Abbildung 8.1.: Operationen auf zwei Mengen M und N

Wir geben noch weitere Resultate über Mengen an, darunter auch das bereits erwähnte, was uns die o.a. Notation erlaubt. Um die Gleichheit zweier Mengen M und N nachzuweisen, beweist man oft $M \subseteq N$ und $M \supseteq N$. Mit Definition 8.3 und Satz 8.5 folgt daraus nämlich direkt $M = N$. Nutze diesen Hinweis in den Beweisen der unten angegebenen Behauptungen.

Aufgabe 8.5.
Überzeuge dich anhand der Venn-Diagramme aus Abb. 8.1 von der Richtigkeit der folgenden Aussagen für alle Mengen L, M, N. Beweise sie dann mit Hilfe der Techniken, die du bisher erlernt hast.

- $M \cap M = M$
- $M \cup M = M$
- $M \cap N = N \cap M$ *(Kommutativgesetz des Schnitts)*
- $M \cup N = N \cup M$ *(Kommutativgesetz der Vereinigung)*
- $L \cap (M \cap N) = (L \cap M) \cap N$ *(Assoziativgesetz des Schnitts)*
- $L \cup (M \cup N) = (L \cup M) \cup N$ *(Assoziativgesetz der Vereinigung)*
- $L \cap (M \cup N) = (L \cap M) \cup (L \cap N)$ *(1. Distributivgesetz)*
- $L \cup (M \cap N) = (L \cup M) \cap (L \cup N)$ *(2. Distributivgesetz)*
- $M = N$ *gilt genau dann, wenn* $M \Delta N = \emptyset$ *ist.*
- $M \backslash N$ *und* $N \backslash M$ *sind disjunkt.*
- $M \cup N = (M \backslash N) \cup (N \backslash M) \cup (M \cap N)$ *und je zwei der drei Mengen, die hier vereinigt werden, sind disjunkt.*

Ein weiteres wichtiges Konzept in der Mengenlehre ist die *Mächtigkeit* einer Menge. Sie besagt, „wie groß“ diese ist.

Definition 8.10.
endliche Menge
Sei M eine Menge. Wir bezeichnen M als *endlich* , wenn es ein $k \in \mathbb{N}$ gibt, sodass M genau k Elemente hat.

Mächtigkeit
Die Zahl k ist die *Mächtigkeit* oder *Kardinalität* (in Zeichen $|M|$) von M. Hat M unendlich viele Elemente, so ist seine Mächtigkeit ebenfalls unendlich. Wir schreiben dann $|M| = \infty$ (lies: $\infty =$ „unendlich").

Endliche Mengen werden meistens explizit angegeben, Mengen, deren Mächtigkeit unendlich ist, hingegen mit Pünktchen (vgl. $\mathbb{Z}$ in Beispiel 8.2) oder implizit durch eine Umschreibung.

Beispiel 8.11.

- $|\emptyset| = 0$, da die leere Menge keine Elemente hat.
- $|\{\emptyset\}| = 1$, da diese Menge ein Element, nämlich die leere Menge, hat.
- $|\{\text{er, sie, es}\}| = 3$
- $|\{\text{Haus}, 6000, \text{weiß}\}| = 3$
- $|\{1,1,1,1,2\}| = 2$, da $\{1,1,1,1,2\} = \{1,2\}$.
- $|\{\{a,b\},c\}| = 2$, da $\{a,b\}$ hier ein *Element* der „äußeren" Menge ist.
- $|\mathbb{R}| = \infty$, da es unendlich viele reelle Zahlen gibt.

Um dich an den neuen Begriff zu gewöhnen und als Einstimmung auf die folgenden Sätze und Beweise versuche dich doch mal an diesen Übungen:

Aufgabe 8.6.
Bestimme für jede Menge aus Beispiel 8.2 ihre Mächtigkeit.

Wie müssen zwei Mengen aussehen, deren Vereinigung die Mächtigkeit 0 *hat?*

Was kannst du über die Mächtigkeiten zweier Mengen sagen, deren Schnitt die Mächtigkeit ∞ *hat?*

Alle Behauptungen, die wir hier als *Satz* vorstellen, sind wahr und wurden bereits von jemandem bewiesen. Es gibt aber auch Aussagen, für die bisher noch kein Beweis gefunden wurde, es ist offen, ob sie gelten. Außerdem existieren natürlich Behauptungen, die falsch sind. Wie beweist man, dass eine Behauptung nicht korrekt ist? Man kann zum Beispiel versuchen, das Gegenteil mit einem Beweis zu zeigen. Wenn die Behauptung die Form „Für alle ... gilt ..." hat, ist es noch einfacher: Man muss zeigen, dass die Aussage *nicht* für alle ... gilt. Dafür reicht es, ein Gegenbeispiel zu finden, für das die Aussage nicht wahr ist.

Beispiel 8.12.
Seien M und N zwei Mengen mit $|M| = |N| = \infty$.

Behauptung: $|M \cap N| = \infty$, d.h. der Schnitt zweier Mengen mit Mächtigkeit ∞ hat immer auch Mächtigkeit ∞.

Um diese Behauptung zu widerlegen, müssen wir zwei Mengen M und N finden, die jeweils unendlich viele Elemente enthalten, deren Schnitt aber endlich ist. Setze zum Beispiel

$$M := \{x \mid \text{Es gibt ein } k \in \mathbb{Z} \text{ mit } x = 2 \cdot k\},$$
$$N := \{x \mid \text{Es gibt ein } k \in \mathbb{Z} \text{ mit } x = 2 \cdot k + 1\}.$$

M ist die Menge aller geraden Zahlen, N die Menge aller ungeraden Zahlen. Jede der beiden enthält offensichtlich unendlich viele Elemente, erfüllt also die Voraussetzungen der Behauptung. Trotzdem ist

$$M \cap N = \emptyset$$

und daher

$$|M \cap N| = |\emptyset| = 0 < \infty.$$

Wir haben also zwei Mengen gefunden, über die die Behauptung die Aussage macht, ihr Schnitt habe Mächtigkeit ∞, obwohl die Mächtigkeit 0 ist. Folglich stimmt die Behauptung nicht.

Vorsicht! Im Beispiel oben zeigen wir nicht, dass der Schnitt von zwei Mengen mit Mächtigkeit ∞ *immer* endliche Kardinalität hat. Für manche Mengen, z.B. im Fall $M := \mathbb{N}, N := \mathbb{N}$, ergibt sich ein Schnitt mit Mächtigkeit ∞. Trotzdem haben wir bewiesen, dass nicht die Behauptung, sondern ihr Gegenteil stimmt: *Es gibt* zwei Mengen M und N mit Mächtigkeit ∞, deren Schnitt endlich ist.

Aufgabe 8.7.
Seien M und N zwei beliebige Mengen. Widerlege die folgenden Behauptungen:

- $|\mathcal{P}(M)| \geq 2$
- $|M \cup N| = |M| + |N|$
- $|M \cap N| \geq |M \cup N|$
- $|M \cap N| < |M \cup N|$
- *Aus $M \subseteq N$ folgt $|M| < |N|$.*
- *Aus $M \subsetneq N$ folgt $|M| < |N|$. (Hinweis: Untersuche Mengen mit Mächtigkeit ∞.)*

Die zweite falsche Behauptung aus Aufgabe 8.7 können wir zu einer korrekten Aussage „reparieren“. Es handelt sich wieder um eine Genau-dann-wenn-Aussage, der Beweis besteht daher wie auch der von Satz 8.5 aus zwei Teilen.

Satz 8.13.
Seien M und N endliche Mengen. Dann gilt $|M \cup N| = |M| + |N|$ genau dann, wenn M und N disjunkt sind.

Beweis. Seien M und N Mengen mit endlichen Mächtigkeiten $k_M < \infty$ und $k_N < \infty$.

„$\Leftarrow$“: Wir zeigen zunächst: Wenn M und N disjunkt sind, dann folgt $|M \cup N| = |M| + |N|$.

Angenommen, M und N sind disjunkt.

Jedes x, das in die linke oder rechte Seite der zu beweisenden Gleichung eingeht, liegt in einer der drei Mengen $M \cap N, M \backslash N, N \backslash M$. Nach Definition 8.7 gilt $M \cap N = \emptyset$, denn M und N sind disjunkt. Es gibt daher kein $x \in M \cap N$. Demnach liegt jedes x, das in eine der beiden Seiten der zu zeigenden Gleichungen eingeht, in $M \backslash N$ oder in $N \backslash M$.

Jedes x, das *entweder* Element von M *oder* von N ist, d.h. jedes $x \in (M \backslash N) \cup (N \backslash M)$, trägt zu $|M \cup N|$ genau 1 bei, da es nach Definition 8.3 auch Element von $M \cup N$ ist. Es erhöht aber auch den Wert von $|M| + |N|$ um 1, denn es tritt in genau einer der beiden Mengen M und N auf.

Fallunterscheidung

Was haben wir durch diese *Fallunterscheidung* gezeigt? Jedes Element jeder der drei Mengen, die in der zu beweisenden Gleichung auftreten, trägt zur linken Seite gleich viel wie zur rechten

Seite bei. D.h. für die Mächtigkeit von $M \cup N$ ergibt sich derselbe Wert wie für die Summe der Mächtigkeiten von M und N, was wir zeigen wollten.

„$\Rightarrow$“: Nun zeigen wir: Wenn $|M \cup N| = |M| + |N|$ gilt, dann müssen M und N disjunkt sein.

Mit dem letzten Resultat aus Aufgabe 8.5 können wir $M \cup N$ als $(M \backslash N) \cup (N \backslash M) \cup (M \cap N)$ schreiben, wobei alle drei vereinigten Mengen paarweise, d.h. je zwei von ihnen, disjunkt sind. Daher gilt auch

$$|M \cup N| = |(M \backslash N) \cup (N \backslash M) \cup (M \cap N)|.$$

Nur die Elemente, die in mindestens einer der beiden Mengen M und N sind, tragen links oder rechts etwas bei, da alle Mengen, die in der Gleichung auftauchen, aus Operationen ausschließlich auf M und N entstehen. D.h. wir müssen nur betrachten, was die Elemente $x \in M \cup N$ beitragen. Links tragen sie jeweils genau 1 bei, weil sie alle Element der Menge $M \cup N$ sind und in einer Menge jedes Element nur einmal zählt. Für den Beitrag rechts benutzen wir wieder eine Fallunterscheidung:

- Gilt $x \in M \backslash N$, so trägt x in der Summe $|M| + |N|$ auch nur 1 bei, weil es in M, aber nicht in N ist und somit nur zum ersten Summanden beiträgt.
- Gilt $x \in N \backslash M$, so trägt x in der Summe $|M| + |N|$ auch nur 1 bei, weil es in N, aber nicht in M ist und somit nur zum zweiten Summanden beiträgt.
- Gilt $x \in M \cap N$, so trägt x in der Summe $|M| + |N|$ jedoch 2 bei, weil es in M und in N ist und somit zu beiden Summanden je 1 beiträgt.

Das heißt, alle Elemente aus $M \cup N$ tragen links und rechts in der Gleichung $|M \cup N| = |M| + |N|$ gleich viel, nämlich 1 bei – außer die $x \in M \cap N$, sie tragen zur Kardinalität links 1, aber rechts 2 bei. Damit die Gleichung trotzdem gilt (und das setzen wir ja in diesem Teil des Beweises voraus), darf es also keine $x \in M \cap N$ geben: M und N müssen disjunkt sein und das war zu zeigen.

Damit haben wir beide Implikationen bewiesen und sind fertig. □

Im obigen Beweis haben wir eine weitere wichtige Technik benutzt: Fallunterscheidungen. Wenn wir nicht überblicken, wie wir mit einem direkten Beweis die Aussage folgern können, bieten sie sich oft an. Wir vereinfachen die aktuelle Situation im Beweis, indem wir mehrere Möglichkeiten separat voneinander behandeln. In jedem davon führen wir den Beweis weiter oder im Idealfall zu Ende, dann müssen wir nur noch erklären, weshalb keine anderen Fälle als die behandelten auftreten können.

Nun kannst du dich selbst an ähnlichen Beweisen versuchen.

Aufgabe 8.8.
Seien M und N beliebige Mengen. Zeige:

- $|M \cup N| \leq |M| + |N|$
- $|M \cap N| \leq \min\{|M|, |N|\}$ *($\min\{a, b\}$ ist die kleinere der beiden Zahlen a und b.)*

  Tipp: Zeige $|M \cap N| \leq |M|$ und $|M \cap N| \leq |N|$ und folgere daraus die Behauptung.

Binomialkoeffizient

Der Binomialkoeffizient $\binom{n}{k}$ *ist die Anzahl der Teilmengen mit Mächtigkeit k einer Menge mit Mächtigkeit n. Beweise die folgende Formel zur Berechnung von $\binom{n}{k}$:*

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

Hinweis: Für eine Menge M mit n Elementen markiere eines davon. Unterscheide zwischen den Teilmengen, die das markierte Element enthalten, und denen, die es nicht enthalten. Wie kannst du hier Satz 8.13 anwenden?

Inzwischen kennst du dich mit Mengen schon recht gut aus. In den kommenden beiden Kapiteln benutzen wir die hier vorgestellten Begriffe, um uns mit Relationen und Funktionen zu beschäftigen. Diesen wirst du im Zusammenhang mit Mengen, insbesondere mit $\mathbb{N}$ und $\mathbb{R}$, im Laufe deines Studiums häufig begegnen.

9. Relationen

Wenn uns eine Menge vorliegt, spielt die Reihenfolge ihrer Elemente keine Rolle. Das heißt, zwei Mengen sind gleich, wenn sie dieselben Elemente enthalten, auch wenn diese nicht in derselben Reihenfolge notiert sind. Das haben wir in Definition 8.3 so festgelegt. Manchmal wollen wir aber verschiedene Reihenfolgen von Objekten unterscheiden. Zum Beispiel müssen bei einem Seminar im Laufe des Semesters alle Teilnehmer einen Vortrag halten. Zur Planung der Vorbereitung ist es entscheidend, zu wissen, ob man den ersten oder den letzten Vortrag halten muss! Auch in einer Warteschlange an der Kinokasse bekommen im Normalfall alle ihre Tickets, aber der/die Erste wird kaum seinen Platz mit jemand anderem tauschen wollen, denn er/sie hat noch freie Platzwahl.

Definition 9.1.
Seien $m, n \in \mathbb{N}$.

(a) Seien $a_1, a_2, \ldots, a_n$ beliebige Objekte. Wir nennen $(a_1, a_2, \ldots, a_n)$ das *geordnete Tupel* mit den *Komponenten* $a_1, a_2, \ldots, a_n$. Dabei ist a_i die *i-te Komponente* und n die *Länge* des Tupels. (Tupel)

(b) Ein Tupel der Länge 2 wird auch als *geordnetes Paar* bezeichnet, ein Tupel der Länge 3 als *Tripel*, ein Tupel der Länge 4 als *Quadrupel*, usw. Tupel der Länge n nennen wir kurz n-Tupel. (geordnetes Paar)

(c) Zwei Tupel $(a_1, a_2, \ldots, a_m)$ und $(b_1, b_2, \ldots, b_n)$ sind *gleich*, wenn $m = n$ und für alle $1 \leq i \leq n$ ihre i-ten Komponenten gleich sind, d.h. $a_i = b_i$ gilt. (Gleichheit)

Nach dieser Definition gelten zwei Tupel gleicher Länge nur dann als identisch, wenn sie an jeder Position, d.h. in jeder Komponente, übereinstimmen. Deshalb ist es an dieser Stelle auch nicht sinnvoll, zu definieren, wann wir ein Tupel als „Teiltupel" eines anderen verstehen wollen, was bei Mengen sehr wohl nützlich war. In einem Tupel können Objekte auch mehrfach auftreten, sie gelten aufgrund ihrer unterschiedlichen Positionen als verschieden.

Beachte: Tupel werden mit runden Klammern gekennzeichnet, Mengen hingegen schreibt man mit geschweiften Klammern!

Beispiel 9.2.

- Das leere Tupel () hat Länge 0.
- Es ist $(1, 2) \neq (1, 2, 1) \neq (1, 2, 2)$, aber $\{1, 2, 1\} = \{1, 2, 2\} = \{1, 2\}$.
- (Anna, Peter) ist ein geordnetes Paar, {Anna, Peter} ist eine zweielementige Menge, manchmal auch *Paar* genannt.
- Es gilt $(a_1, a_2, \ldots, a_n) \neq \{a_1, a_2, \ldots, a_n\}$ für beliebige Objekte $a_1, a_2, \ldots, a_n$.

Auch in diesem Skript ist die Reihenfolge entscheidend: Würden wir erst den Satz und nachträglich die Definition vorstellen, so könnte den Satz niemand verstehen und erst recht nicht beweisen!

Aufgabe 9.1.
Überlege dir weitere Beispiele für Alltagssituationen und Objekte, die man besser als Komponenten eines Tupes als in Form von Mengen erfassen sollte, weil ihre Reihenfolge wichtig ist. Gib auch Beispiele an, in denen nur die Zugehörigkeit zu einer Menge von Interesse ist.

Wir stellen jetzt eine neue Mengenoperation vor. Das Ergebnis ist eine Menge von Tupeln.

Definition 9.3.
kartesisches Produkt
Sei $n \in \mathbb{N}$ und seien $M_1, M_2, \ldots, M_n$ Mengen. Als *kartesisches Produkt* von $M_1, M_2, \ldots, M_n$ bezeichnen wir

$$M_1 \times M_2 \times \ldots \times M_n := \big\{(a_1, a_2, \ldots, a_n) \mid a_i \in M_i \text{ für alle } i \in \{1, 2, \ldots, n\}\big\},$$

also die Menge aller Tupel, bei denen die erste Komponente ein Element von M_1, die zweite eines von M_2 ist usw.

Wenn alle Mengen, von denen wir das kartesische Produkt bilden wollen, identisch sind, können wir dieses kürzer ausdrücken:

Notation 9.4.
Falls $M := M_1 = M_2 = \ldots = M_n$, so schreiben wir M^n anstelle von $M_1 \times M_2 \times \ldots \times M_n$.

Kartesischen Produkten begegnen wir im Alltag immer wieder. Wir zählen hier einige Beispiele auf, vielleicht fallen dir ja auch noch welche ein?

Beispiel 9.5.

- Das kartesische Produkt der Menge W aller weiblichen und der Menge M aller männlichen Besucher eines Tanzkurses ist $W \times M$, die Menge aller Tanzpaare der Form (Frau, Mann).
- Die Menge aller Uhrzeiten im Format SS:MM ist $\{00, 01, \ldots, 23\} \times \{00, 01, \ldots, 59\}$.
- Die Menge aller dreigängigen Menüs, die eine Speisekarte bietet, ist $V \times H \times N$, wobei V die Menge aller Vorspeisen, H die Menge aller Hauptspeisen, N die Menge aller Nachspeisen ist.
- Die reelle Zahlenebene ist $\mathbb{R}^2 = \mathbb{R} \times \mathbb{R}$. Menge aller geographischen Koordinaten ist $L \times B = \{0, 1, \ldots, 59\} \times \{0, 1, \ldots, 59\}$, wobei L die Menge aller Längen- und B die Menge aller Breitengrade ist.

Aufgabe 9.2.
Gib die folgenden kartesischen Produkte in expliziter Schreibweise sowie ihre Mächtigkeit an.

- $\{0, 1\} \times \{0, 1\}$
- M^0 *und* M^1 *für eine beliebige Menge* M
- $M \times \emptyset$ *für eine beliebige Menge* M
- $\{$*rot, gelb, grün*$\} \times \{$*Ball, Kleid, Auto*$\}$

Wie lässt sich die Menge aller Spielkarten eines Skatblatts als kartesisches Produkt zweier Mengen auffassen? Welche Mächtigkeiten haben diese?

Nach dem Lösen der obigen Aufgabe hast du wahrscheinlich schon eine Vermutung, wie sich die Mächtigkeit des kartesischen Produkts $M_1 \times M_2 \times \ldots \times M_n$ aus denen der M_i ergibt. Vielleicht kennst du auch das folgende Resultat noch aus dem Stochastikunterricht in der Schule.

Satz 9.6.
Sei $n \in \mathbb{N}$, seien $M_1, M_2, \ldots, M_n$ Mengen. Dann gilt

$$|M_1 \times M_2 \times \ldots \times M_n| = |M_1| \cdot |M_2| \cdot \ldots \cdot |M_n|.$$

Beweis. Am besten kann man den Satz mittels *vollständiger Induktion* zeigen. Da wir diese noch nicht thematisiert haben und es eine anfangs kompliziert erscheinende Beweistechnik ist, deuten wir sie hier nur an.

Um ein Element des kartesischen Produkts eindeutig zu identifizieren, müssen alle seine Komponenten festgelegt sein. Für die erste Komponente können wir jedes Element aus M_1 nehmen, also haben wir hier $|M_1|$ Optionen. Das so definierte 1-Tupel können wir für die Konstruktion eines 2-Tupels benutzen, wobei wir als zweite Komponente nur Elemente aus $|M_2|$ erlauben. Jedes 1-Tupel aus dem ersten Schritt liefert für ein festes $m \in M_2$ ein anderes 2-Tupel, d.h. jedes $m \in M_2$ erzeugt $|M_1|$ neue 2-Tupel. Also ist die Anzahl der 2-Tupel, deren erste Komponente aus M_1 und deren zweite Komponente aus M_2 stammt, $|M_1| \cdot |M_2|$.

Analog ergibt sich als Anzahl der 3-Tupel, bei denen für alle i die i-te Komponente aus M_i stammt, $|M_1| \cdot |M_2| \cdot |M_3|$ und als Anzahl der n-Tupel mit dieser Eigenschaft

$$|M_1| \cdot |M_2| \cdot \ldots \cdot |M_n|$$

und das war zu zeigen. □

Die im Beweis erwähnte *vollständige Induktion* ist eine beliebte Beweistechnik, wenn man eine Behauptung für alle natürlichen Zahlen n zeigen möchte. Sobald du sie in Kapitel 12 kennen gelernt hast, kannst du den obigen Beweis eleganter formulieren!

vollständige Induktion

Den zugehörigen Satz kannst du dir in der nächsten Aufgabe gleich zunutze machen.

Aufgabe 9.3.
Überprüfe deine Antworten zu Aufgabe 9.2, indem du mit Hilfe von Satz 9.6 *die Kardinalitäten der beschriebenen Mengen ausrechnest.*

Auch wenn es, wie vorhin bemerkt, kompliziert wäre, das Teilmengensymbol $\subseteq$ sinnvoll für Tupel zu definieren, können wir es im Kontext des kartesischen Produkts wieder ins Spiel bringen – denn das kartesische Produkt ist ja eine *Menge* von Tupeln.

Definition 9.7.
Sei $n \in \mathbb{N}$ und seien $M_1, M_2, \ldots, M_n$ Mengen. Eine Teilmenge R des kartesischen Produkts $M_1 \times M_2 \times \ldots \times M_n$ heißt *Relation* von $M_1, M_2, \ldots, M_n$ mit *Stelligkeit n*.

Relation
Stelligkeit

Nach der obigen Definition ist jedes kartesische Produkt $M_1 \times M_2 \times \ldots \times M_n$ selbst eine (n-stellige) Relation von $M_1, M_2, \ldots, M_n$.

Zwei Relationen sind, da sie beide Mengen sind, genau dann gleich, wenn sie dieselben Tupel enthalten. (Beachte hierbei Definition 9.1.)

Beispiel 9.8.

- Sei M das deutsche Alphabet, als Menge aufgefasst. Wenn wir jedes Wort als Tupel seiner Buchstaben interpretieren, ist die Menge aller dreibuchstabigen deutschen Wörter eine Relation von M, M, M.
- Für die in Beispiel 9.5 definierte Menge $W \times M$ ist die Menge aller tatsächlich gebildeten Tanzpaare für den Abschlussball eine Relation von W und M, denn jede(r) TeilnehmerIn des Kurses wird nur mit einem/einer einzigen TanzpartnerIn zum Ball gehen.
- Die Menge aller bereits bestellten dreigängigen Menüs ist eine Relation der Mengen V, H und N, die in Beispiel 9.5 definiert wurden.
- Die Menge $\{(x, y) \mid x = |y|\}$, bei deren Elementen die erste Komponente der Betrag der zweiten ist, stellt eine Relation von $\mathbb{R}$ und $\mathbb{R}$ dar.

Du findest bestimmt noch zahlreiche weitere Beispiele für Relationen, insbesondere in der Mathematik tauchen sie oft auf.

Aufgabe 9.4.
Gib weitere zweistellige Relationen von $\mathbb{R}$ und $\mathbb{R}$ an. Findest du auch welche von $\mathbb{N}$ und $\mathbb{R}$?

Wie kannst du die dir aus der Schule bekannten Funktionsgraphen als Relationen auffassen? Welche Stelligkeit haben sie?

Die Antwort auf die letzten beiden Fragen in Aufgabe 9.4 liefern wir im nächsten Abschnitt, dort thematisieren wir Funktionen als spezielle Relationen.

10. Funktionen

Mit Funktionen hatten wir alle in der Schule schon zu tun. Eine Funktion ist, ausgedrückt mit den Begriffen, die wir neu kennen gelernt haben, eine Relation von zwei Mengen, bei der jedes Element der ersten Menge in genau einem Tupel der Relation als erste Komponente auftritt. D.h. zu jedem Element der ersten Menge gibt es *genau ein* „zugehöriges“ der zweiten Menge.

Definition 10.1.
Seien X und Y Mengen.

Funktion

(*a*) Eine Relation $f \subseteq X \times Y$, bei der es zu jedem $x \in X$ genau ein $y \in Y$ mit $(x, y) \in f$ gibt, nennen wir *Funktion von X nach Y* (in Zeichen: $f : X \to Y$). Die Menge X heißt *Definitionsbereich* von f und die Menge Y *Bildbereich* von f.

(*b*) Sei f eine Funktion von X nach Y. Für jedes $x \in X$ bezeichnen wir mit $f(x)$ das eindeutige $y \in Y$, für das $(x, y) \in f$ gilt. Ist Y eine Menge von Zahlen, so sprechen wir statt von $f(x)$ auch vom *Funktionswert* von x.

Bild

(*c*) Sei f eine Funktion von X nach Y. Wir nennen die Menge

$$f(X) := \{y \in Y \mid \text{ Es gibt ein } x \in X \text{ mit } f(x) = y\}$$

das *Bild* von f.

Bei einer Funktion von X nach Y gibt es also zu jedem $x \in X$ genau ein zugehöriges $y \in Y$. Dabei sind der Definitionsbereich X und der Bildbereich Y nicht zwangsläufig Mengen von Zahlen, ebensowenig wie bei einer Relation.

Aufgabe 10.1.
Bei welchen der folgenden Relationen handelt es sich um Funktionen? Kannst du deine Behauptung beweisen? Versuche, für jede Funktion auch ihr Bild anzugeben.

- *Die geordneten Paare der Form (Personalausweisnummer, Einwohner Deutschlands über* 16*) als Teilmenge des kartesischen Produkts von* $\mathbb{N}$ *mit den Einwohnern Deutschlands*
- *Die geordneten Paare der Form (Adresse, Einwohner Deutschlands) als Teilmenge des kartesischen Produkts der Menge aller Adressen in Deutschland und der Einwohner Deutschlands*
- *Die Menge aller tatsächlich gebildeten Tanzpaare der Form (Frau, Mann) als Teilmenge des kartesischen Produkts der weiblichen und der männlichen Tanzkursteilnehmer (vorausgesetzt, jede Frau findet einen Tanzpartner)*
- $M \times N \subseteq M \times N$ *für beliebige Mengen* M *und* N *mit* $|N| \leq 1$
- $\mathbb{N} \times \mathbb{N} \subseteq \mathbb{R} \times \mathbb{R}$
- $\{(x, x) \in \mathbb{R}^2\}$
- $\{(x, y) \in \mathbb{R}^2 \mid y = 5x\}$
- $\{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 = 1\}$

- $\{(x, y) \in \mathbb{Z}^2 \mid x \textit{ ist Teiler von } y\}$
- $\{(x, y) \in \mathbb{Z}^2 \mid y \textit{ ist Teiler von } x\}$
- $\{(x, y) \in \mathbb{Z}^2 \mid x \textit{ ist eine Primzahl und } y \textit{ ist Teiler von } x\}$

Wann sind eigentlich zwei Funktionen „gleich“? Wenn wir sie als zweistellige Relationen auffassen, können wir mit den bereits vorgestellten Definitionen die Aussage der nächsten Aufgabe zeigen.

Aufgabe 10.2.
Seien f und g zwei Funktionen mit demselben Definitionsbereich X und demselben Bildbereich Y. Zeige: f und g sind genau dann gleich, wenn für alle $x \in X$ gilt: $f(x) = g(x)$. Fasse hierzu f und g als zweistellige Relationen auf.

Oft werden Funktionen aber auch als eindeutige Zuordnungen eingeführt. Dann wird die Menge, die wir hier mit f identifiziert haben, meist als *Graph von f* bezeichnet.

Notation 10.2.
Funktionen können in Mengenschreibweise mit geschweiften Klammern angegeben werden. Alternativ kann man eine Funktionsvorschrift der Form $f : x \mapsto y$ oder eine Funktionsgleichung der Form $f(x) := y$ (die Definitionsdoppelpunkte werden häufig weggelassen) verwenden, um $(x, y) \in f$ auszudrücken.

Im folgenden Beispiel geben wir zum Verständnis alle Notationsarten an, um die Funktion von $\mathbb{N}$ nach $\mathbb{N}$ zu beschreiben, die jeder natürlichen Zahl ihr Doppeltes zuweist.

Beispiel 10.3.
Die Varianten

- $f_1 = \{(0, 0), (1, 2), (2, 4), (4, 8), \dots\} \subseteq \mathbb{N} \times \mathbb{N}$
- $f_2 = \{(x, y) \in \mathbb{N}^2 \mid y = 2x\}$
- $f_3 : \mathbb{N} \to \mathbb{N}, f(n) := 2n$
- $f_4 : \mathbb{N} \to \mathbb{N}, f : n \mapsto 2n$

beschreiben allesamt dieselbe Funktion. Hierbei sind die Darstellungen von f_2, f_3 und f_4 der von f_1 vorzuziehen, weil sie unmissverständlicher sind. Dies gilt immer, wenn die Mächtigkeit der Funktion unendlich ist (vgl. hierzu den Kommentar nach Definition 8.10).

Wenn zwei Funktionen f und g nicht denselben Definitionsbereich X haben, können sie auch nicht gleich sein:

Satz 10.4.
Seien $f : X_f \to Y_f$ und $g : X_g \to Y_g$ Funktionen.

Falls $X_f \neq X_g$, so sind f und g nicht gleich im Sinne von Definition 8.3.

Beweis. Seien f und g wie in den Voraussetzungen des Satzes beschrieben und gelte $X_f \neq X_g$. Dann gibt es nach der 9. Teilübung aus Aufgabe 8.5 ein Element in $X_f \backslash X_g$ oder eines in $X_g \backslash X_f$, denn $X_f \Delta X_g \neq \emptyset$.

Wenn es ein Element $\tilde{x} \in X_f \backslash X_g$ gibt, so ist

$$\big(\tilde{x}, f(\tilde{x})\big) \in f,$$

da es nach Definiton 10.1 zu *jedem* $x \in X_f$ ein $y \in Y_f$ mit $(x, y) \in f$ gibt. Aber

$$\big(\tilde{x}, f(\tilde{x})\big) \notin g,$$

denn $x \notin X_g$ und $g \subseteq X_g \times Y_g$.

Analog ergibt sich für $\tilde{x} \in X_g \backslash X_f$, dass $\big(\tilde{x}, g(\tilde{x})\big) \in g$, doch $\big(\tilde{x}, g(\tilde{x})\big) \notin f$ ist.

Also ist $f \Delta g \neq \emptyset$ und mit Aufgabe 8.5 (9. Teilübung) können dann f und g nicht gleich sein. □

Wenn wir zwei Funktionen als gleich ansehen, sollen sich diese auch genau gleich verhalten bzw. die gleichen Eigenschaften haben. Mit Hinblick auf die Begriffe „injektiv“, „surjektiv“ und „bijektiv“, die wir für Funktionen einführen werden, ist es sinnvoll, zusätzlich zum selben Definitionsbereich die Gleichheit der Bildbereiche zu fordern. Das führt uns zur folgenden Definition:

Definition 10.5.
Seien X und Y Mengen, seien f und g Funktionen mit dem gleichen Definitionsbereich X und dem gleichen Bildbereich Y.

(a) Wir bezeichnen f und g als *gleich* (in Zeichen: $f \equiv g$), wenn für alle $x \in X$ gilt: $f(x) = g(x)$. Gleichheit

(b) Seien $X, Y \subseteq \mathbb{R}$. Die Funktion f ist *kleiner als oder gleich* g (in Zeichen $f \leq g$), wenn für alle $x \in X$ gilt: $f(x) \leq g(x)$.

(c) Seien $X, Y \subseteq \mathbb{R}$. Die Funktion f ist *kleiner als* g (in Zeichen $f < g$), wenn für alle $x \in X$ gilt: $f(x) < g(x)$.

(d) Seien $X, Y \subseteq \mathbb{R}$. Die Funktion f ist *größer als oder gleich* g (in Zeichen $f \geq g$), wenn für alle $x \in X$ gilt: $f(x) \geq g(x)$.

(e) Seien $X, Y \subseteq \mathbb{R}$. Die Funktion f ist *größer als* g (in Zeichen $f > g$), wenn für alle $x \in X$ gilt: $f(x) > g(x)$.

Eine Funktion f ist also kleiner als eine Funktion g, wenn der Graph von f „unterhalb“ des Graphen von g verläuft – und sowohl ihre Definitions- als auch ihre Bildbereich übereinstimmen! Denn nur dann lassen sich nach unserer Definition die beiden Funktionen überhaupt vergleichen. Aus $f < g$ folgt, unserer Intuition entsprechend, direkt $f \leq g$, genauso verhält es sich mit $f > g$ und $f \geq g$.

Haben die Graphen der Funktionen Schnittpunkte, die nicht bloß Berührpunkte sind, so gilt weder $f < g$ noch $f > g$ (noch $f \leq g$ noch $f \geq g$ noch $f \equiv g$).

Aufgabe 10.3.
Gib für je zwei der folgenden Funktionen an, ob sie vergleichbar sind, und falls ja, ob $\equiv, \leq, <, \geq, >$ *gilt. Gelingt es dir, ohne die Graphen zu zeichnen?*

- $f_0 : \mathbb{N} \to \mathbb{N},\ f_0(n) := n$
- $f_1 : \mathbb{N} \to \mathbb{Z},\ f_1(n) := n$

- $f_2 : \mathbb{Z} \to \mathbb{Z},\ f_2(n) := -1$
- $f_3 : \mathbb{Z} \to \mathbb{Z},\ f_3(n) := n - 3$
- $f_4 : \mathbb{N} \to \mathbb{N},\ f_4(n) := 4n$
- $f_5 : \mathbb{Z} \to \mathbb{Z},\ f_5(n) := 4n$
- $f_6 : \mathbb{N} \to \mathbb{N},\ f_6(n) := n^3$
- $f_7 : \mathbb{Z} \to \mathbb{Z},\ f_7(n) := n^3$
- $f_8 : \mathbb{N} \to \mathbb{N},\ f_8(n) := 2^n$
- $f_9 : \mathbb{Z} \to \mathbb{R},\ f_9(n) := 2^n$

Definition 10.5(d) kann man mit Hilfe der Definition für $f \leq g$ kürzer ausdrücken. Weißt du, wie? Versuche, auch Definition 10.5(e) kürzer zu fassen.

Zeige: Für zwei Funktionen mit demselben Definitions- und demselben Bildbereich, die beide Teilmengen von $\mathbb{R}$ sind, gilt $f \equiv g$ genau dann, wenn $f \leq g$ und $g \leq f$ gilt.

Du hast bestimmt herausgefunden, dass f_0 und f_1 in der obigen Aufgabe nicht gleich sind. Dennoch nehmen sie auf ihrem gemeinsamen Definitionsbereich die gleichen Werte an, bloß dass der Bildbereich von f_1 „größer als nötig" gewählt wurde: Er beinhaltet auch Zahlen, die gar nicht als Funktionswerte auftreten. Dies drückt man auch mit der Aussage „f_1 ist nicht surjektiv"aus. Denn Begriff der Surjektivität sowie zwei weitere wollen wir jetzt exakt definieren.

Definition 10.6.
Sei $f : X \to Y$ eine Funktion.

surjektiv

(*a*) Wir bezeichnen f als *surjektiv*, wenn es für jedes $y \in Y$ mindestens ein $x \in X$ mit $f(x) = y$ gibt.

injektiv

(*b*) Wir bezeichnen f als *injektiv*, wenn es für jedes $y \in Y$ höchstens ein $x \in X$ mit $f(x) = y$ gibt.

bijektiv

(*c*) Wir bezeichnen f als *bijektiv*, wenn es für jedes $y \in Y$ genau ein $x \in X$ mit $f(x) = y$ gibt.

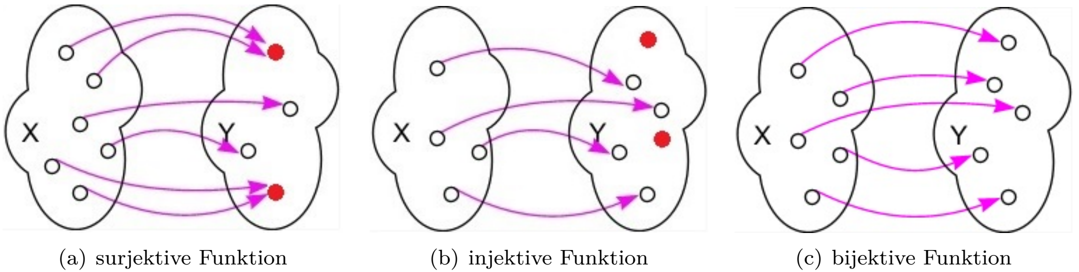
Es lässt sich schnell zeigen:

Aufgabe 10.4.
Sei $f : X \to Y$ eine Funktion.

f ist genau dann bijektiv, wenn f injektiv und surjektiv ist.

Abbildung 10.1 visualisiert eine surjektive, eine injektive und eine bijektive Funktion von X nach Y. Ein rosa Pfeil zwischen zwei Kringeln bedeutet, dass das Element x am Pfeilanfang durch die Funktion f auf das Element y am Pfeilende abgebildet wird, d.h. $f(x) := y$.

Wir untersuchen im folgenden Beispiel noch eine Auswahl an Funktionen auf Surjektivität, Injektivität und Bijektivität, damit du lernst, wie man diese für eine Funktion nachweisen bzw. widerlegen kann.



(a) surjektive Funktion (b) injektive Funktion (c) bijektive Funktion

Abbildung 10.1.: Veranschaulichung der Begriffe „surjektiv“, „injektiv“ und „bijektiv“.
Quelle: http://de.wikibooks.org/wiki/Mathematik:_Analysis:_Grundlagen:_Funktionen

Beispiel 10.7.

- Die Betragsfunktion $f : \mathbb{R} \to \mathbb{R},\ f(x) := |x|$ ist nicht surjektiv, da es z.B. zu -1 kein $x \in \mathbb{R}$ mit $f(x) = -1$ gibt.

  Sie ist auch nicht injektiv, da z.B. 1 und -1 denselben Funktionswert, nämlich 1 haben: $f(1) = f(-1) = 1$.

  Mit Aufgabe 10.4 kann f dann auch nicht bijektiv sein.

- Die Betragsfunktion $g : \mathbb{R} \to \mathbb{R}_{\geq 0},\ g(x) := |x|$ ist surjektiv, da für jedes nichtnegative y gilt: $f(y) = y$, d.h. zu jedem y im Bildbereich gibt es ein Element im Definitionsbereich, was auf y abgebildet wird.

  Sie ist aus dem gleichen Grund wie f aber weder injektiv noch bijektiv.

- Die Betragsfunktion $h : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0},\ h(x) := |x|$ ist aus dem gleichen Grund wie g surjektiv.

  Sie ist auch injektiv, denn zu jedem y im Bildbereich ist das einzige Element im Definitionsbereich, welches auf y abgebildet wird, y selbst.

  Mit Aufgabe 10.4 ist h auch bijektiv.

Durch Verkleinerung des Definitionsbereichs kann man jede Funktion zu einer injektiven Funktion machen und indem man den Bildbereich einer Funktion „so klein wie möglich“ macht, kann man jede nicht-surjektive Funktion in eine surjektive verwandeln.

Satz 10.8.
Sei $f : X \to Y$ eine Funktion.

f ist genau dann surjektiv, wenn $Y = f(X)$ gilt.

Beweis. Sei $f : X \to Y$ eine Funktion. Wir müssen wieder zwei Richtungen zeigen, da es sich um eine Genau-dann-wenn-Aussage (eine *Äquivalenz*) handelt

„$\Leftarrow$“: Es gelte $Y = f(X)$. Wir wollen zeigen, dass daraus die Surjektivität von f folgt. Also müssen wir nach Definition 10.6 beweisen, dass es für jedes $\tilde{y} \in Y$ mindestens ein $x \in X$ mit $f(x) = \tilde{y}$ gibt.

Sei $\tilde{y} \in Y$ beliebig. Da $Y = f(X)$, ist also $\tilde{y} \in f(X)$. Nach Definition 10.1 gehört demnach $\tilde{y}$ zur Menge $\{y \in Y \mid \text{ Es gibt ein } x \in X \text{ mit } f(x) = y\}$. Das heißt, es gibt ein $x \in X$ mit $f(x) = \tilde{y}$ und das wollten wir zeigen. Also ist f surjektiv.

„⇒“: Wir müssen noch beweisen, dass aus der Surjektivität von f folgt, dass $Y = f(X)$ sein muss.

Das erledigen wir mittels eines Beweises durch Widerspruch. Angenommen, f ist surjektiv, d.h. für jedes $y \in Y$ gibt es ein $x \in X$ mit $f(x) = y$. Wäre $Y \neq f(X)$, so wäre $Y \Delta f(X) \neq \emptyset$ nach Aufgabe 8.5. Also wäre $f(X) \backslash Y \neq \emptyset$ oder $Y \backslash f(X) \neq \emptyset$.

Im Falle von $f(X) \backslash Y \neq \emptyset$ müsste es ein Element in $f(X)$ geben, das nicht in Y liegt. Nach Definition 10.1(c) sind aber alle Elemente aus $f(X)$ erst recht Elemente aus Y, d.h. $f(X) \subseteq Y$, also kann es das gesuchte Element nicht geben.

Falls $Y \backslash f(X) \neq \emptyset$, so müsste es ein Element in Y geben, das nicht in $f(X)$ liegt, d.h. ein $y \in Y$, sodass es *kein* $x \in X$ mit $f(x) = y$ gibt. Das ist aber ein Widerspruch zur Surjektivität von f nach Definition 10.6, denn diese besagt, dass es für jedes $y \in Y$ so ein $x \in X$ gibt.

Jede Möglichkeit in der Fallunterscheidung hat zu einem Widerspruch geführt, also muss die Annahme $Y \neq f(X)$ falsch gewesen sein. Somit gilt $Y = f(X)$. □

Satz 10.8 wird dir bei der Bearbeitung der nächsten Aufgabe hilfreich sein, denn du kannst ihn anwenden, um aus einer beliebigen Funktion eine surjektive zu konstruieren.

Aufgabe 10.5.
Welche der Funktionen aus Aufgabe 10.3 sind injektiv bzw. surjektiv bzw. bijektiv? Gelingt es dir, die nicht-injektiven in injektive bzw. die nicht-surjektiven in surjektive zu verwandeln, indem du Definitions- und Bildbereich anpasst? Verändere diese dafür so wenig wie möglich!

Anwendungen des letzten Satzes, den wir in diesem Kapitel vorstellen wollen, werden dir oft begegnen. Er bildet die Grundlage für die Definition der *Abzählbarkeit* einer nicht-endlichen Menge. Insbesondere die „Durchnummerierungstechnik“, die im Beweis verwendet wird, findet man auch in anderen Kontexten und Argumentationen wieder.

Satz 10.9.
Seien X und Y endliche Mengen.

Es gilt $|X| = |Y|$ genau dann, wenn es eine bijektive Funktion von X nach Y gibt.

Beweis. Seien X und Y endliche Mengen. Wir beweisen erneut beide Teile des Satzes separat.

„⇒“: Angenommen, es gibt eine bijektive Funktion $f : X \to Y$. Wir wollen beweisen, dass dann die Mächtigkeiten von X und Y gleich sind, also beide gleich viele Elemente beinhalten.

Dazu nummerieren wir die Elemente von X durch, sodass $X = \{x_1, x_2, \ldots, x_n\}$ gilt. Das heißt, wir weisen einem der Elemente in X die Bezeichnung x_1 zu, einem anderen die Bezeichnung x_2 usw. Für jedes x_i bezeichnen wir $f(x_i)$ mit y_i. Weil f bijektiv ist, ist es nach Aufgabe 10.4 insbesondere surjektiv. D.h. nach Definition 10.6 existiert kein $y \in Y$, zu dem es kein $x \in X$ gibt. Also ist jedes $y \in Y$ von der Form $f(x_i)$ für irgendein i und jedes erhält daher eine Bezeichnung der Form y_i.

Aus der Bijektivität von f folgt aber ebenso mit Aufgabe 10.4 seine Injektivität. D.h. zu jedem $y \in Y$ gibt es höchstens ein $x_i \in X$ mit $y = f(x_i)$. Also bekommt jedes y nur eine Bezeichnung zugewiesen.

Aus der Bijektivität von f konnten wir so folgern, dass jedes $y \in Y$ genau eine Nummer zugewiesen bekommt. Weil f eine Funktion ist, gibt es zu jedem $xinX$ genau ein y, auf das x abgebildet wird, d.h. keine Nummer wird in Y doppelt vergeben. Also werden genauso viele verschiedene Nummern

bei der beschriebenen Durchnummerierung der Elemente von Y wie für die der Elemente von X vergeben, nämlich $|X|$. Also haben X und Y gleich viele Elemente, d.h. $|X| = |Y|$.

„$\Leftarrow$“: Angenommen, X und Y haben die gleiche Mächtigkeit n. Gesucht ist eine bijektive Funktion von X nach Y. Dazu nummerieren wir die Elemente von X erneut zu $X = \{x_1, x_2, \ldots, x_n\}$ durch, ebenso verfahren wir mit Y und erhalten $Y = \{y_1, y_2, \ldots, y_n\}$. Wir definieren nun noch $f(x_i) := y_i$ für alle $i \in \{1, 2, \ldots, n\}$. Dieses f weist jedem Element aus X nach Konstruktion genau eines aus Y zu, also ist f eine Funktion von X nach Y. Zu jedem $y_i \in Y$ gibt es ein Element aus X, was darauf abgebildet wird, nämlich x_i. Demnach ist f surjektiv. Außerdem werden keine zwei Elemente aus X auf dasselbe $y \in Y$ abgebildet, denn für $x_i \neq x_j$ gilt auch $f(x_i) = y_i \neq y_j = f(x_j)$. Das bedeutet, f ist injektiv.

Mit Aufgabe 10.4 folgt die Bijektivität von f und damit die Existenz der gesuchten bijektiven Funktion. □

In der letzten Aufgabe dieses Kapitels kannst du die Anwendung von Satz 10.9 üben sowie prüfen, ob du verstanden hast, wie man die Bijektivität einer Funktion nachweist. Auch die Durchnummerierungstechnik wird hier benutzt.

Aufgabe 10.6.
Sei M eine endliche Menge mit Kardinalität n. Bestimme die Mächtigkeit von $\mathcal{P}(M)$, der Potenzmenge von M.

Hinweis: Definiere dazu X als kartesisches Produkt $\{0,1\}^n$. Nummeriere die Elemente von M durch. Konstruiere dann eine Funktion $f : X \to \mathcal{P}(M)$, indem du jedem $x = (x_1, x_2, \ldots, x_n) \in X$ das Element aus $\mathcal{P}(M)$ zuweist, das genau die $m_i \in M$ mit $x_i = 1$ enthält. Weise nach, dass f tatsächlich eine Funktion und außerdem bijektiv ist. Wende Satz 9.6 an, um die Mächtigkeit von X zu bestimmen und folgere mit Satz 10.9 die Mächtigkeit von $\mathcal{P}(M)$.

Im nächsten Kapitel erklären wir die bereits im Beweis zu Satz 9.6 erwähnte vollständige Induktion als wichtige Beweistechnik ausführlich. Außerdem lernst du Rekursionen kennen, rekursiv definierte Funktionen werden dir insbesondere im Zusammenhang mit Laufzeiten von Algorithmen oft begegnen.

11. Beweistechniken

Ein Beweis ist eine **logisch vollständige Begründung** einer Aussage. Solange eine Aussage nicht bewiesen ist, kann es sein, dass sie falsch ist, auch wenn zahlreiche Beispiele die Aussage zu bestätigen scheinen.

Beispiel 11.1 (FERMAT-Zahlen)**.**
Benannt nach dem französischen Mathematiker *Pierre de Fermat*, brechnen sich die Zahlen aus $F_n = 2^{2^n} + 1$.

$$\begin{aligned} F_0 &= 2^{2^0} + 1 &&= 2^1 + 1 = 2 + 1 &&= 3 \\ F_1 &= 2^{2^1} + 1 &&= 2^2 + 1 = 4 + 1 &&= 5 \\ F_2 &= 2^{2^2} + 1 &&= 2^4 + 1 = 16 + 1 &&= 17 \end{aligned}$$

FERMAT vermutete 1637, dass alle F_n Primzahlen sind. Erst im Jahr 1732 konnte EULER beweisen, dass diese Vermutung nicht stimmt. Er zeigte, dass $F_5 = 4\,294\,967\,297$ durch 641 teilbar und somit keine Primzahl ist.

Jeder Beweis ist aus einzelnen, leicht nachvollziehbaren Schritten aufgebaut.

11.1. Direkter Beweis

Um die Aussage $A \to B$ zu beweisen, beginnt man mit der Prämisse A und argumentiert dann unter Verwendung von Definitionen und bereits bewiesenen Aussagen schrittweise, bis man bei der Konklusion B angelangt ist. Um $A \to B$ zu beweisen, argumentieren wir also $A \to A_1$ und $A_1 \to A_2$ usw. bis wir bei $A_i \to B$ angelangt sind.

Beispiel 11.2.
Wir wollen nun die Korrektheit des folgenden Satz mit einem direkten Beweis zeigen.

Satz 11.3.
Die Summe zweier gerader Zahlen ist wiederum eine gerade Zahl

Bevor wir mit dem Beweis loslegen können, müssen wir erstmal klären, wie eine gerade Zahl definiert ist.

Definition 11.4.
Eine Zahl ist genau dann gerade, wenn sie durch 2 teilbar ist.

Jetzt müssen wir noch klarstellen, was wir mit "*teilbar sein*" meinen.

Definition 11.5.
Eine Zahl $a \in \mathbb{Z}$ ist genau dann durch eine andere Zahl $b \neq 0$ teilbar, wenn es eine Zahl $k \in \mathbb{Z}$ gibt, sodass $a = b \cdot k$.[1](Beispiel: Für $a = 24, b = 6$ und $k = 4$ gilt: $24 = 6 \cdot 4$. 24 ist also durch 6

[1] $\mathbb{Z}$ bezeichnet die Menge der ganzen Zahlen, also $\mathbb{Z} = \{0, -1, 1, -2, 2, -3, 3, \dots\}$.

teilbar.)

Umgangssprachlich sagen wir also, dass eine Zahl a gerade ist, wenn sie ohne Rest durch 2 teilbar ist.

Nachdem wir also geklärt haben, was eine gerade Zahl ist, schauen wir nach Prämisse A und Konklusion B. Damit die zwei Teilaussagen der Aussage deutlicher werden, formulieren wir die Aussage um, **ohne** den Sinn der Aussage zu verändern.

$$\underbrace{\text{Wenn zwei gerade Zahlen addiert werden}}_{A \qquad\qquad\qquad \rightarrow}, \underbrace{\text{dann ist das Ergebnis wieder eine gerade Zahl}}_{B}$$

Kommen wir jetzt zum Beweis.

Beweis

Zu zeigen ist also, dass bei der Addition zweier gerader Zahlen, wieder eine gerade Zahl entsteht; und zwar bei der Addition egal welcher beider geraden Zahlen und nicht nur bei der Addition zweier bestimmter gerader Zahlen. Es handelt sich hier also auch um eine "Für alle" Aussage, nicht um eine "Es gibt" Aussage. Es ist also nicht damit getan zwei gerade Zahlen anzugeben, deren Ergebniss wieder eine gerade Zahl ist. Wir werden für alle geraden Zahlen argumentieren müssen.

Seien also a und b zwei beliebige gerade Zahlen. Laut Definition 11.4 sind a und b durch 2 teilbar. Das bedeutet demnach laut Definition 11.5, dass $a = 2 \cdot k$ und $b = 2 \cdot l$ für zwei Zahlen $k, l \in \mathbb{Z}$. Wenn wir also jetzt a und b addieren, dann können wir schreiben:

$$\begin{aligned} a + b &= 2 \cdot k + 2 \cdot l \\ &= 2 \cdot (k + l) \\ &= 2 \cdot m && \text{für } m = k + l \end{aligned}$$

Wenn wir nun argumentieren können, dass $m \in \mathbb{Z}$ ist, haben wir's geschafft, denn dann ist $a + b$ nach Definition 11.5 durch zwei teilbar und nach Definition 11.4 somit gerade.

m ist eine ganze Zahl, da die ganzen Zahlen unter $+$ abgeschlossen sind. Das bedeutet, dass bei der Addition zweier ganzer Zahlen das Ergebnis auch immer eine ganze Zahl ist. Strenggenommen müssten wir diese Abgeschlossenheit jetzt noch beweisen, das ginge aber hier zu weit. Daher nehmen wir es als bereits bewiesene Aussage an.

Zusammenfassung
Ausgehend von der Prämisse, dass a und b gerade Zahlen sind, haben wir unter Verwendung der Definitionen für gerade Zahlen und Teilbarkeit, sowie die als bewiesen angenommene Aussage, dass die Summer zweier ganzer Zahlen wieder eine ganze Zahl ist, schrittweise argumentiert, dass dann die Summe $a + b$ ebenfalls eine gerade Zahl ist.

Betrachten wir einen weiteren direkten Beweis.

Satz 11.6.
Alle Primzahlen bis auf die 2 sind ungerade.
Umformuliert ergibt sich der Satz:

$$\underbrace{\text{Wenn } x \neq 2 \text{ eine Primzahl ist}}_{\text{A} \qquad\qquad \rightarrow}, \underbrace{\text{dann ist } x \text{ ungerade}}_{B}.$$

Beweis
Zu zeigen ist, dass für jede Primzahl $\neq 2$ gilt, dass sie ungerade ist.
Sei x also eine Primzahl. Laut Definition der Primzahlen, ist x somit durch 1 und sich selbst teilbar und sonst durch keine andere Zahl. Insbesondere ist x dann auch nicht durch 2 teilbar, denn das würde nur für $x = 2$ gelten und das ist ja ausgeschlossen. Laut Definition der Teilbarkeit (Def 11.5) ist x also **nicht** durch $2 \cdot k, k \in \mathbb{Z}$ darstellbar und somit laut Definition der geraden Zahlen (Def 11.4) **nicht** gerade. Damit muss x also ungerade sein. □

11.1.1. Abgeschlossenheit einer Zahlenmenge bezüglich einer Verknüpfung

Zahlen kann man auf verschiedene Weise miteinander "verknüpfen". Z. B. kann man sie addieren, subtrahieren, multiplizieren und dividieren. Als Verallgemeinerung schreiben wir hier für eine solche Verknüpfung das Zeichen $\circ$ [2].

Definition 11.7 (Abgeschlossenheit)**.**
Eine Zahlenmenge heißt *abgeschlossen* bezüglich einer Verknüpfung, wenn für alle Zahlen a, b aus der Menge gilt, dass das Ergebnis der Verknüpfung $a \circ b$ auch wieder in der Zahlenmenge ist.

Im Vorkurs dürfen Sie folgende Abgeschlossenheiten als bereits bewiesen annehmen:

- Die natürlichen Zahlen $\mathbb{N}$ sind abgeschlossen bezüglich der Verknüpfungen $+$ und $\cdot$.
- Die ganzen Zahlen $\mathbb{Z}$ sind abgeschlossen bezüglich der Verknüpfungen $+$, - und $\cdot$.

Beispiel 11.8.
Den direkten Beweis können wir z.B. auch nutzen, um die in Aufgabe 8.5 gemachte Behauptung der Distributivität von Schnitt und Vereinigung von Mengen zu zeigen. Zu zeigen ist: Für die Mengen A, B und C gilt:

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C).$$

Nach Definition 8.3 müssen wir für die Gleichheit der beiden Mengen $A \cup (B \cap C)$ und $(A \cup B) \cap (A \cup C)$ zeigen, dass sie dieselben Elemente enthalten, oder, nach Satz 8.5, dass die erste in der zweiten und die zweite in der ersten Menge enhalten ist.

Wir beginnen damit zu zeigen, dass $A \cup (B \cap C) \subset (A \cup B) \cap (A \cup C)$ ist. Nach Definition 8.3 müssen wir hierfür zeigen, dass jedes Element von $A \cup (B \cap C)$ auch in $(A \cup B) \cap (A \cup C)$ enthalten ist.

Sei $x \in A \cup (B \cap C)$ beliebig gewählt. Für x gilt dann:

$$\begin{array}{rcll} x \in A & \cup & (B \cap C) & \\ \leftrightarrow x \in A & \vee & x \in (B \cap C) & \text{(Def. Vereinigung von Mengen)} \\ \leftrightarrow x \in A & \vee & (x \in B \ \wedge \ x \in C) & \text{(Def. Schnitt von Mengen)} \end{array}$$

[2] Beispiel: $5 \circ 2$ kann also für $5 + 2, 5 - 2, 5 \cdot 2$ oder $5 : 2$ stehen.

Wir müssen also die zwei Fälle $x \in A$ und $x \notin A$ betrachten:

Fall 1: $x \in A$. Aus Definition 8.7 zu Vereinigung und Schnitt von Mengen folgt, dass dann auch $x \in (A \cup B)$ und $x \in (A \cup C)$ ist und somit $x \in (A \cup B) \cap (A \cup C)$.
Fall 2: $x \notin A$. Dann muss nach $x \in A \vee (x \in B \wedge x \in C)$ also $x \in B$ und $x \in C$ sein. Damit ist $x \in (A \cup B) \cap (A \cup C)$.

Somit gilt in beiden Fällen $x \in (A \cup B) \cap (A \cup C)$. Da x beliebig gewählt war, gilt also allgemein für $x \in A \cup (B \cap C)$, dass $x \in (A \cup B) \cap (A \cup C)$ und somit $A \cup (B \cap C) \subset (A \cup B) \cap (A \cup C)$.

Nun müssen wir noch zeigen, dass $(A \cup B) \cap (A \cup C) \subset A \cup (B \cap C)$ ist.

Sei $x \in (A \cup B) \cap (A \cup C)$ beliebig gewählt. Für x gilt dann:

$$\begin{aligned} & x \in (A \cup B) && \cap && (A \cup C) && \\ \leftrightarrow\ & x \in (A \cup B) && \wedge && x \in (A \cup C) && \text{(Def. Schnitt von Mengen)} \\ \leftrightarrow\ & (x \in A \vee x \in B) && \wedge && (x \in A \vee x \in C) && \text{(Def. Vereinigung von Mengen)} \end{aligned}$$

Es gibt wieder zwei Fälle, nämlich $x \in A$ und $x \notin A$ zu betrachten:

Fall 1: $x \in A$. In diesem Fall ist auch $x \in A \cup (B \cap C)$.
Fall 2: $x \notin A$. Dann muss aber $x \in B$ und $x \in C$ sein. Somit ist $x \in B \cap C$ und ebenfalls $x \in A \cup (B \cap C)$.

Da x beliebig gewählt war, gilt somit allgemein für $x \in (A \cup B) \cap (A \cup C)$, dass $x \in A \cup (B \cap C)$ ist. Damit gilt nach der Definition von Teilmengen: $(A \cup B) \cap (A \cup C) \subset A \cup (B \cap C)$ □

11.2. Indirekter Beweis / Beweis durch Kontraposition

Manchmal ist es schwierig von der Prämisse auf die Konklusion zu schließen. Wie wir oben gesehen haben, ist die Aussage $A \to B$ aber äquivalent zu der Aussage $\neg B \to \neg A$. Wir können also auch versuchen zu zeigen, dass wenn die Konklusion (Aussage B) nicht gilt, dann auch die Prämisse (Aussage A) nicht gelten kann und damit die Aussage $\neg B \to \neg A$ zeigen. Da die beiden Aussagen äquivalent sind, hätten wir damit indirekt gezeigt, dass auch $A \to B$ gilt.

A	B	$A \to B$	$\neg A$	$\neg B$	$\neg B \to \neg A$
0	0	1	1	1	1
0	1	1	1	0	1
1	0	0	0	1	0
1	1	1	0	0	1

Beispiel 11.9.

Satz 11.10.
Wenn a^2 gerade ist, dann ist auch a gerade.

Wenn wir diese Aussage direkt beweisen wollen, wird es schwierig, denn wie sollen wir aus $a^2 = 2 \cdot k, k \in \mathbb{Z}$ argumentieren, dass daraus folgt, dass $a = \sqrt{2 \cdot k}, k \in \mathbb{Z}$ ebenfalls gerade ist?

Also versuchen wir statt $A \to B$, $\neg B \to \neg A$ zu beweisen.

	Aussage	negierte Aussage
A	a^2 ist eine gerade Zahl	a^2 ist eine ungerade Zahl
B	a ist eine gerade Zahl	a ist eine ungerade Zahl

Statt "*Wenn a^2 eine gerade Zahl ist, dann ist auch a eine gerade Zahl.*" beweisen wir also "*Wenn a eine ungerade Zahl ist, dann ist auch a^2 eine ungerade Zahl.*".

Beweis
Sei a also eine beliebige ungerade Zahl. In Anlehnung an die Definition gerader Zahlen (Def 11.4) lässt sich a dann als $a = 2 \cdot k + 1$ mit $k \in \mathbb{Z}$ darstellen. Wenn wir a nun quadrieren, passiert folgendes:

$$\begin{aligned}
a^2 &= (2 \cdot k + 1)^2 && \text{(quadrieren)} \\
&= (2 \cdot k)^2 + 2 \cdot (2 \cdot k \cdot 1) + 1^2 && \text{(Binomische Formel mit } a = 2 \cdot k \text{ und } b = 1) \\
&= \mathbf{2} \cdot 2 \cdot k^2 + \mathbf{2} \cdot (2 \cdot k \cdot 1) + 1^2 \\
&= \mathbf{2} \cdot (2 \cdot k^2 + 2 \cdot k) + 1^2 && \text{(ausklammern)} \\
&= 2 \cdot l + 1 && \text{mit } l = 2 \cdot k^2 + 2 \cdot k
\end{aligned}$$

Bleibt also zu argumentieren, dass $l \in \mathbb{Z}$ ist. Das ist aber der Fall, denn 2 und k sind ganze Zahlen, also $2, k \in \mathbb{Z}$ und die ganzen Zahlen sind abgeschlossen unter $+$ und $\cdot$ (siehe oben). Somit ist $l = 2 \cdot k^2 + 2 \cdot k$ ebenfalls eine ganze Zahl und $a^2 = 2 \cdot l + 1$ ungerade. □

Damit haben wir indirekt unsere ursprüngliche Aussage "*Wenn a^2 gerade ist, dann ist a ebenfalls gerade*" bewiesen.

11.3. Beweis durch Widerspruch

Das Vorgehen beim Beweis durch Widerspruch ist ähnlich wie beim indirekten Beweis. Statt $A \to B$ zeigen wir, dass $A \wedge \neg B$ zu einem Widerspruch führt und somit falsch ist. Das heißt wir zeigen indirekt, dass $\neg(A \wedge \neg B)$ wahr ist. Diese Aussage ist äquivalent zur Aussage $A \to B$ (siehe Wahrheitstabelle) und somit haben wir indirekt auch $A \to B$ gezeigt.

A	B	$A \to B$	$\neg A \vee B$	$A \wedge \neg B$	$\neg(A \wedge \neg B)$
0	0	1	1	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	1	1	1	0	1

Manchmal gibt es auch keine Prämisse. Man möchte einfach nur eine Aussage C beweisen. Das Vorgehen ist in diesem Fall, anzunehmen, dass die Aussage nicht gilt (also $\neg C$ anzunehmen) und zu zeigen, dass das zu einem Widerspruch führt. Damit hat man gezeigt, dass $\neg C \to \mathbf{F}$ wahr ist und somit $\neg C$ falsch ist. Im Umkehrschluss muss C aber dann wahr sein, denn wenn $\neg C$ falsch ist, muss $\neg(\neg C)$ wahr sein. $\neg(\neg C)$ ist aber nichts anderes als die ursprüngliche Aussage C.

Satz 11.11.
$\sqrt{2}$ ist eine irrationale Zahl.

In diesem Beispiel gibt es keine Prämisse. Wir wollen einen Beweis durch Widerspruch führen und nehmen an, dass die negierte Aussage gilt. Dann hoffen wir durch schrittweises Argumentieren auf einen Widerspruch zu unserer Annahme zu stoßen und damit zu zeigen, dass die negierte Aussage falsch ist. Somit muss die ursprüngliche Aussage wahr sein.

Beweis

Nehmen wir also an, $\sqrt{2}$ sei rational. D. h. $\sqrt{2} \in \mathbb{Q}$.[3] Also können wir $\sqrt{2} = \frac{p}{q}$ schreiben, mit $p, q \in \mathbb{Z}$ und p, q teilerfremd, also $\text{ggT}(p, q) = 1$. Wenn wir jetzt quadrieren, erhalten wir $2 = \frac{p^2}{q^2}$ und durch Umformung $2 \cdot q^2 = p^2$. Da $q^2 \in \mathbb{Z}$, ist p^2 laut Definition der Teilbarkeit (Def 11.5) durch 2 teilbar. Wie soeben bewiesen, ist dann aber auch p durch 2 teilbar und als $p = 2 \cdot k$ mit $k \in \mathbb{Z}$ darstellbar. Also lässt sich die Gleichung $2 \cdot q^2 = p^2$ auch als $2 \cdot q^2 = (2 \cdot k)^2 = 2 \cdot 2 \cdot k^2$ schreiben. Wenn man jetzt auf beiden Seiten durch 2 teilt, erhält man $q^2 = 2 \cdot k^2$. Damit ist q^2 durch 2 teilbar, denn $k^2 \in \mathbb{Z}$ und somit auch q durch 2 teilbar. Das heißt sowohl p als auch q sind durch 2 teilbar. Das allerdings ist eine Widerspruch zu der Annahme, dass p und q teilerfremd sind, denn sie haben 2 als gemeinsamen Teiler. Somit ist die Aussage *"$\sqrt{2}$ ist eine rationale Zahl"* widerlegt und entsprechend ist die Aussage *"$\sqrt{2}$ ist eine irrationale Zahl"* bewiesen. □

[3] Rationale Zahlen sind dadurch gekennzeichnet, dass sie als Bruch zweier ganzer Zahlen darstellbar sind

12. Induktion und Rekursion

Strenggenommen sollte dieses Kapitel ein Unterkapitel des vorherigen Kapitels sein, denn die *vollständige Induktion* ist eine mathematische Beweistechnik. Allerdings ist diese Technik komplexer als die bisher behandelten Beweistechniken daher widmen wir ihr hier ein eigenes Kapitel.

12.1. Vollständige Induktion

Ziel der vollständigen Induktion ist es zu beweisen, dass eine Aussage $A(n)$ für alle $n \in \mathbb{N}_0$[1] gilt. Dabei verwendet man das **Induktionsprinzip**, d.h. man schließt vom Besonderen auf das Allgemeine. (Im Gegensatz zur *Deduktion*, wo man vom Allgemeinen auf das Besondere schließt.) Das Vorgehen ist folgendermaßen:

Induktions-prinzip

1. Für eine gegebene Aussage A zeigt man zunächst, dass die Aussage für ein (meist $n = 0$, oder $n = 1$) oder einige kleine n wahr ist. Diesen Schritt nennt man **Induktionsanfang**. (Häufig findet sich in der Literatur auch *Induktionsbasis* oder *Induktionsverankerung.*)

   Induktions-anfang

2. Dann zeigt man im **Induktionsschritt**, dass für jede beliebige Zahl $n \in \mathbb{N}$ gilt:
   Falls die Aussage $A(n)$ wahr ist, so ist auch die Aussage $A(n+1)$ wahr. (**Induktionsbehauptung**)

   Induktions-schritt
   Induktions-behauptung

Wenn man also gezeigt hat, dass $A(n+1)$ aus $A(n)$ folgt, dann gilt insbesondere $A(1)$, falls $A(0)$ wahr ist. Damit gilt dann aber auch $A(2)$, da $A(1)$ gilt, $A(3)$ da $A(2)$ gilt, usw. .

Für das erste Beispiel zur vollständigen Induktion werden abkürzende Schreibweisen für Summen und Produkte eingeführt.

Definition 12.1.
Sei $n \in \mathbb{N}$, und seien $a_1, \ldots, a_n$ beliebige Zahlen. Dann ist:

- $\sum_{i=1}^{n} a_i := a_1 + a_2 + \cdots + a_n$ (Summe)
  insbesondere ist die leere Summe $\sum_{i=1}^{0} a_i = 0$.
- $\prod_{i=1}^{n} a_i := a_1 \cdot a_2 \cdot \cdots \cdot a_n$ (Produkt)
  insbesondere ist das leere Produkt $\prod_{i=1}^{0} a_i = 1$.

Beispiel 12.2.

Satz 12.3 (kleiner Gauß)**.**
$A(n)$:Für alle $n \in \mathbb{N}$ gilt:

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

Induktionsanfang: $n = 0$
Behauptung: $A(0)$: Der Satz gilt für $n = 0$.

[1] Wir schreiben $\mathbb{N}_0$ für die natürlichen Zahlen inklusive der 0 ($\mathbb{N} \cup \{0\}$)

Beweis:

$$\sum_{i=1}^{0} i = 0 = \frac{0}{2} = \frac{0(0+1)}{2} = \frac{n(n+1)}{2}$$

Induktionsschritt: $A(n) \to A(n+1)$
Induktionsvoraussetzung: Es gilt $A(n)$, also $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
Unter dieser Voraussetzung muss nun gezeigt werden, dass der Satz auch für $n+1$ gilt.
Induktionsbehauptung: Es gilt $A(n+1)$:

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

Beweis:

$$\begin{aligned}
\sum_{i=1}^{n+1} i &= 1 + 2 + \cdots + n + (n+1) \\
&= (\sum_{i=1}^{n} i) + (n+1) && \text{Induktionsvoraussetzung anwenden} \\
&= \frac{n(n+1)}{2} + (n+1) && \text{mit 2 erweitern} \\
&= \frac{n(n+1) + 2(n+1)}{2} \\
&= \frac{(n+1)(n+2)}{2} \\
&= \frac{(n+1)((n+1)+1)}{2}
\end{aligned}$$

□

Im Folgenden wird der besseren Lesbarkeit wegen, statt $A(n) \to A(n+1)$ lediglich $n \to n+1$ geschrieben und vorausgesetzt, dass dem Leser klar ist, dass im Fall $n = 0$ die Aussage $A(0)$ bzw. im Fall $n = 1$ die Aussage $A(1)$ gemeint ist.

Beispiel 12.4.

Satz 12.5.
Für alle $n \in \mathbb{N}$ und $q \neq 1$ gilt:

$$\sum_{i=0}^{n} q^i = \frac{1 - q^{n+1}}{1 - q}$$

Induktionsanfang: $n = 0$
Behauptung: Es gilt: $\sum_{i=0}^{0} q^i = \frac{1-q^{0+1}}{1-q}$
Beweis:

$$\sum_{i=0}^{0} q^i = q^0 = 1 = \frac{1-q}{1-q} = \frac{1-q^1}{1-q} = \frac{1-q^{0+1}}{1-q}$$

Induktionsschritt: $n \to n+1$
Induktionsvoraussetzung: Es gilt $\sum_{i=0}^{n} q^i = \frac{1-q^{n+1}}{1-q}$.
Induktionsbehauptung: Es gilt:

$$\sum_{i=0}^{n+1} q^i = \frac{1 - q^{(n+1)+1}}{1 - q}$$

Beweis:

$$
\begin{aligned}
\sum_{i=0}^{n+1} q^i &= q^1 + q^2 + \cdots + q^n + q^{n+1} \\
&= \sum_{i=0}^{n} q^i + q^{n+1} && \text{Induktionsvoraussetzung} \\
&= \frac{1-q^{n+1}}{1-q} + q^{n+1} && \text{erweitern mit } (1-q) \\
&= \frac{1-q^{n+1}}{1-q} + \frac{q^{n+1}\cdot(1-q)}{1-q} \\
&= \frac{1-q^{n+1}+q^{n+1}-q^{n+1}\cdot q}{1-q} \\
&= \frac{1-q\cdot q^{n+1}}{1-q} \\
&= \frac{1-q^{(n+1)+1}}{1-q}
\end{aligned}
$$

□

Man kann mit der Beweistechnik der vollständigen Induktion jedoch nicht nur Gleichungen beweisen.

Beispiel 12.6.

Satz 12.7.
Sei $n \in \mathbb{N}$. $n^2 + n$ ist eine gerade (d.h. durch 2 teilbare) Zahl.

Beweis. durch vollständige Induktion.

Induktionsanfang: $n = 0$
Behauptung: $0^2 + 0$ ist eine gerade Zahl.
Beweis: $0^2 + 0 = 0 + 0 = 0$ ist eine gerade Zahl. Das stimmt, denn $0 = 2 \cdot 0, 0 \in \mathbb{Z}$ und damit nach Def 11.5 und Def. 11.4 gerade.
Da die 0 aber so speziell ist, kann man zur Sicherheit und zur Übung den Satz auch noch für $n = 1$ beweisen. Notwendig, ist das allerdings nicht.
Behauptung: $1^2 + 1$ ist eine gerade Zahl.
Beweis: $1^2 + 1 = 1 + 1 = 2$. $2 = 2 \cdot 1, 1 \in \mathbb{Z}$. Damit ist 2 eine gerade Zahl.

Induktionsschritt: $n \to n + 1$
Induktionsvoraussetzung: Für $n \geq 0$ gilt: $n^2 + n$ ist eine gerade Zahl.
Induktionsbehauptung: $(n + 1)^2 + (n + 1)$ ist eine gerade Zahl.
Beweis:

$$
\begin{aligned}
(n+1)^2 + (n+1) &= n^2 + 2n + 1 + n + 1 \\
&= n^2 + 3n + 2 \\
&= (n^2 + n) + (2n + 2) \\
&= (n^2 + n) + 2 \cdot (n + 2)
\end{aligned}
$$

$(n^2 + n) + 2 \cdot (n + 2)$ ist eine gerade Zahl, da laut Induktionsvoraussetzung $n^2 + n$ eine gerade Zahl ist, und $2 \cdot (n + 1)$ ist ein Vielfaches von 2. Somit ist auch der zweite Summand eine gerade Zahl, und die Summe gerader Summanden ist ebenfalls gerade. Das haben wir in Beispiel 11.2 bewiesen.

□

12.1.1. Wann kann man vollständige Induktion anwenden?

Die vollständige Induktion eignet sich, um Behauptungen zu beweisen, die sich auf Objekte (Zahlen, Geraden, Spielzüge,...) beziehen, die als natürliche Zahlen betrachtet werden können. Mathematisch korrekt ausgedrückt, muss die Objektmenge die sog. *Peano-Axiome* erfüllen. Diese sagen im wesentlichen, dass es ein erstes Element geben muss, jedes Element einen eindeutig bestimmten Nachfolger haben muss und das Axiom der vollständigen Induktion gelten muss.

Aussagen über reelle Zahlen lassen sich beispielsweise nicht mit vollständiger Induktion beweisen.

Oftmals ist man versucht zu meinen, dass Induktion immer dann möglich ist, wenn die Behauptung ein n enthält. Allerdings, ist folgender Satz nicht mit vollständiger Induktion zu beweisen.

Satz 12.8.
Sei $n \in \mathbb{N}$. Dann ist die Folge $a(n) = 1/n$ immer positiv.

Obiger Satz ist zwar wahr, aber wie soll man aus $\frac{1}{n} > 0$ folgern, dass $\frac{1}{n+1} > 0$? Genau das wäre für den Induktionsschritt aber notwendig. Im Induktionsschritt muss gezeigt werden, dass aus $A(n)$, $A(n+1)$ folgt.

12.1.2. Was kann schief gehen?

Das Prinzip der vollständigen Induktion lässt sich auch mit einem Domino-Effekt vergleichen. Die Bahn läuft durch, d.h. alle Dominosteine fallen um, wenn man den ersten Stein umstoßen kann und gesichert ist, dass jeder Stein n seinen Nachfolger $n+1$ umstößt.

In obigem Beispiel lässt sich nicht beweisen, dass jeder Stein n seinen Nachfolger $n+1$ umstößt. Daher ist die vollständige Induktion als Beweismethode für Satz 12.8 ungeeignet.

Der Beweis, dass der erste Stein umgestoßen werden kann, $A(n)$ gilt für ein *bestimmtes* n, der *Induktionsanfang*, ist genau so wichtig, wie der *Induktionsschritt*.

Beispiel 12.9 (fehlender Induktionsanfang)**.**
Im folgenden sei Teilbarkeit wie in Definition 11.5 definiert.
Zum Beispiel lässt sich aus der Aussage $A(5 \textit{ ist durch } 2 \textit{ teilbar})$ logisch korrekt folgern, dass auch $B(7 \textit{ ist durch } 2 \textit{ teilbar})$ gilt. Die Schlussfolgerung ist logisch korrekt, die Aussagen gelten aber nicht, da eben die Voraussetzung nicht gegeben ist. Denn 5 ist nunmal nicht durch 2 teilbar.

Während der Induktionsanfang meist relativ einfach zu beweisen ist, macht der Induktionsschritt häufiger Probleme. Die Schwierigkeit liegt darin, dass ein konstruktives Argument gefunden werden muss, das in Bezug auf die Aufgabenstellung tatsächlich etwas aussagt. Dies ist der Fehler im folgenden Beispiel.

Beispiel 12.10 (fehlerhafte Induktion)**.**
Behauptung: In einen Koffer passen unendlich viele Socken.
Induktionsanfang: $n = 1$
Behauptung: Ein Paar Socken passt in einen leeren Koffer.
Beweis: Koffer auf, Socken rein, Koffer zu. Passt.
Induktionsschritt: $n \to n+1$**:**
Induktionsvoraussetzung: n Paar Socken passen in den Koffer.
Induktionsbehauptung: $n+1$ Paar Sochen passen in den Koffer.
Beweis: n Paar Socken befinden sich im Koffer. Aus Erfahrung weiß man, ein Paar Socken passt immer noch rein. Also sind nun $n+1$ Paar Socken im Koffer. □
Somit ist bewiesen, dass unendlich viele Socken in einen Koffer passen.

Was ist passiert?
Das Argument „aus Erfahrung weiß man, ein Paar Socken passt immer noch rein“, ist in Bezug auf die Aufgabenstellung nicht konstruktiv. Ein konstruktives Argument hätte sagen müssen, wo genau das extra Paar Socken noch hinpasst.

Ferner muss man darauf achten, dass das n der Aussage $A(n)$ aus der man dann $A(n+1)$ folgert keine Eigenschaften hat, die im Induktionsanfang nicht bewiesen wurden.

Beispiel 12.11 (fehlerhafte Induktion).
Behauptung: Alle Menschen einer Menge M mit $|M| = n$ sind gleich groß.
Induktionsanfang: $n = 1$
Behauptung: In einer Menge von einem Menschen, sind alle Menschen dieser Menge gleich groß.
Beweis: Sei M eine Menge von Menschen mit $|M| = 1$. Da sich genau ein Mensch in M befindet, sind offensichtlich alle Menschen in M gleich groß.
Induktionsschritt: $n \to n+1$
Induktionsvoraussetzung: Sei $n \in \mathbb{N}$ beliebig. In einer Menge Menschen M', mit $|M'| = n$, haben alle Menschen die gleiche Größe.
Induktionsbehauptung: Ist M eine Menge Menschen mit $|M| = n+1$, so sind alle Menschen in M gleich groß.
Beweis: Sei $M = \{m_1, m_2, \ldots, m_{n+1}\}$ eine Menge von $n+1$ Menschen. Sei $M' = \{m_1, m_2, \ldots, m_n\}$ und $M'' = \{m_2, m_3, \ldots, m_{n+1}\}$. Damit sind M' und M'' Mengen von je n Menschen. Laut Induktionsannahme gilt dann:

1. Alle Menschen in M' haben die gleiche Größe g'.
2. Alle Menschen in M'' haben die gleiche Größe g''.

Insbesondere hat Mensch $m_2 \in M'$ Größe g' und Mensch $m_2 \in M''$ Größe g''. Da aber jeder Mensch nur eine Größe haben kann, muss gelten: $g' = g''$. Wegen $M = M' \cup M''$, haben somit alle Menschen in M die gleiche Größe $g = g' = g''$. □

Was ist passiert?
Der Induktionsschluss funktioniert nur für $n > 1$. Denn nur, wenn es mindestens 2 Menschen mit der gleichen Größe gibt, kann ich m_1, m_2 in M' und m_2, m_{n+1} in M'' einteilen. Im Fall $n = 1$, und $n+1 = 2$, gilt $M' = \{m_1\}$ und $M'' = \{m_2\}$. Dann ist $m_2 \in M''$, jedoch keinesfalls in M'. Die Argumentation im Induktionsschritt fällt in sich zusammen, denn es gibt keinen Grund, warum m_1 und m_2 die gleiche Größe haben sollten. Man hätte also im Induktionsanfang beweisen müssen, dass die Aussage auch für $n = 2$ gilt. Wie leicht einzusehen ist, wird das nicht gelingen, denn zwei willkürlich herausgegriffene Menschen sind keineswegs zwangsläufig gleich groß.

12.2. Rekursion

Rekursion ist eine Technik bei der Funktionen durch sich selbst definiert werden. Bei der rekursiven Definition wird das Induktionsprinzip angewendet. Zunächst wird, meist für kleine Eingaben, der Funktionswert explizit angegeben (*Rekursionsanfang*). Dann wird im *Rekursionsschritt* eine Vorschrift formuliert, wie die Funktionswerte für größere Eingaben mit Hilfe der Funktionswerte kleinerer Eingaben berechnet werden können.

Rekursionsanfang
Rekursionsschritt

Ein bekannter Spruch zur Rekursion lautet:

„Wer Rekursion verstehen will, muss Rekursion verstehen “

Definition 12.12 (Fakultätsfunktion).
Die Funktion $f : \mathbb{N} \to \mathbb{N}$, gegeben durch:

Fakultätsfunktion

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

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

Was genau beschreibt nun diese rekursive Definition? Einen besseren Überblick bekommt man meist, wenn man ein paar konkrete Werte für n einsetzt.

$$\begin{aligned}
&f(0) = 1 && n = 0\\
&f(1) = 1 \cdot f(0) = 1 \cdot 1 = 1 && n = 1\\
&f(2) = 2 \cdot f(1) = 2 \cdot (1 \cdot f(0)) = 2 \cdot (1 \cdot 1) = 2 && n = 2\\
&f(3) = 3 \cdot f(2) = 3 \cdot (2 \cdot f(1)) = 3 \cdot (2 \cdot (1 \cdot f(0))) = 3 \cdot (2 \cdot (1 \cdot 1)) = 6 && n = 3
\end{aligned}$$

Es liegt nahe, dass die Fakultätsfunktion das Produkt der ersten n natürlichen Zahlen beschreibt.

Satz 12.13.
Für die Fakultätsfunktion $f(n)$ gilt: $f(n) = \prod_{i=1}^{n} i$.

Beweis. Hier eignet sich vollständige Induktion zum Beweis des Satzes.

Induktionsanfang: $n = 0$
Behauptung: Der Satz gilt für $n = 0$.
Beweis: Nach Definition 12.12 gilt $f(0) = 1$. Nach Definition 12.1 gilt $\prod_{i=1}^{0} i = 1$. Im Fall von $n = 0$ ist somit $f(0) = 1 = \prod_{i=1}^{0} i = \prod_{i=1}^{n} i$

Induktionsschritt: $n \to n + 1$**:**
Induktionsvoraussetzung: Es gilt $f(n) = \prod_{i=1}^{n} i$ für n. Unter dieser Voraussetzung zeigt man nun, dass
Induktionsbehauptung: $f(n + 1) = \prod_{i=1}^{n+1} i$ gilt.
Beweis:

$$\begin{aligned}
f(n+1) &= (n+1) \cdot f(n) && \text{Definition 12.12}\\
&= (n+1) \cdot \prod_{i=1}^{n} i && \text{Induktionsvoraussetzung}\\
&= (n+1) \cdot n \cdot (n-1) \cdot \dots \cdot 2 \cdot 1\\
&= \prod_{i=1}^{n+1} i
\end{aligned}$$

□

Nicht nur Funktionen lassen sich rekursiv definieren, auch Mengen können rekursiv definiert werden.

Beispiel 12.14.
So lässt sich die Menge der natürlichen geraden Zahlen $\mathbb{N}_G = 0, 2, 4, \ldots$ folgendermaßen rekursiv definieren.

Basisregel: $0 \in \mathbb{N}_G$

Rekursive Regel: Ist $x \in \mathbb{N}_G$, so ist auch $x + 2 \in \mathbb{N}_G$.

Die implizite Definition der Menge $\mathbb{N}_G$ lautet: $\mathbb{N}_G = \{x | x = 2n, n \in \mathbb{N}\}$.

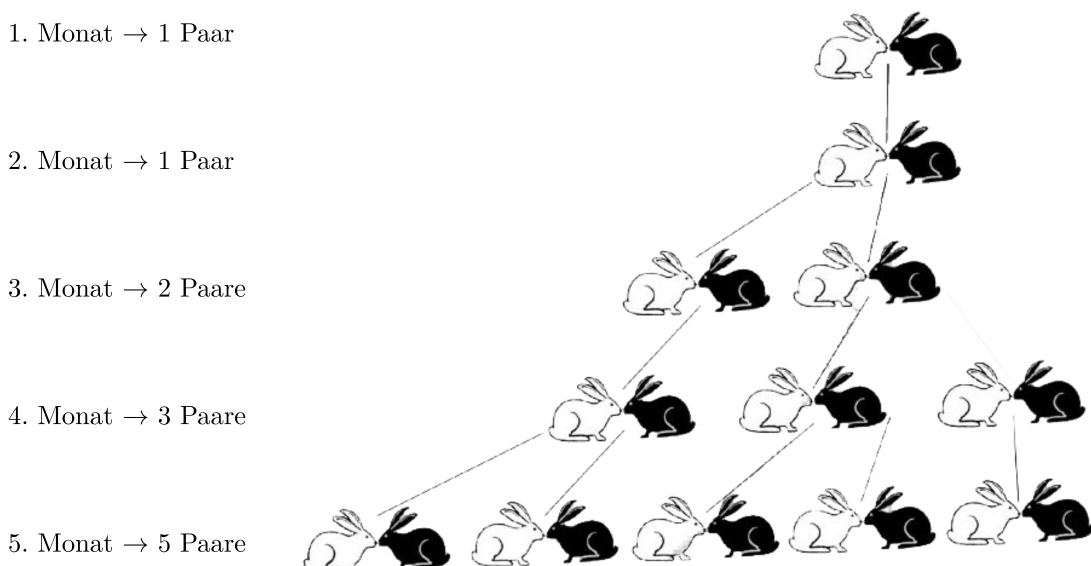


Tabelle 12.1.: Vermehrung der idealen Kaninchenpaare

12.2.1. Wozu Rekursion?

Schaut man sich die obigen Beispiele an, so kann man sich berechtigter Weise fragen, wozu Rekursion gut sein soll. Betrachten wir ein anderes Beispiel.

Beispiel 12.15.
Ein Freund hat uns ein frischgeborenes Kaninchenpaar (♂,♀) geschenkt. Netterweise hat er uns vorgewarnt, dass das Paar in einem Monat geschlechtsreif wird und dann jeden Monat ein neues Kaninchenpaar werfen wird. Mit einem besorgten Blick auf unseren teuren, begrenzten Frankfurter Wohnraum und den schmalen Geldbeutel fragen wir uns:
„Wie viele Kaninchenpaare werden wir in einem Jahr haben, wenn sich die Kaninchen ungehindert vermehren können und keines stirbt? “

Im ersten Monat wächst unser erstes Kaninchenpaar (♂,♀) heran. Somit haben wir zum Ende des 1. Monats immernoch 1 Kaninchenpaar. Nun gebärt dieses Kaninchenpaar am Ende des 2. Monats ein weiteres Kaninchenpaar, also haben wir zu Beginn des 3. Monats bereits 2 Kaninchenpaare. Am Ende des 3. Monats gebärt unser ursprüngliches Kaninchenpaar dann erneut ein Kaninchenpaar, das zweite Kaninchenpaar aber wächst ersteinmal heran. Daher haben wir zu Beginn des 4. Monats insgesamt 3 Kaninchenpaare. Unser ursprüngliches, das Paar welches am Ende des 2. Monats geboren wurde und das Paar das gerade erst geboren wurde. Am Ende des 4. Monats gebären dann sowohl unser urspüngliches Kaninchenpaar, als auch unser Kaninchenpaar das am Ende des 2. Monats geboren wurde je ein Kaninchenpaar. Zu Beginn des 5. Monats sind wir also schon stolze Besitzer von 5 Kaninchenpaaren. Von denen werden alle Paare trächtig, die älter als 2 Monate sind. Das sind 3, also haben wir zu Beginn des 6. Monats 8 Kaninchenpaare, usw... Wir ahnen schon Böses, was in den ersten 2 Monaten so harmlos anfing, wächst uns über den Kopf.

Können wir eine Funktion $fib : \mathbb{N} \to \mathbb{N}$ angeben, die uns sagt, wie viele Kaninchenpaare wir zu Beginn des n-ten Monats haben werden?
Zu Beginn des 1. Monats haben wir 1 Paar, zu Beginn des 2. Monats haben wir immernoch nur ein Paar, zu Beginn des n-ten Monats haben wir immer so viele Kaninchenpaare, wie frisch geboren wurden, zusätzlich zu denen, die wir zu Beginn des vorigen Monats bereits hatten. Da alle Paare, die mindestens 2 Monate alt sind, also alle Paare, die wir vor 2 Monaten bereits hatten, ein

Kaninchenpaar gebären, ist die Anzahl der neugeborenen Paare gleich der Anzahl der Paare vor 2 Monaten. Somit ergibt sich:

Definition 12.16 (Fibonacci-Folge)**.**

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

Fibonacci Folge

Zu diesem Schluss kam 1202 bereits der italienische Mathematiker *Leonardo Fibonacci*, als er über eben dieses Problem nachdachte. Nach ihm wurde die durch $fib(n)$ definierte Zahlenfolge benannt.

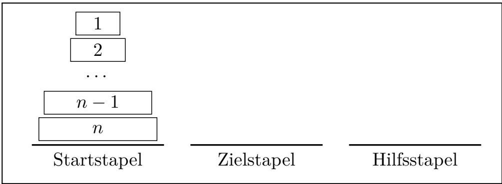
Beispiel 12.17.
Betrachten wir ein weiteres Beispiel:

Türme von Hanoi

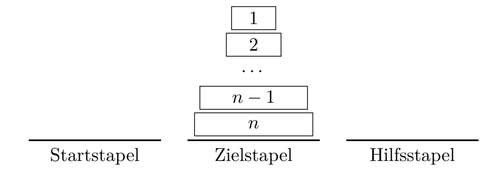
Bei dem Spiel *Türme von Hanoi*, muss ein Stapel von n, nach Größe sortierter Scheiben, von einem *Startstapel* mithilfe eines *Hilfsstapels* auf einen *Zielstapel* transportiert werden (siehe Abbildung). Dabei darf

Türme von Hanoi

- immer nur eine Scheibe bewegt werden und
- es darf nie eine größere auf einer kleineren Scheibe liegen.

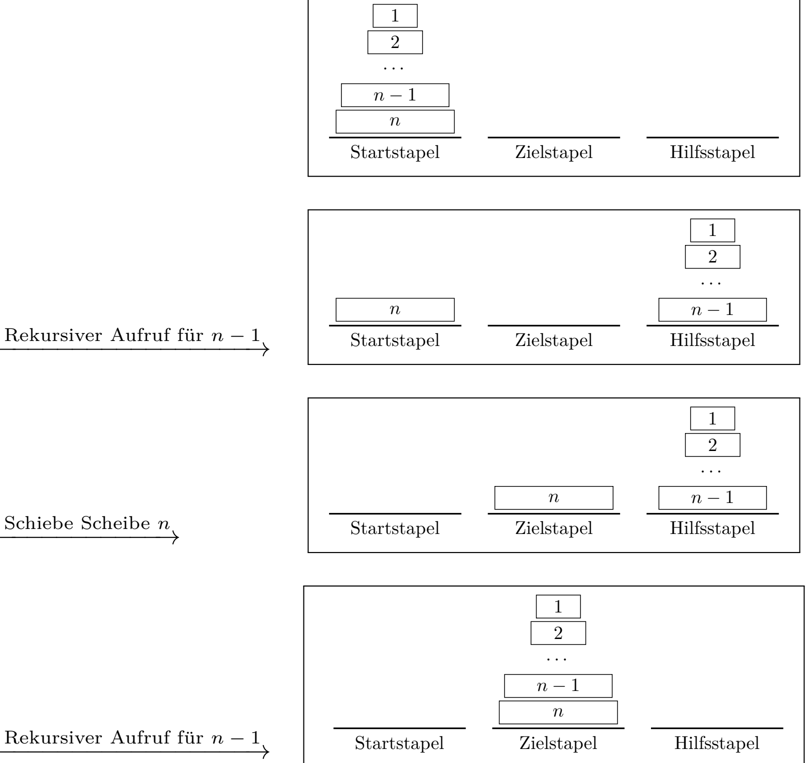


Ausgangssituation



Zielsituation

Für $n = 1$ Scheibe ist die Lösung des Problems noch trivial, auch für $n = 2$ Scheiben ist die Lösung offensichtlich. Für $n = 3$ Scheiben muss man schon etwas „rumprobieren“, um das Problem zu lösen. Für noch mehr Scheiben benötigen wir eine Lösungsstrategie. Bei der Betrachtung des Problems fällt auf, dass wir, um n Scheiben vom Startstapel auf den Zielstapel zu transportieren, zunächst die oberen $n - 1$ Scheiben vom Startstapel auf den Hilfsstapel transportieren müssen. Dann brauchen wir lediglich die n-te Scheibe vom Startstapel auf den Zielstapel zu legen und dann die $n - 1$ Scheiben vom Hilfsstapel auf den Zielstapel (siehe Abbildung).



Um $n-1$ Scheiben vom Start- zum Hilfsstapel zu bewegen, müssen wir die oberen $n-2$ Scheiben vom Start- zum Zielstapel bewegen. Um das zu tun, müssen wir $n-3$ Scheiben vom Start- auf den Hilfsstapel bewegen, usw. Wir verkleinern also sukzessive die Problemgröße und wählen Start-, Ziel- und Hilfsstapel passend, bis wir lediglich eine Scheibe vom Start- zum Zielstapel transportieren müssen. Das Problem können wir sofort lösen.

So haben wir mithilfe von Rekursion rasch eine Lösung für das Problem gefunden.

Algorithmus 12.18.

```
1 bewege n Scheiben von Start zu Ziel, benutze Hilf
2   falls n>1
3     bewege n-1 Scheiben von Start zu Hilf, benutze Ziel
4   verschiebe Scheibe n von Start auf Ziel
5   falls n>1
6     bewege n-1 Scheiben von Hilf zu Ziel, benutze Start
```

Würden wir nun unser Leben darauf verwetten, dass das immer funktioniert? Vielleicht beweisen wir vorher lieber die Korrektheit.

Korrektheit

Satz 12.19.
Algorithmus 12.18 löst das Problem der „Türme von Hanoi".

Beweis durch vollständige Induktion
Sei $n \in \mathbb{N}$ und sei die Aufgabe einen Stapel mit n Scheiben von Stapel A (start) nach B (ziel) zu transportieren, wobei Stapel C (hilf) als Hilfsstapel verwendet werden darf.

Induktionsanfang: $n = 1$
Behauptung: Der Algorithmus arbeitet korrekt für $n = 1$.
Beweis: Da $n = 1$ ist, führt der Aufruf von `bewege 1 Scheibe von A nach B, benutze C` dazu, dass lediglich Zeile 5 ausgeführt wird. Die Scheibe wird also von *start* auf *ziel* verschoben. Genau das sollte geschehen.
Zur Sicherheit und Übung betrachten wir auch noch $n = 2$
Behauptung: Der Algorithmus arbeitet korrekt für $n = 2$.
Beweis:

```
1 bewege 2 Scheiben von A nach B, benutze C   //Aufruf mit n
2   bewege 1 Scheibe von A nach C, benutze B  //Z.3, 1.Aufruf mit (n-1)
3     verschiebe Scheibe 1 von A nach C       //Aufruf mit (n-1), Z.5
4   verschiebe Scheibe 2 von A nach B     //Aufruf mit n, Z.5
5   bewege 1 Scheibe von C nach B, benutze A  //Aufruf mit n, Z.7
6     verschiebe Scheibe 1 von C nach B       //2.Aufruf mit (n-1), Z.5
```

Die oberste Scheibe wird also von A auf Stapel C gelegt (3), dann wird die untere Scheibe von Stapel A auf B gelegt (4) und zum Schluss die kleinere Scheibe von Stapel C auf B gelegt (6). Somit ist der Stapel mit $n = 2$ Scheiben unter Beachtung der Regeln von A nach B verschoben worden.

Induktionsschritt: $n \to n + 1$
Induktionsvoraussetzung: Der Algorithmus arbeitet korrekt für $n \geq 1$.
Induktionsbehauptung: Wenn der Algorithmus für n korrekt arbeitet, dann auch für $n + 1$.
Beweis:

```
1 bewege n+1 Scheiben von A nach B, benutze C //Aufruf mit n+1
2   bewege n Scheiben von A nach C, benutze B //Z.3, Aufruf mit n
3   verschiebe Scheibe n+1 von A auf B        //Z.5
4   bewege n Scheiben von C nach B, benutze A //Z.7, Aufruf mit n
```

Zuerst werden also die obersten n Scheiben von Stapel A nach Stapel C transportiert (2). Laut Induktionsvoraussetzung arbeitet der Algorithmus für n Scheiben korrekt und transportiert den Stapel mit n Scheiben von A nach C. Dann wird Scheibe $n + 1$ von A nach B verschoben (3),

anschließend werden die n Scheiben auf Stapel C auf Stapel B transportiert (4). Das verstößt nicht gegen die Regeln, da

1. die Scheibe $n+1$ größer ist als alle anderen Scheiben, denn sie war zu Beginn die unterste Scheibe. Somit kann die Regel, dass niemals ein größere auf einer kleineren Scheibe liegen darf, nicht verletzt werden, da B frei ist und Scheibe $n+1$ somit zuunterst liegt.
2. der Algorithmus für n Scheiben korrekt arbeitet und somit der Stapel mit n Scheiben korrekt von C nach B verschoben wird.

Damit arbeitet der Algorithmus auch für $n+1$ korrekt. □

Hier ist im Induktionsschritt gleich zweimal die Induktionsvoraussetzung angewendet worden. Einmal, um zu argumentieren, dass die ersten n Scheiben korrekt von A nach C transportiert werden, und dann, um zu argumentieren, dass diese n Scheiben auch korrekt von C nach B transportiert werden können.

Anzahl der Spielzüge

Es drängt sich einem schnell der Verdacht auf, dass das Spiel „Die Türme von Hanoi“ ziemlich schnell ziemlich viele Versetzungen einer Scheibe (oder Spielzüge) benötigt, um einen Stapel zu versetzen. Um zu schauen, ob sich eine Gesetzmäßigkeit feststellen lässt, zählt man zunächst die Spielzüge für kleine Werte von n.

$n=1$	Schiebe 1 von A nach B	1 Zug
$n=2$	1$\curvearrowright$ C, 2$\curvearrowright$ B, 1$\curvearrowright$ B	3 Züge
$n=3$	1$\curvearrowright$ B, 2$\curvearrowright$ C ,1$\curvearrowright$ C, 3$\curvearrowright$ B, 1$\curvearrowright$ A, 2$\curvearrowright$ B, 1$\curvearrowright$ B	7 Züge

Nach einigem Nachdenken kommt man auf die Gesetzmäßigkeit:

Satz 12.20.
Um n Scheiben von einem Stapel zu einem anderen zu transportieren, werden mindestens $2^n - 1$ *Spielzüge benötigt.*

Beweis durch vollständige Induktion
Induktionsanfang: $n = 1$
Behauptung: Um eine Scheibe von einem Stapel auf einen anderen zu transportieren, wird mindesten $2^1 - 1 = 2 - 1 = 1$ Spielzug benötigt.
Beweis: Setze die Scheibe vom Startstapel auf den Zielstapel. Das entspricht einem Spielzug und man ist fertig.

Induktionsschritt: $n \to n+1$
Induktionsvoraussetzung: Um n Scheiben von einem Stapel auf einen anderen zu transportieren, werden mindestens $2^n - 1$ Spielzüge benötigt.
Induktionsbehauptung: Um $n+1$ Scheiben von einem Stapel auf einen anderen zu transportieren, werden mindestens $2^{n+1} - 1$ Spielzüge benötigt.
Beweis: Um $n+1$ Scheiben von einem Stapel A auf einen Stapel B zu transportieren, transportiert man nach Algorithmus 12.18 zunächt n Scheiben von Stapel A auf Stapel C, dann Scheibe $n+1$ von Stapel A nach Stapel B und zum Schluss die n Scheiben von Stapel C nach Stapel B. Nach der Induktionsvoraussetzung benötigt das Versetzen von n Scheiben von Stapel A auf Stapel C mindestens $2^n - 1$ Spielzüge, das Versetzen der Scheibe $(n+1)$, 1 Spielzug und das Versetzen der n Scheiben von C nach B nochmals mindestens $2^n - 1$ Spielzüge (Induktionsvoraussetzung). Das sind insgesamt mindestens:

$$2^n - 1 + \mathbf{1} + 2^n - 1 = 2 \cdot (2^n - 1) + 1 = 2 \cdot 2^n - 2 + 1 = 2^{n+1} - 1$$

Spielzüge. □

A. Appendix

A.1. Unix und Linux

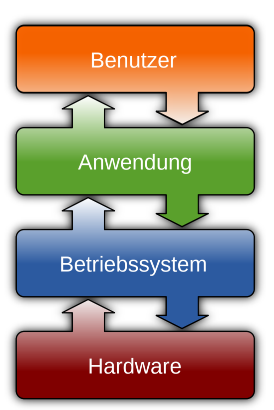


Abbildung A.1.

Unix ist ein Betriebssystem und wurde 1969 in den *Bell Laboratories* (später *AT&T*) entwickelt. Als Betriebssytem verwaltet Unix den Arbeitsspeicher, die Festplatten, CPU, Ein- und Ausgabegeräte eines Computers und bildet somit die Schnittstelle zwischen den Hardwarekomponenten und der Anwendungssoftware (z.B. Office) des Benutzers (Abb. A.1). Da seit den 80er Jahren der Quellcode von Unix nicht mehr frei verfügbar ist und bei der Verwendung von Unix hohe Lizenzgebühren anfallen, wurde 1983 das GNU-Projekt (**G**NU's **N**ot **U**nix) ins Leben gerufen, mit dem Ziel, ein freies Unix-kompatibles Betriebssystem zu schaffen. Dies gelang 1991 mithilfe des von Linus Torvalds programmierten Kernstücks des Betriebssystems, dem Linux-Kernel. GNU Linux ist ein vollwertiges, sehr mächtiges Betriebssystem.

Unix

GNU Linux

Da der Sourcecode frei zugänglich ist, kann jeder seine eigenen Anwendung und Erweiterungen programmieren und diese veröffentlichen. Es gibt vielzählige Linux-Distributionen (Red Hat, SuSe, Ubuntu,...), welche unterschiedliche Software Pakete zur Verfügung stellen. Auf den Rechnern der **R**echner**b**etriebsgruppe **I**nformatik der Goethe Universität (RBI) ist Red Hat Linux installiert.

Linux stellt seinen Benutzern sog. *Terminals* zur Verfügung, an denen gearbeitet werden kann. Ein Terminal ist eine Schnittstelle zwischen Mensch und Computer. Es gibt *textbasierte* und *graphische* Terminals.

Terminal

Textbasierte Terminals stellen dem Benutzer eine Kommandozeile zur Verfügung. Über diese kann der Benutzer, durch Eingabe von Befehlen, mithilfe der Computertastatur, mit Programmen, die über ein CLI (**c**ommand **l**ine **i**nterface) verfügen, interagieren. Einige solcher Programme werden wir gleich kennenlernen. Das Terminal stellt Nachrichten und Informationen der Programme in Textform auf dem Bildschirm dar. Der Nachteil textbasierter Terminals ist für Anfänger meist, dass die Kommandozeile auf eine Eingabe wartet. Man muss den Befehl kennen, den man benutzen möchte, oder wissen, wie man ihn nachschauen kann. Es gibt nicht die Möglichkeit sich mal irgendwo „durchzuklicken“. Der Vorteil textbasierter Terminals ist, dass die Programme mit denen man arbeiten kann häufig sehr mächtig sind. Ein textbasiertes Terminal bietet sehr viel mehr Möglichkeiten, als ein graphisches Terminal.

textbasiertes Terminal
CLI

Graphische Terminals sind das, was die meisten Menschen, die heutzutage Computer benutzen, kennen. Ein graphisches Terminal lässt sich mit einer Computermaus bedienen. Der Benuter bedient die Programme durch Klicken auf bestimmte Teile des Bildschirms, welche durch Icons (kleine Bilder) oder Schriftzüge gekennzeichnet sind. Damit das funktionert, benötigen die Programme eine graphische Benutzeroberfläche, auch GUI (**g**raphical **u**ser **i**nterface) genannt. Auf den Rechnern der RBI findet man, unter anderem, die graphischen Benutzeroberflächen Gnome und KDE.

graphisches Terminal
GUI

Ein einzelner Rechner stellt sieben, voneinander unabhängige Terminals zur Verfügung. Mit den Tastenkombinationen Strg + Alt + F1 , Strg + Alt + F2 bis Strg + Alt + F7 kann

zwischen den sieben Terminals ausgewählt werden. Tastatureigaben werden immer an das angezeigte Terminal weitergeleitet. In der Regel ist lediglich das durch die Tastenkombination Strg + Alt + F7 erreichbare Terminal graphisch. Alle andern Terminals sind textbasiert. Auf den RBI-Rechnern ist das graphische Terminal als das aktive Terminal eingestellt, sodass ein Benutzer der nicht Strg + Alt + F1 , . . . , Strg + Alt + F6 drückt, die textbasierten Terminals nicht zu Gesicht bekommt.

A.1.1. Dateien und Verzeichnisse

Eines der grundlegenden UNIX-Paradigmen ist: „Everything is a file". Die Zugriffsmethoden für Dateien, Verzeichnisse, Festplatten, Drucker, etc. folgen alle den gleichen Regeln, grundlegende Kommandos sind universell nutzbar. Über die Angabe des Pfades im UNIX-Dateisystem lassen sich Quellen unabhängig von ihrer Art direkt adressieren. So erreicht man beispielsweise mit `/home/hans/protokoll.pdf` eine persönliche Datei des Benutzers „hans", ein Verzeichnis auf einem Netzwerklaufwerk mit `/usr`, eine Festplattenpartition mit `/dev/sda1/` und sogar die Maus mit `/dev/mouse`.

Dateibaum
Verzeichnis

Das UNIX-Betriebssystem verwaltet einen *Dateibaum*. Dabei handelt es sich um ein virtuelles Gebilde zur Datenverwaltung. Im Dateibaum gibt es bestimmte Dateien, welche *Verzeichnisse* (engl.: *directories*) genannt werden. Verzeichnisse können andere Dateien (und somit auch Verzeichnisse) enthalten. Jede Datei muss einen Namen haben, dabei wird zwischen Groß- und Kleinschreibung unterschieden. `/home/hans/wichtiges` ist ein anderes Verzeichnis als `/home/hans/Wichtiges`. 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 Namen „/".

übergeordnetes Verzeichnis
Wurzelverzeichnis

Beispiel A.1 (Ein Dateibaum)**.**
Nehmen wir an, das Wurzelverzeichnis enthält zwei Verzeichnisse `Alles` und `Besser`. Beide Verzeichnisse enthalten je ein Verzeichnis mit Namen `Dies` und `Das`. In Abbildung A.2 lässt sich der Baum erkennen. Die Bäume mit denen man es meist in der Informatik zu tun hat, stehen auf dem Kopf. Die Wurzel befindet sich oben, die Blätter unten.

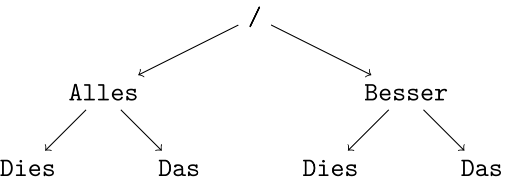


Abbildung A.2.: Ein Dateibaum

Pfad
absolut
relativ
slash

Die beiden Verzeichnisse mit Namen `Dies` lassen sich anhand ihrer Position im Dateibaum leicht auseinanderhalten. Den Weg durch den Dateibaum zu dieser Position nennt man *Pfad* (engl::*path*). Gibt man den Weg von der Wurzel aus an, so spricht man vom *absoluten* Pfad. Gibt man den Weg vom Verzeichnis aus an, in dem man sich gerade befindet, so spricht man vom *relativen* Pfad. Die absoluten Pfade zu den Verzeichnissen mit Namen `Dies` lauten `/Alles/Dies` und `/Besser/Dies`. Unter UNIX/LINUX dient der Schrägstrich `/` (engl.:*slash*) als Trennzeichen zwischen Verzeichnissen. Im Gegensatz zu Windows, wo dies durch den back-slash \ geschieht. Wenn wir uns im Verzeichnis `/Besser` befinden, so können die Unterverzeichnisse mit *Dies* und `Das` direkt adressiert werden. Das Symbol `..` bringt uns ins übergeordnete Verzeichnis, in diesem Fall das Wurzelverzeichnis. Somit erreichen wir aus dem das Verzeichnis `Alles` aus dem Verzeichnis `Besser` über den relativen Pfad `../Alles`. Befinden wir uns in Verzeichnis `/Alles/Dies` so erreichen wir das Verzeichnis `/Besser/Das` über den relativen Pfad `../../Besser/Das`.

Dateizugriffsrechte

Unter UNIX können auch die Zugriffsrechte einzelner Benutzer auf bestimmte Dateien verwaltet werden. Dabei wird unterschieden zwischen Lese-(*r*ead), Schreib-(*w*rite) und Ausführrechten (*x* execute). Für die Vergabe dieser Rechte, wird zwischen dem Besitzer (*o*wner) der Datei, einer Gruppe (*g*roup) von Benutzern und allen Nutzern, die nicht zu der Gruppe gehören (*o*ther), unterschieden. Um bestimmten Nutzern Zugriffsrechte für bestimmte Dateien zu erteilen, können diese Nutzer zu einer Gruppe zusammengefasst werden. Dann können allen Mitgliedern der Gruppe Zugriffsrechte für diese Dateien erteilt werden.

A.1.2. Login und Shell

Um an einem Terminal arbeiten zu können, muss sich der Benutzer zunächst anmelden. Dies geschieht durch Eingabe eines Benutzernamens und des zugehörigen Passwortes. Diesen Vorgang nennt man „sich *Einloggen*“. Loggt man sich an einem textbasierten Terminal ein, so startet nach dem Einloggen automatisch eine (Unix)-*Shell*. Dies ist die traditionelle Benutzerschnittstelle unter UNIX/Linux. Der Benutzer kann nun über die Kommandozeile Befehle eintippen, welche der Computer sogleich ausführt. Wenn die Shell bereit ist Kommandos entgegenzunehmen, erscheint eine *Eingabeaufforderung* (engl.: *prompt*). Das ist eine Zeile, die Statusinformationen, wie den Benutzernamen und den Namen des Rechners auf dem man eingeloggt ist, enthält und mit einem blinkenden *Cursor* (Unterstrich) endet.

Einloggen
Shell
Eingabe-aufforderung

Benutzer, die sich an einem graphischen Terminal einloggen, müssen zunächst ein virtuelles text-basiertes Terminal starten, um eine Shell zu Gesicht zu bekommen. Ein virtuelles textbasiertes Terminal kann man in der Regel über das Startmenü, Unterpunkt „Konsole“ oder „Terminal“, gestartet werden. Unter der graphischen Benutzeroberfläche KDE kann man solch ein Terminal auch starten, indem man mit der rechten Maustaste auf den Desktop klickt und im erscheinenden Menü den Eintrag „Konsole“ auswählt (Abb.: B.2).

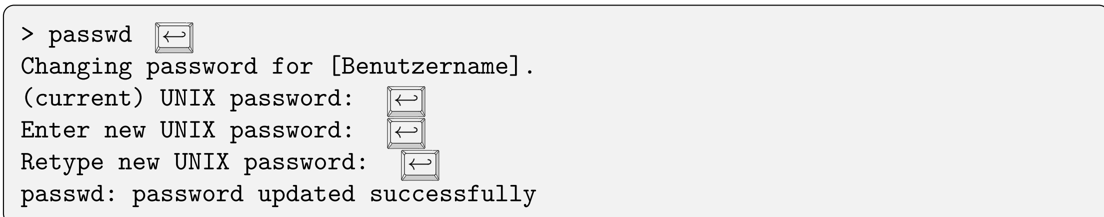
A.1.3. Befehle

Es gibt unzählige Kommandos die die Shell ausführen kann. Wir beschränken uns hier auf einige, die wir für besonders nützlich halten. Um die Shell aufzuforden den eingetippten Befehl auszuführen, muss die *Return*-Taste (↩) betätigt werden. Im Folgenden sind Bildschirmein- und -ausgaben in `Schreibmaschinenschrift` gegeben und

> in grau hinterlegten Kästen mit durchgezogenen Linien und runden Ecken zu finden.

`passwd:` ändert das Benutzerpasswort auf dem Rechner auf dem man eingeloggt ist. Nach Eingabe des Befehls, muss zunächt einmal das alte Passwort eingegeben werden. Dannach muss zweimal das neue Passwort eingegeben werden. Dieser Befehl ist ausreichend, wenn der Account lediglich auf einem Rechner existiert, z.B. wenn man Linux auf seinem privaten Desktop oder Laptop installiert hat.

Passwort ändern



```
> passwd ↩
Changing password for [Benutzername].
(current) UNIX password: ↩
Enter new UNIX password: ↩
Retype new UNIX password: ↩
passwd: password updated successfully
```

Für den RBI-Account wird folgender Befehl benötigt.

Netzwerkpasswort

yppasswd: ändert das Passwort im System und steht dann auf allen Rechnern des Netzwerks zur Verfügung.

```
> passwd ↩
Changing NIS account information for [Benutzername] on [server].
Please enter old password: ↩
Changing NIS password for [Benutzername] on [server].
Please enter new password: ↩
Please retype new password: ↩

The NIS password has been changed on [server].
```

Arbeitsverzeichnis
Homeverzeichnis

pwd (*print working directory*): gibt den Pfad des Verzeichnisses aus, in dem man sich gerade befindet. Dieses Verzeichnis wird häufig auch als *„aktuelles Verzeichnis“*, oder *„Arbeitsverzeichnis“* bezeichnet. Unmittelbar nach dem Login, befindet man sich immer im *Homeverzeichnis*. Der Name des Homeverzeichnis ist der gleiche wie der Benutzername. Hier werden alle persönlichen Dateien und Unterverzeichnisse gespeichert.

```
> pwd ↩
/home/ronja/Lernzentrum/Vorkurs/WS1314/Skript/
```

whoami : gibt den Benutzernamen aus.

```
> whoami ↩
ronja
```

hostname: gibt den Rechnernamen, auf dem man eingeloggt ist, aus.

```
> hostname ↩
nash
```

Verzeichnis erstellen
Argument

mkdir: (*make directory*): mit diesem Befehl wird ein neues Verzeichnis (Ordner) angelegt. Dieser Befehl benötigt als zusätzliche Information den Namen, den das neue Verzeichnis haben soll. Dieser Name wird dem Befehl als *Argument* übergeben. Zwischen Befehl und Argument muss immer ein Leerzeichen stehen. Der folgende Befehl legt in dem Verzeichnis, in dem sich der Benutzer gerade befindet, ein Verzeichnis mit Namen „Zeug“ an.

```
> mkdir Zeug ↩
```

cd (*change directory*): wechselt das Verzeichnis. Wird kein Verzeichnis explizit angegeben, so wechselt man automatisch in das Homeverszeichnis.

cd ..: wechselt in das nächsthöhere Verzeichnis. Dabei wird `..` als Argument übergeben.

```
> pwd ↩
/home/ronja/Lernzentrum/Vorkurs/
> mkdir Wichtiges ↩
> cd Wichtiges ↩
> pwd ↩
 /home/ronja/Lernzentrum/Vorkurs/Wichtiges/
> cd .. ↩
> pwd ↩
/home/ronja/Lernzentrum/Vorkurs/
```

ls (*list*): zeigt eine Liste der Namen der Dateien und Verzeichnisse, die sich im aktuellen Verzeichnis befinden. Dateien die mit einem „.“ anfangen, meist Systemdateien, werden nicht angezeigt.

ls -a : zeigt eine Liste der Namen *aller* (engl.: *all*) Dateien und Verzeichnisse, die sich im aktuellen Verzeichnis befinden an. Auch Dateien die mit einem „.“ anfangen, werden angezeigt. Bei dem `-a` handelt es sich um eine *Option* , die dem Befehl übergeben wird. Optionen werden mit einem oder zwei Bindestrichen eingeleitet. Dabei können mehrer Optionen gemeinsam übergeben werden, ohne dass erneut ein Bindestrich eingegeben werden muss. Wird dem Kommando als Argument der absolute oder relative Pfad zu einem Verzeichnis angegeben, so werden die Namen der in diesem Verzeichnis enthaltenen Dateien angezeigt.

Option

```
> ls ↩
Wichtiges  sichtbareDatei1.txt  sichtbareDatei2.pdf
> ls -a ↩
.  ..  Wichtiges  sichtbareDatei1.txt  sichtbareDatei2.pdf
> ls -a Wichtiges ↩
.  ..
```

ls -l: zeigt eine Liste der Namen und Zusatzinformationen (`l` für engl.: *long*) der Dateien und Verzeichnisse, die sich im aktuellen Verzeichnis befinden. Die Einträge ähneln dem Folgenden.

```
> ls -l ↩
-rw-r--r-- 1    alice    users   2358  Jul 16   14:23 protokoll.pdf
```

Von rechts nach links gelesen, sagt uns diese Zeile, dass die Datei „protokoll.pdf“ um 14:23 Uhr am 15. Juli diesen Jahres erstellt wurde. Die Datei ist 2358 Byte groß, und gehört der Gruppe „users“, insbesondere gehört sie der Benutzerin „alice“ und es handelt sich um eine (1) Datei. Dann kommen 10 Positionen an denen die Zeichen `-`, `r` oder `w`, stehen. Der Strich (`-`) an der linkesten Position zeigt an, dass es sich hierbei um eine gewöhnliche Datei handelt. Bei einem Verzeichnis würde an dieser Stelle ein `d` (für directory) stehen. Dann folgen die Zugriffsrechte. Die ersten drei Positionen sind für die Zugriffsrechte des Besitzers (engl.: *owner*) der Datei. In diesem Fall darf die Besitzerin alice die Datei lesen (*r*ead) und verändern (*w*rite). Alice darf die Datei aber nicht ausführen (*x* execute). Eine gewöhnliche .pdf-Datei möchte man aber auch nicht ausführen. Die Ausführungsrechte sind wichtig für Verzeichnisse und Programme. Die mittleren drei Positionen geben die Zugriffsrechte der Gruppe (engl.: *group*) an. Die Gruppe users darf hier die Datei lesen, aber nicht schreiben. Die letzen drei Positionen sind für die Zugriffrechte aller andern Personen (engl.: *other*). Auch diesen ist gestattet die Datei zu lesen, sie dürfen sie aber nicht verändern.

Zugriffsrechte

chmod (*change mode*): ändert die Zugriffsberechtigungen auf eine Datei. Dabei muss dem Programm die Datei, deren Zugriffsrechte man ändern will, als Argument übergeben werden. Ferner muss man dem Programm mitteilen, wessen (*u*ser,*g*roup,*o*ther, oder *a*ll) Rechte man wie ändern (`+` hinzufügen, `-` wegnehmen) will.

```
> chmod go +w protokoll.pdf ↩
```

Dieser Befehl gibt der Gruppe `g` und allen anderen Nutzern `o` Schreibrechte `+w` für die Datei `protokoll.pdf`. Vermutlich ist es keine gute Idee, der ganzen Welt die Erlaubnis zu erteilen die Datei zu ändern.

```
> chmod o -rw protokoll.pdf ↩
```

nimmt allen anderen Nutzern `o` die Schreibrechte wieder weg `-w` und nimmt ihnen auch die Leserechte `-r`.

Alternativ kann die gewünschte Änderung auch als Oktalzahl eingegeben werden. Für die Erkärung dazu verweisen wir auf das Internet, oder die *man-page*, s.u.

`man` (*manual*): zeigt die Hilfe-Seiten zu dem, als Argument übergebenen, Kommando an.

```
ronja@nash: ~
MKDIR(1)                         User Commands                        MKDIR(1)

NAME
       mkdir - make directories

SYNOPSIS
       mkdir [OPTION]... DIRECTORY...

DESCRIPTION
       Create the DIRECTORY(ies), if they do not already exist.
 Manual page mkdir(1) line 1 (press h for help or q to quit)
```

Abbildung A.3.: man page des Befehls `mkdir`

A.1.4. History und Autovervollständigung

History

Ein sehr nützliches Hilfmittel beim Arbeiten mit der Shell ist die *history*. Alle Befehle, die man in der Shell eingibt, werden in der history gespeichert. Mit den Cursortasten ↑ und ↓ kann man in der history navigieren. ↑ holt den zuletzt eingegebenen Befehl in die Eingabezeile, ein erneutes Drüchen von ↑ den vorletzen, usw. ↓ arbeitet in die andere Richtung, also z.B. vom vorletzen Befehl zum zuletzt eingegebenen Befehl. Mit den Cursortasten ← und →, kann man innerhalb des Befehls navigieren, um Änderungen vorzunehmen.

Autovervollständigung

Ein weiteres nützliches Hilfmittel ist die *Autovervollständigung*. Hat man den Anfang eines Befehls, oder eines Datei- (oder Verzeichnis-) Namens eingegeben, so kann man den Namen durch Betätigen der *Tab*-Taste ⇆ automatisch vervollständigen lassen, solange der angegebene Anfang eindeutig ist. Ist dies nicht der Fall, so kann man sich mit nochmaliges Betätigen der Tab-Taste ⇆, eine Liste aller in Frage kommenden Vervollständigungen anzeigen lassen (Abb. A.4).

```
ronja@nash: ~
ronja@nash:~$ pr
pr              preunzip        printafm        printf
precat          prezip          printenv        protoc
preconv         prezip-bin      printerbanner   prove
prename         print           printer-profile prtstat
ronja@nash:~$ pr
```

Abbildung A.4.: Autovervollständigung für die Eingabe `pr`

B. Kochbuch

B.1. Erste Schritte

Falls Sie an einem Laptop arbeiten, können/müssen Sie Punkte 1 und 3 überspringen.

1. **Login**: Auf dem Bildschirm sollte eine Login-Maske zu sehen sein, ähnlich wie die in Abb. B.1.

Abbildung B.1.: Login-Maske, „(spuele)“ ist in diesem Fall der Rechnername, der ist bei Ihnen anders

   - unter `Login:` muss der Login-Name, den Sie von uns erhalten haben,
   - unter `Password:` muss das Passwort, welches Sie erhalten haben, eingegeben werden.
   - den `Go!`- Button klicken

2. **Shell öffnen**:

RBI-PC: rechte Maustaste irgendwo auf dem Bildschirm klicken, nicht auf einem Icon. Im sich öffnenden Menü auf „Konsole“ klicken (Abb. B.2). Eine Shell öffnet sich (Abb. B.4).

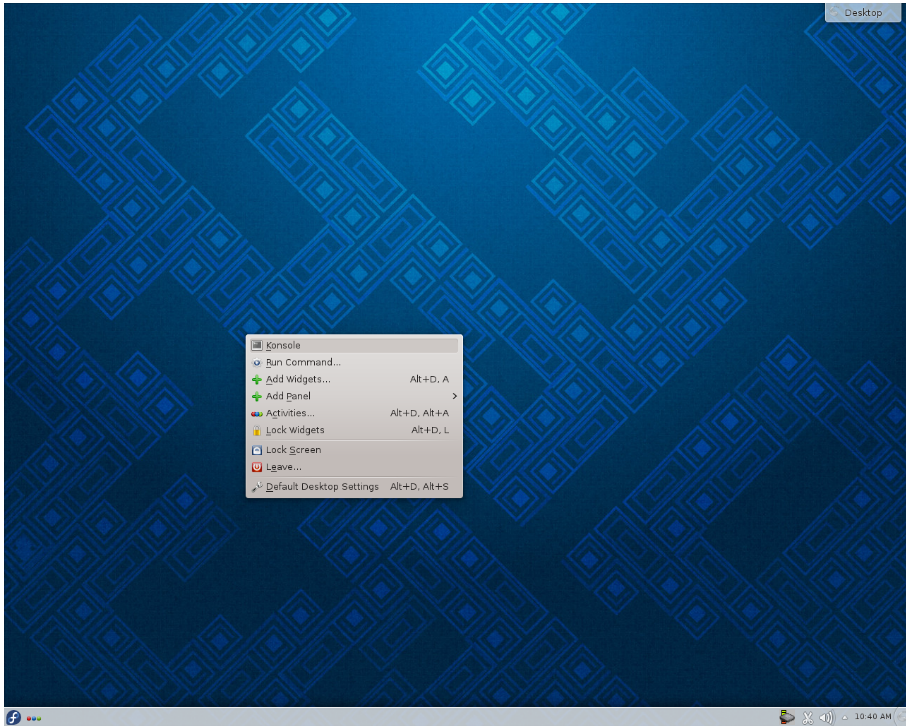


Abbildung B.2.: Terminal öffnen unter KDE

Laptop: mit der linken Maustaste auf das Startsymbol in der linken unteren Ecke des Bildschirms klicken. Das Menü „Systemwerkzeuge“ wählen, im Untermenü „Konsole“ anklicken (Abb. B.3). Eine Shell öffnet sich (Abb. B.4).

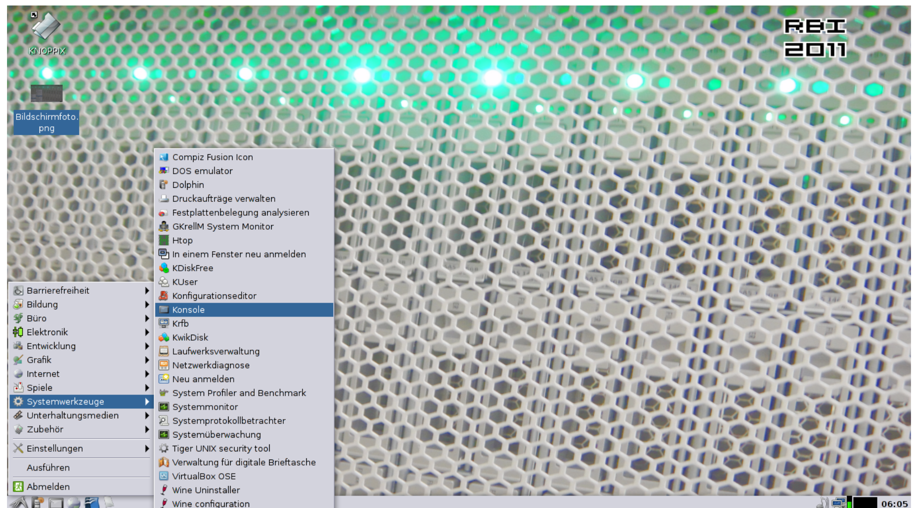


Abbildung B.3.: Terminal öffnen unter Koppix

```
ronja@nash: ~
ronja@nash:~$
```

Abbildung B.4.: Shell; in der Eingabezeile steht `[Benutzername]@[Rechnername]`

3. **Passwort ändern**:

    - in der Shell `yppasswd` eingeben, ↩ drücken
    - in der Shell erscheint die Nachricht: `Please enter old password:`
    - aktuelles Passwort (das das Sie von uns erhalten haben) eingeben. Die eingegebenen Zeichen erscheinen **nicht** auf dem Bildschirm. Es sieht aus, als hätten Sie gar nichts eingegeben. Am Ende ↩ -Taste drücken.
    - in der Shell erscheint die Nachricht: `Please enter new password:`
    - neues Passwort eingeben. Das Passwort muss aus mindestens 6 Zeichen bestehen. Wieder erscheint die Eingabe nicht auf dem Bildschirm. ↩ -Taste drücken.
    - in der Shell erscheint die Nachricht: `Please retype new password:`
    - neues Passwort erneut eingeben, ↩ -Taste drücken.
    - in der Shell erscheint die Nachricht: `The NIS password has been changed on zeus.`

4. **Arbeit beenden**

    RBI-PC: Die RBI-Rechner bitte **niemals** ausschalten. Sie brauchen sich lediglich Auszuloggen. Dies geschieht, indem Sie in der linken, unteren Bildschirmecke auf das Startsymbol klicken, dann im sich öffnenden Menü den Punkt „Leave“ auswählen und auf „Log out“ klicken (Abb.: B.5). Danach erscheint wieder die Login-Maske auf dem Bildschirm.

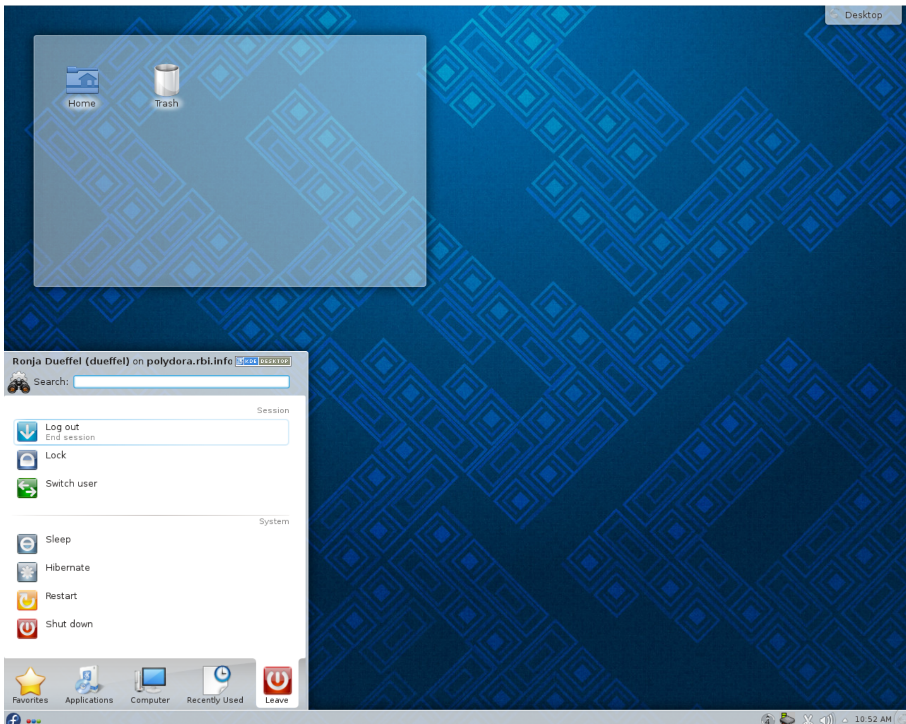


Abbildung B.5.: Ausloggen

Laptop : Den Laptop können Sie, wie Sie es von ihrem Computer zu Hause gewohnt sind, herunterfahren. Dazu auf das Startmenü (unten links) klicken, und „Abmelden“ auswählen (Abb.: B.6). Es öffnet sich ein Dialog, in dem man unterschiedliche Optionen wählen kann. Bitte klicken Sie auf „Herunterfahren“.

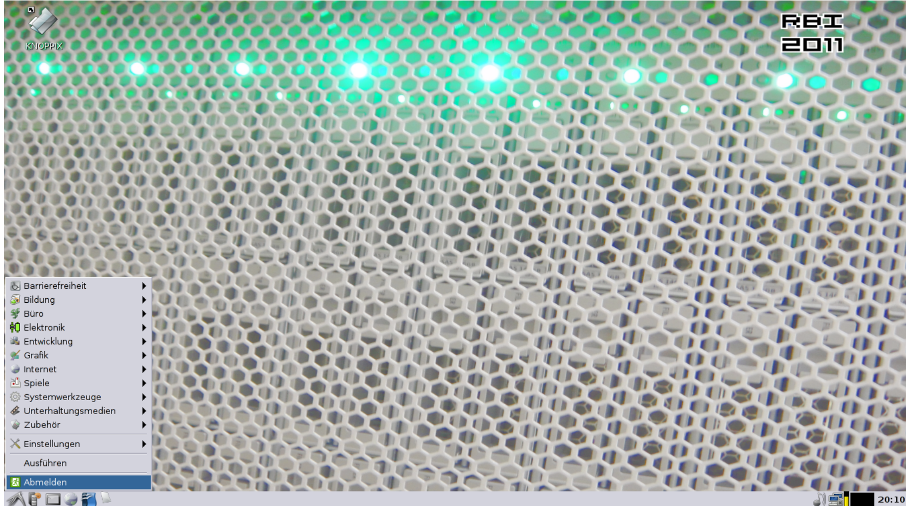


Abbildung B.6.: Herunterfahren

B.2. Remote Login

Auf die Rechner der RBI kann man sich auch über das Internet von einem anderen Rechner (z.B. von zu Hause) aus einloggen. Das ist nützlich, wenn man Programme nicht auf dem eigenen Rechner installieren möchte, oder um Lösungen zu Programmieraufgaben zu testen, denn meist

wird gefordert, dass die Programme auf den RBI-Rechnern laufen. Der Rechner, von dem aus man auf einem RBI-Rechner arbeiten möchte, benötigt hierfür ein *ssh-Client*-Programm. Ferner benötigt man ein *scp-Client*-Programm, um Dateien zwischen den Rechnern auszutauschen. Außerdem muss man wissen, auf welchem RBI-Rechner man sich einloggen möchte. Eine Liste der Rechnernamen findet man auf der RBI-Webseite[1].

B.2.1. Unix-artige Betriebssysteme (Linux, MacOS, etc)

Bei allen UNIX-artigen Betriebssystemen sind ssh- und scp-Client-Programme bereits installiert und können über die Shell gestartet werden.

1. **Austausch der Daten** Mit folgendem Kommando kann man Daten aus dem aktuellen Verzeichnis seines eigenen Rechners, in sein RBI-Homeverzeichnis kopieren.

```
> scp [myprog.py] [benutzername]@[rechnername].rbi.cs.uni-frankfurt.de:~/
```

   Um Daten vom RBI-Account in das aktuelle Verzeichnis auf dem eigenen Rechner zu kopieren, benutzt man dieses Kommando:

```
> scp [benutzername]@[rechnername].rbi.cs.uni-frankfurt.de:~/[datei] .
```

   Der . steht dabei stellvertretend für das aktuelle Verzeichnis. Es ist auch möglich einen relativen oder absoluten Pfad zu einem anderen Verzeichnis anzugeben.

2. **Einloggen**

```
> ssh [benutzername]@[rechnername].rbi.informatik.uni-frankfurt.de
```

   Loggt man sich zum ersten Mal auf diesem Rechner ein, so wird man zunächst gefragt, ob man sich tatsächlich auf diesem unbekannten Rechner einloggen möchte. Bestätigt man dies, so wird man aufgefordert das Passwort einzugeben. Ist das erfolgreich, ist man auf dem RBI-Rechner eingeloggt. Der Name des RBI-Rechners erscheint nun in der Eingabezeile (Abb.: B.7).

```
ronja@nash: ~
ronja@nash:~$ ssh lz_inf@eos.rbi.informatik.uni-frankfurt.de
The authenticity of host 'eos.rbi.informatik.uni-frankfurt.de (141.2.15.135)' ca
n't be established.
RSA key fingerprint is 10:71:22:7e:dd:13:e4:6e:14:26:aa:bf:49:00:32:41.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'eos.rbi.informatik.uni-frankfurt.de,141.2.15.135' (R
SA) to the list of known hosts.
lz_inf@eos.rbi.informatik.uni-frankfurt.de's password:
eos:~>
```

   Abbildung B.7.: Auf einem RBI-Rechner einloggen

3. **Programm starten**

   Ist man auf dem RBI-Rechner eingeloggt, kann das Programm mit dem Befehl

```
> python3 [meinprogramm.py]
```

   gestartet werden.

[1] http://www.rbi.informatik.uni-frankfurt.de/rbi/informationen-zur-rbi/raumplane/

4. **Verbindung beenden**

```
> exit ↵
```

Dieser Befehl schließt die Verbindung zu dem RBI-Rechner.

Abbildung B.8 zeigt das ganze am Beispiel unseres „Hello world!"-Programms welches unter dem Namen `hey.py` im Verzeichnis `/home/ronja/Documents/ME/` gespeichert ist.

```
ronja@nash: ~
ronja@nash:~$ scp /home/ronja/Documents/ME/hey.py lz_inf@eos.rbi.cs.uni-frankfur
t.de:~/
lz_inf@eos.rbi.cs.uni-frankfurt.de's password:
/home/users3/leitung/lz_inf/.bashrc: line 20: /bin/ttytype: No such file or dire
ctory
tset: standard error: Inappropriate ioctl for device

hey.py                                    100%   22     0.0KB/s   00:00
ronja@nash:~$ ssh lz_inf@eos.rbi.cs.uni-frankfurt.de
lz_inf@eos.rbi.cs.uni-frankfurt.de's password:
Last login: Tue Sep  3 15:31:37 2013 from nash.rbi.informatik.uni-frankfurt.de
eos:~> python3 hey.py
Hello world!
eos:~> exit
logout
Connection to eos.rbi.cs.uni-frankfurt.de closed.
ronja@nash:~$
```

Abbildung B.8.: Ein Python-Programm vom eigenen Rechner auf einem RBI-Rechner starten

B.2.2. Windows

1. **Austausch der Daten** Windows-Nutzer müssen sich zunächst ein scp-Client-Programm herunterladen. Eines der populärsten ist *WinSCP*[2]. Um Daten auszutauschen, muss man sich erst einmal mit dem RBI-Rechner verbinden. Das geschieht über die WinSCP-Login-Maske (Abb.: B.9).

[2] http://winscp.net/eng/download.php

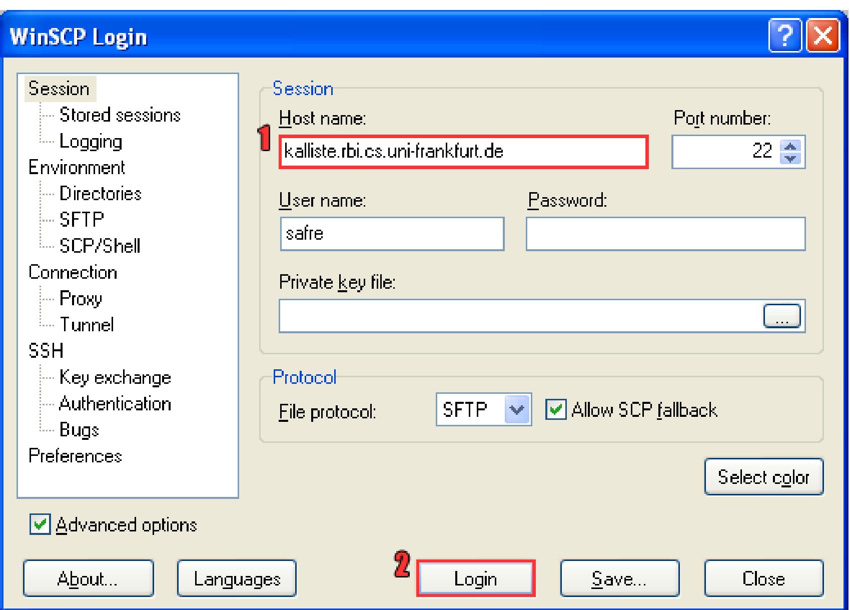


Abbildung B.9.: Das WinSCP-Login-Fenster

Sind die Rechner verbunden, können Daten wie man es vom Dateimanager gewohnt ist, mit drag-and-drop oder über das Menü ausgetauscht werden (Abb. B.10).

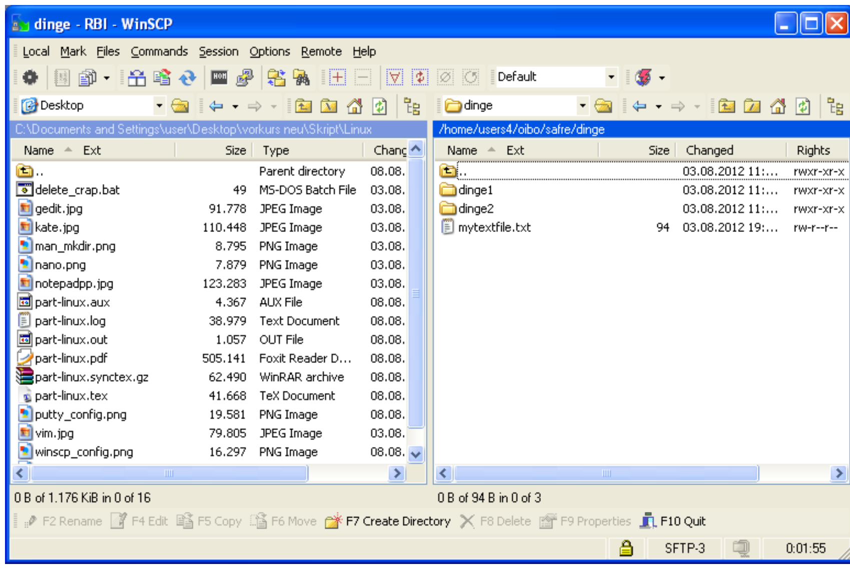


Abbildung B.10.: Das WinSCP-Hauptfenster. Links die Dateien des Windows-Rechners, rechts die Dateien im RBI-Verzeichnis

2. **Einloggen** Um sich von einem Windows-Rechner auf einem RBI-Rechner einzuloggen, benötigt man ein ssh-Client-Programm. *Putty*[3] ist solch ein Programm. Bei der Einstellung des Programms, muss die Adresse des RBI-Rechners angegeben werden, auf dem man sich einloggen möchte (Abb.: B.11).

[3] http://www.chiark.greenend.org.uk/~ sgtatham/putty/download.html

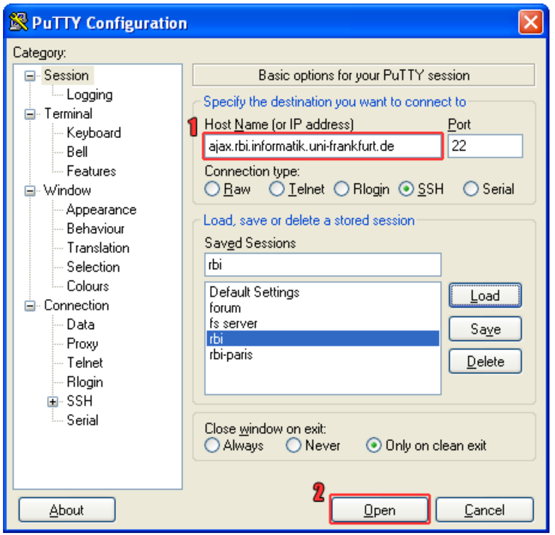
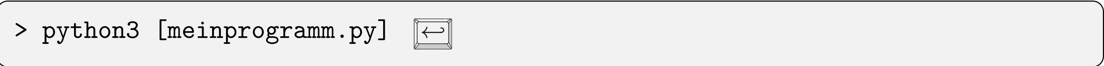


Abbildung B.11.: Einstellung von Putty

Anschließend kann man sich auf dem RBI-Rechner einloggen. Ist der Vorgang erfolgreich, so öffnet sich eine Shell, über die man auf dem RBI-Rechner Kommandos ausführen kann.

3. **Programm starten**

   In der Putty-Shell kann man nun mit folgendem Kommando das Python-Programm starten.



```
> python3 [meinprogramm.py] ↩
```

4. **Verbindung beenden** Der Befehl `exit` schließt die Verbindung zum RBI-Rechner. Wenn die Trennung erfolgreich war, sollte sich das Putty-Fenster selbstständig schießen.

C. Zum Lösen von Übungsaufgaben

C.1. Neu im Studium

Das Studium geht los. Alles funktioniert irgendwie anders als gewohnt und neben den ganzen organisatorischen Fragen rund ums Studium sollen gleichzeitig noch die Vorlesungsthemen gelernt werden. Im Folgenden sollen einige (hoffentlich nützliche) Hinweise zur Bearbeitung der Übungsaufgaben den Start ins Informatik-Studium erleichtern.

Zu jeder Vorlesung gibt es dazugehörige Übungstermine, sogenannte Tutorien. Dort werden die wöchentlich oder zweiwöchentlich fälligen Übungsblätter zusammen mit einem Tutor (meist ein Student aus einem höheren Semester) besprochen und von den Teilnehmern oder dem Tutor vorgerechnet. Dort können auch Fragen zum Vorlesungsstoff oder zum nächsten Übungsblatt gestellt werden.

Die Lösungen werden also in den Tutorien vorgestellt. Dann kann ich also einfach abwarten, ins Tutorium gehen und dort zugucken, wie die richtige Lösung zur Aufgabe auf dem Übungsblatt vorgerechnet wird. Das ist völlig legitim. In fast allen Vorlesungen sind die Übungen freiwillig und keine Zulassungsvoraussetzung zur Klausur. Auch die von den Professoren gehaltenen Vorlesungen sind nur ein freiwilliges Angebot, um letztlich die Klausur erfolgreich bestehen zu können. Es gibt keine Anwesenheitspflichten.

C.1.1. Wozu soll ich die Übungsaufgaben überhaupt (selbst) machen?

Stellt Euch vor, Ihr müsstet die praktische Führerscheinprüfung machen ohne vorher je am Steuer gesessen zu haben. Stattdessen habt Ihr von der Rückbank aus zugeschaut und konntet sehen, wie das Auto fährt.
Warum würdet Ihr trotzdem aller Wahrscheinlichkeit nach durch die Prüfung fallen?

1. „Der Fahrlehrer hatte keine Ahnung!“ oder
2. „Ihr habt nur gesehen, dass das Auto fährt, aber Ihr habt nicht selber an den Pedalen, dem Lenkrad, den Spiegeln und der Gangschaltung geübt.“ oder
3. „Es waren einfach zu wenige Fahrstunden. Wenn ich nur oft genug zuschaue, dann schaffe ich die Prüfung locker!“.

Zwei Antworten waren Quatsch und Ihr wisst sicherlich, welche beiden das waren. Ihr müsst in Fahrstunden selbst am Steuer sitzen, weil Ihr sonst das Autofahren nicht lernt. Und aus dem gleichen Grund solltet Ihr auch die Übungsaufgaben machen. Nur so könnt Ihr die nötigen Fähigkeiten und Kenntnisse erwerben, die für das Bestehen der Prüfung nötig sind. Wenn Ihr von den Tutoren Eure bearbeiteten Übungsaufgaben korrigiert zurück erhaltet, habt Ihr außerdem ein zeitnahes Feedback über Euren Wissenstand. Könnt Ihr den Stoff anwenden und auch Transferaufgaben lösen? Wo liegen Eure Fehler? Was habt Ihr gut gemacht? Welche Schritte waren nötig bis Ihr auf Eure Lösung gekommen seid, welche Fehlversuche habt Ihr unternommen und was habt Ihr aus den Fehlern gelernt? All dies entgeht Euch, wenn Ihr die Übungsaufgaben nicht oder nur halbherzig bearbeitet. Außerdem werdet Ihr nicht nur dadurch belohnt, dass Ihr viel bessere Chancen habt, den Vorlesungsstoff zu verstehen und eine gute Note in der Klausur zu erzielen. Eure Note wird sogar noch besser ausfallen, weil es in vielen Vorlesungen Bonuspunkte für das Lösen und

Vorrechnen der Aufgaben in den Tutorien gibt. So könnt Ihr teilweise bis zu 20% der Klausurpunkte schon vor dem tatsächlichen Klausurtermin sicher haben und braucht am Prüfungstag nur noch weitere 30% der Klausur lösen, um sie zu bestehen. Die genauen Bonuspunkte-Regelungen hängen von der jeweiligen Vorlesung ab und werden vom Dozenten festgelegt. Manchmal werden die Bonuspunkte erst auf eine bestandene Prüfung angerechnet, z. B. indem sich die Note 4.0 um eine Notenstufe auf 3.7 verbessert oder die 1.3 auf eine 1.0.

C.1.2. Was halten wir fest?

Das Lösen der Übungsblätter lohnt sich und ist ein wichtiger Bestandteil eines erfolgreichen Studiums! Hier investierte Zeit ist sehr gut angelegt.

C.1.3. Konkrete Tipps

Betrachtet das folgende Szenario: Pro Woche gibt es zwei Vorlesungstermine, zu denen ein Dozent die Vorlesungsinhalte vorträgt. In der Regel wird kurz ein allgemeines Konzept erläutert, ein Beispiel dazu gebracht und dann mit dem nächsten Konzept fortgefahren. Ihr seid die meiste Zeit in der Zuhörerrolle. Aktiv werdet Ihr erst, wenn es an das Bearbeiten des Übungsblattes geht, das den Stoff aus der aktuellen Vorlesungswoche behandelt. Dafür habt Ihr meistens eine Woche Zeit. Wie solltet Ihr vorgehen, damit Ihr am Ende der Woche alle Aufgaben gelöst habt und Eure Bonuspunkte sammeln könnt?

- Frühzeitig anfangen! Gute Lösungen brauchen Zeit. Im Idealfall habt Ihr Euch das Übungsblatt schon vor der Vorlesung angeschaut. Dann könnt Ihr nämlich ganz gezielt nach den Informationen Ausschau halten, die Ihr für die Aufgaben des Übungsblattes benötigt.
- In der Vorlesung zuhören und Fragen stellen.
- Die Aufgaben vollständig lesen. Nicht bekannte Begriffe nachschlagen.
- zusätzliche Materialien benutzen (Skript, Folien etc.)
- Die Übungsaufgaben gemeinsam in einer Gruppe besprechen und verschiedene Lösungsalternativen vergleichen; am Ende sollte jeder, unabhängig von den anderen, seine eigene Lösung aufschreiben. Insgesamt kann man den Wert einer guten Arbeitsgruppe gar nicht genug betonen.
- Nicht abschreiben! Es klingt besserwisserisch, aber damit betrügt Ihr Euch selbst und lernt nichts. Außerdem setzt Ihr damit Eure Bonuspunkte aufs Spiel.
- Schreibt Eure Lösungen ordentlich auf. Falls Ihr eine unleserliche Handschrift habt, dann arbeitet daran. Wenn die Korrektur Eurer Lösung für den Tutor zu reiner Dechiffrierung verkommt, dann heißt es im Zweifelsfall, dass Ihr darauf keine Punkte erhaltet. Spätestens in der Klausur wäre das sehr ärgerlich. Und auch wenn Eure Handschrift einem Kunstwerk gleicht und Ihr in der Schule immer ein Sternchen für das Schönschreiben erhalten habt solltet Ihr am Ende trotzdem noch einmal alle Eure Lösungsnotizen auf frischen DIN-A4-Blättern als endgültige Version aufschreiben.
  Wenn Ihr Eure Lösungen am Computer verfassen wollt, dann bietet sich insbesondere in Vorlesungen mit einem großen Anteil an mathematischer Notation das Textsatzsystem LaTeX[1] an. Die meisten Skripte und Übungsblätter werden auf diese Weise verfasst und spätestens für Seminar- oder Abschlussarbeiten sind LaTeX-Kenntnisse von großem Vorteil. Für die gängigen Betriebssysteme gibt es komfortable Entwicklungsumgebungen, die Ihr nutzen könnt, aber nicht müsst. Es gibt auch Menschen, die es lieber einfach halten und nur mit dem Texteditor ihrer Wahl arbeiten.

[1] http://de.wikipedia.org/wiki/LaTeX

- Und zu guter Letzt: Habt Geduld und Durchhaltevermögen! Gerade am Anfang erscheint manches viel schwieriger als es später im Rückblick eigentlich ist. Weitere Tipps zum Thema Übungsaufgaben findet Ihr im folgenden Text von Prof. Dr. Manfred Lehn von der Universität Mainz: `http://www.mathematik.uni-mainz.de/Members/lehn/le/uebungsblatt`. Er spricht dort zwar von „Mathematik“, aber der Text ist genauso auf „Informatik“ anwendbar.

Mit der Zeit werdet Ihr Euch an die universitäre Arbeitsweise gewöhnen und Techniken entwickeln, mit denen Ihr effektiv und effizient arbeiten könnt. Nicht alles funktioniert für jeden. Das hängt von vielen Faktoren ab, beispielsweise von Eurem Lerntyp.[2] Die meisten Menschen sind vorwiegend visuelle Lerntypen, aber es gibt auch Personen, die am besten durch reines Zuhören lernen und wieder andere müssen etwas im wahrsten Sinne des Wortes „begreifen“, um es zu verstehen. Seid Euch über diese Unterschiede bewusst, wählt dementsprechend Eure Lerntechniken, und geht auf Eure Lernpartner ein. Verbindet dabei ruhig auch verschiedene Eingangskanäle, beispielsweise indem Ihr Euch über eine bestimmte Aufgabe unterhaltet, gleichzeitig auf einem Blatt Papier oder an der Tafel das Gesprochene festhaltet und dabei selbst aktiv seid. So bringt Ihr das Gehirn auf Trab und sorgt dafür, dass möglichst viel hängenbleibt.

C.2. Wie geht man an eine Aufgabe in theoretischer Informatik heran?

Wer ein Informatik-Studium an der Goethe-Universität aufnimmt, wird im ersten Semester neben den Programmier- und Mathematik-Vorlesungen auch auf eine Vorlesung aus dem Bereich Theoretische Informatik treffen. Für diejenigen, die im Wintersemester beginnen, ist das die Diskrete Modellierung und im Sommersemester sind es die Datenstrukturen. Auch hier gibt es Übungsaufgaben zu lösen, um insbesondere mathematische Methoden angewandt auf informatische Aufgabenstellungen zu erlernen. Gerade am Anfang stellt die mathematische Genauigkeit für viele eine Hürde da, die es zu meistern gilt. Auch hier ist die Devise: Üben, üben, üben! Doch was ist hier anders als bei den Programmierübungsaufgaben?

Zunächst das Offensichtliche: Es handelt sich um Theorie-Aufgaben. Es muss also nichts programmiert werden in dem Sinne, dass Programmcode über die Tastatur in den Computer eingegeben werden muss und am Ende ein syntaktisch korrektes Programm rauskommt. Stattdessen geht es um theoretische Ideen, die man später für die eigentliche Programmierarbeit verwenden kann. Diese Ideen können mal mehr und mal weniger offensichtlich mit der Programmierung zu tun haben und manchmal geht es auch nur darum, die analytisch-logische Denkweise zu trainieren und für sich selbst Techniken zu entwickeln, wie man am Besten an komplexe Aufgabenstellungen herangeht.

C.2.1. Die Ausgangssituation

Das neue Übungsblatt wurde auf der Vorlesungshomepage veröffentlicht oder in der Vorlesung verteilt.

Welche Ziele wollen wir erreichen?

- Alle Aufgaben richtig lösen.
- Innerhalb der Abgabefrist.
- Dadurch den Vorlesungsstoff so gut verstehen, dass wir ihn auch selbst erklären können.

Was brauchen wir dafür?

[2] `http://www.philognosie.net/index.php/article/articleview/163/`

- Papier, Stift, Mülleimer und Gehirn

Zunächst ist kein Unterschied zu den Praxis-Aufgaben erkennbar. Dort haben wir dieselben Ziele und verwenden ebenfalls Papier und Stift, um Lösungsansätze zu skizzieren, den Mülleimer, da nicht jeder Lösungsansatz zum Erfolg führt und sowieso ist es ein sehr guter Rat, das Gehirn zu benutzen.
Was ist nun der große Unterschied? Wir können nicht mithilfe eines Compilers oder Interpreters am Computer testen, ob unsere Lösung korrekt ist. Davon müssen wir uns selbst überzeugen oder wir müssen warten bis wir vom Tutor unser korrigiertes Übungsblatt zurück erhalten. Wir können daher nicht wie beim Programmieren „einfach mal probieren und gucken, was passiert", da der direkte Feedback-Kanal fehlt. Aber genau dies ist der Sinn der Theorie-Aufgaben. Bei größeren Software-Projekten können wir nicht einfach auf gut Glück programmieren, um dann beim Test festzustellen, dass unser Ansatz schon grundlegend ungeeignet war. Wir müssen vorher nachdenken, analysieren und Lösungen mittels mathematischer Notationen zu Papier bringen.

C.2.2. Konkret: Wie legen wir los?

Wir können grundsätzlich die gleichen Tipps befolgen, die auch schon in dem allgemeinen Abschnitt über die Bearbeitung der Übungsaufgaben dargestellt werden. Da uns der direkte Computer-Feedback-Kanal fehlt, müssen wir von den Folien und Skripten Gebrauch machen. Das ist keine Option, sondern in den allermeisten Fällen dringend erforderlich. In der Theoretischen Informatik haben wir es mit exakten Definitionen und Schreibweisen zu tun, an die wir uns halten müssen. Trotzdem haben wir ebenso viele Freiheiten beim Aufschreiben wie das z. B. beim Programmieren der Fall ist. Es gibt jedoch eine ganze Reihe von typischen Anfängerfehlern, die wir im Folgenden kennenlernen werden.

Fehler:	**Benutzung undefinierter Symbole**
Erklärung:	Im Text tauchen plötzlich Bezeichnungen auf, ohne dass vorher festgelegt wurde, was die Bezeichnungen bedeuten sollen.
Lösung:	Alle Bezeichnungen vor ihrer Benutzung definieren und ggf. erlaubte Wertebereiche angeben.
Beispiel:	„Berechne nun $100/n$." Was ist n? Wahrscheinlich eine Zahl, aber bestimmt nicht 0. Wir müssen sicher stellen, dass die Rechnung gültig ist und ergänzen: „Sei $n \in \mathbb{R} \setminus \{0\}$. Berechne nun $100/n$."
Fehler:	**Verstoß gegen Definitionen**
Erklärung:	Eine Bezeichnung wurde früher im Text als ein gewisser „Datentyp" definiert. Später wird diese Definition ignoriert und ein falscher „Datentyp" verwendet.
Lösung:	Definitionen ggf. im Skript nachschlagen oder die eigenen Definitionen beachten.
Beispiel:	„Seien $A := \{1, 2, 3\}$ und $B := \{2, 4, 6\}$ Mengen und sei $f : A \to B$ eine Funktion mit $f(a) := 2 \cdot a$ für alle $a \in A$. Berechne nun $f(4)$." Jetzt könnten wir denken, dass $f(4) = 2 \cdot 4 = 8$ ist, aber es gilt $4 \notin A$ und deshalb verstoßen wir hier gegen die Definition der Funktion f.

Fehler:	**Gedankensprünge**
Erklärung:	Innerhalb eines Beweises kommt es zu Argumentationslücken. Meistens liegt das daran, dass der Start und das Ziel des Beweises schon laut Aufgabenstellung bekannt sind, wir aber den Weg dazwischen nicht finden und dann versuchen, uns durchzumogeln nach dem Motto: „Das merkt der Tutor sowieso nicht.“ ;-) Doch, das merkt der Tutor und dann muss er dafür Punkte abziehen.
Lösung:	So oft wie möglich „warum?“ fragen. Welche Definitionen können wir verwenden? Welche konkreten logischen Schlussfolgerungen können wir aus den Definitionen ziehen. Warum geht das? Wie geht es dann weiter?
Beispiel:	„Es gelte Aussage X. Zu zeigen: Aus X folgt Aussage Y. Beweis: Da X gilt, ist es ziemlich logisch, dass auch Y gilt. q.e.d.“
Fehler:	**Inkonsistente Bezeichnungen**
Erklärung:	Das Problem tritt häufig auf, wenn Studenten sich bei der Lösung einer Aufgabe aus mehreren Quellen „inspirieren“ lassen. Häufig werden Definitionen und Schreibweisen verwendet, die nicht zu den in der Vorlesung vereinbarten passen.
Lösung:	Wenn wir andere Quellen nutzen, sollten wir diese nicht blind übernehmen. Stattdessen sollten wir versuchen, das dort Geschriebene zu verstehen und auf die Aufgabenstellung des Übungsblattes zu übertragen. Dafür muss die ggf. die Notation angepasst werden, damit die eigene Lösung zur Aufgabenstellung passt. Oft sind die Quellen auch unvollständig und/oder fehlerhaft. Wenn wir die Aufgabe wirklich verstanden haben, können wir diese Fehler ohne Probleme beseitigen und haben dabei sogar noch etwas gelernt.
Beispiel:	In der Aufgabenstellung ist von einer Menge A die Rede und in der „eigenen“ Lösung heißt die gleiche Menge auf einmal X. Einfach nur X durch A zu ersetzen und dann den Rest der Quelle abzuschreiben ist natürlich nicht das Ziel der Aufgabe. Außerdem können wir uns sicher sein, dass andere Studenten das genauso tun werden und der Tutor bei der Korrektur der Übungsblätter stutzig wird, weil er immer wieder die gleichen (falschen) Formulierungen liest und sich dann mit dem Betrugsversuch herumärgern muss. Bei einem solchen Verhalten sind die Bonuspunkte in Gefahr, also lieber auf Nummer sicher gehen und nichts riskieren.

C.2.3. Wann können wir zufrieden sein?

Die obige Liste mit Fehlern ist keineswegs vollständig. Es können jede Menge Kleinigkeiten schiefgehen, an die wir im ersten Augenblick nicht unbedingt denken. Um diesen Fehlern vorzubeugen, ist es ratsam, wenn wir versuchen, unsere Lösung aus der Sicht einer dritten Person zu betrachten. Alternativ lassen wir unsere Lösung von einer hilfsbereiten dritten Person selbständig durchlesen. Eine gut aufgeschriebene Lösung sollte diese Person mit wenig Aufwand lesen und verstehen können. Um zu kontrollieren, ob unsere Lösung gut aufgeschrieben ist, können wir uns an den folgenden Leitfragen orientieren.

- Können wir die aufgeschriebene Lösung auch ohne Kenntnis der Aufgabenstellung nachvollziehen?
- Ist uns wirklich jeder Schritt klar?
- Lassen die Formulierungen Interpretationsspielraum zu? Was könnten wir absichtlich falsch verstehen? Wie können wir präziser formulieren?
- Haben wir die Korrekturanmerkungen des Tutors von den vorherigen Aufgabenblättern beachtet? Tauchen die alten Fehler noch auf?

- Benutzen wir die Sprache des Professors? Ist die Lösung so aufgeschrieben, dass sie auch Teil des Skriptes sein könnte?

Falls wir bei der Beantwortung der Fragen Ergänzungen unserer Lösung vornehmen müssen und so viel frischen Text in unsere bisher aufgeschriebene Lösung quetschen, dass es unübersichtlich wird, sollten wir die neue Version der Lösung noch einmal sauber aufschreiben. Jede schöne Lösung kann auch schön aufgeschrieben werden.

C.3. Von der Idee zum fertigen Programm

C.3.1. Was ist die Ausgangssituation?

Informatik = Programmieren? Nein, genauso wenig wie Kunst = Malen ist, aber das Programmieren ist ein wichtiges Werkzeug im Informatik-Studium. Programmierkenntnisse benötigen wir, um die Ideen und Konzepte aus den Vorlesungen praktisch mithilfe von Computern umsetzen zu können. Und diese Ideen und Konzepte nehmen an der Universität einen großen Teil des Informatik-Studiums ein.

Die ersten Vorlesungen gehen los. Ziel: Wir sollen Programmieren lernen. Wie? Hier, das wöchentliche Übungsblatt. Dort steht:

Aufgabe C.1.
Schreiben Sie eine Funktion `fak` in Python, die als Eingabe eine positive, natürliche Zahl n erhält und als Ergebnis den Wert $n!$ zurückgibt.

Wird jetzt also vorausgesetzt, dass wir schon programmieren können? Wie sonst sollten wir denn diese Aufgabe überhaupt lösen können? Also stimmt es doch, dass man nur Informatik studieren kann, wenn man schon jahrelang programmiert hat und in der Schule Informatik im Leistungskurs hatte!
Keine Panik! Das gehört auch zum Studium: Informatiker kriegen ein großes Problem vorgesetzt und sollen es lösen. Dafür zerhacken sie es in kleine, einfachere Teilprobleme, die sie leichter lösen können. Die Teillösungen werden dann wieder zu einer großen ganzen Lösung für das ursprüngliche Problem zusammengesetzt. Wie sieht das hier aus? Schauen wir uns Aufgabe C.1 noch einmal an: Diese Aufgabe ist unser großes Problem. Wie zerhacken wir es nun in die kleineren, einfacheren Teilprobleme? Das kommt ganz drauf an, wie viel Erfahrung wir mit dem Programmieren und der logisch-strukturierten Denkweise haben, die im Informatikstudium trainiert wird. An dieser Stelle fangen wir mit ganz kleinen Schritten an.

C.3.2. Schritt für Schritt

- schreiben
- Funktion fak
- Python
- Eingabe
- positive, natürliche Zahl n
- Ergebnis
- $n!$
- zurückgeben

All diese Begriffe müssen wir präzise in unseren eigenen Worten erklären und mit konkreten Arbeitsschritten verbinden können.
„Schreiben Sie ...“ bedeutet „Programmieren Sie ...“, wir müssen also Programmcode schreiben. Es wird zwar nicht gesagt, ob wir das auf Papier oder auf dem Computer oder tun sollen, aber

mit Sicherheit kann das Feedback vom Computer hilfreich sein. Er kann uns auf bestimmte Programmierfehler aufmerksam machen. Weiter geht es mit „eine Funktion `fak`“. Funktionen kennen wir aus der Mathematik. Dort haben wir bestimmt schon so etwas gesehen: $f(x) = x^2$. Diese Funktion mit dem Namen f nimmt einen Wert x entgegen und rechnet das Quadrat x^2 aus. In Programmiersprachen verhält es sich ähnlich. In dieser Aufgabe heißt die Funktion allerdings nicht f, sondern `fak`. Diese Funktion sollen wir in der Programmiersprache Python schreiben. Wie sehen Funktionen in Python aus?

```
1 def funktionsname(eingabe):
2     ...
3     return ergebnis
```

Das ist das Grundgerüst unseres Programmes. Wir müssen es nur auf das aktuelle Problem anpassen. In der Aufgabe wird von einer Eingabe namens n gesprochen. Das ist das, was im obigen Code in den Klammern nach dem Funktionsnamen steht. In dem mathematischen Beispiel $f(x) = x^2$ ist die Eingabe das x bei $f(x)$. Unsere Funktion `fak` soll nun ein Ergebnis ausrechnen, nämlich $n!$ und es zurückgeben. Dieses Zurückgeben wird mit dem `return`-Befehl erreicht.

```
1 def fak(n):
2     ...
3     hier muss n! ausgerechnet werden
4     ...
5     return ergebnis
```

So sind wir schon ein ganzes Stück näher an die Lösung gekommen. Immerhin kennen wir jetzt den äußeren Rahmen des Programms. Jetzt geht es weiter ins Detail: n soll irgendeine positive, natürliche Zahl sein, d. h. n ist 1 oder 2 oder 3 oder 4 usw. Das können wir mit Python allerdings nicht ohne Weiteres erzwingen, da Python eine dynamisch typisierte Sprache ist. Diese Information ist nur für uns gedacht. Die Funktion `fak` soll $n!$ ausrechnen. Was bedeutet das Ausrufezeichen?

Definition C.1 (Fakultät)**.**
Für jede positive, natürliche Zahl n ist

$$n! := n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1.$$

Beispiel C.2.
Verdeutlichen wir uns diese Definition an einigen Beispielen:

- $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$
- $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$
- $100! = 100 \cdot 99 \cdot 98 \cdot \ldots \cdot 3 \cdot 2 \cdot 1$
- $1! = 1$
- $0!$ ist laut der obigen Definition nicht möglich.

Die mathematische Definition müssen wir in Python-Code ausdrücken. Für die Eingabe n müssen wir alle Zahlen von 1 bis n aufmultiplizieren. Auch hier können uns Schreibweisen aus der Mathematik helfen:

$$n! = \prod_{i=1}^{n} i.$$

Was passiert hier? Wir benutzen eine Variable i, die schrittweise alle Zahlen $1, 2, 3, \ldots, n$ durchläuft. Dies entspricht in der Programmierung gerade einer Schleife mit einem Zähler i. In Python können wir dieses Produkt mit dem folgenden Code realisieren:

```
1 prod = 1
2 for i in range(1,n+1):
3     prod = prod * i
```

Diesen Code müssen wir nun in den Code der Funktion `fak(n)` einbauen und schon sind wir fast fertig:

```
1 def fak(n):
2     prod = 1
3     for i in range(1,n+1):
4         prod = prod * i
5     return ergebnis
```

Tut der Code was er soll? Für jede positive, natürliche Zahl n wird die Fakultät $n!$ berechnet und in der Variable `prod` gespeichert. Anschließend geben wir die Variable `ergebnis` zurück. Problem: Welcher Wert steht in `ergebnis`? Das wissen wir nicht. Eigentlich sollte dort $n!$ drin stehen. Das müssen wir noch beheben, z. B. indem wir nicht die Variable `ergebnis` mittels `return` zurückgeben, sondern die Variable `prod`.

```
1 def fak(n):
2     prod = 1
3     for i in range(1,n+1):
4         prod = prod * i
5     return prod
```

Nun sind wir fertig und haben eine Lösung für die Aufgabe gefunden. Das ist aber nur eine Lösung, nicht zwingenderweise die einzige Lösung. Statt uns vom Produkt $n! = \prod_{i=1}^{n} i$ inspirieren zu lassen, hätten wir auch eine andere Beobachtung in Python-Code umsetzen können:

$$1! = 1 \quad \text{und}$$
$$n! = n \cdot (n-1)!, \qquad \text{falls } n \geq 2$$

Die Fakultät von n können wir rekursiv berechnen, indem wir n mit der Fakultät von $n-1$ multiplizieren. Wie das geht, überlassen wir dem Leser als Hausaufgabe.
Viel Spaß beim Programmieren!

C.4. Wie geht man ein größeres Programmierprojekt an?

Im Laufe eines Informatik-Studiums, im Berufsleben oder in Freizeit-Projekten werden wir damit konfrontiert sein, komplexe Anforderungen in Software umzuwandeln. Wir sind die Compiler, die die echte Welt in Quellcode übersetzen müssen. Dafür gibt es kein automatisiertes Verfahren und es ist eher unwahrscheinlich, dass sich dies in absehbarer Zeit ändert. Allerdings können wir uns in der Vorgehensweise an den im Kapitel „Programmieren und Programmiersprachen“ vorgestellten Schritten orientieren:

1. Zerlegung des Problems in Teilprobleme
2. Präzisierung der Teilprobleme
3. Entwickeln einer Lösungsidee für die Teilprobleme und das Gesamtproblem
4. Entwurf eines groben Programmgerüsts
5. Schrittweise Verfeinerung
6. Umsetzung in eine Programmiersprache
7. Testen und Korrigieren von Fehlern

Diese sieben Schritte stellen lediglich einen Vorschlag dar und sind keinesfalls die einzig richtige Methode, ein größeres Programmierprojekt zu bewältigen. Softwareentwickungsprozesse sind komplex und es wäre viel zu kurz gegriffen zu glauben, dass man sie immer in sieben aufeinander folgende Schritte unterteilen könne. Häufig ändern sich während des Projekts die Anforderungen oder neue Aspekte kommen hinzu, wodurch wir einzelne Phasen der Entwicklung durchaus mehrfach wiederholen müssen. Das ist völlig normal und kein Zeichen für eine schlechte Methodik. Außerdem lassen sich die Phasen nicht immer klar abgrenzen und haben fließende Übergänge.
Wir werden nun die sieben oben erwähnten Phasen anhand eines Beispiel-Projekts näher kennenlernen.

Projekt	**Tic Tac Toe** Das Spiel *Tic Tac Toe* soll unter Benutzung der Programmiersprache Python implementiert werden.
Spielregeln:	Zwei Spieler setzen auf einem 3×3-Spielfeld abwechselnd ein Kreuz (X) bzw. einen Kreis (O) in ein leeres Feld. Es gewinnt derjenige, der zuerst drei seiner Symbole in einer horizontalen, vertikalen oder diagonalen Reihe hat.
Anforderungen:	Das Spiel soll als Konsolenanwendung realisiert werden. Eine grafische Benutzeroberfläche ist nicht notwendig. Beim Start des Spiels soll der Benutzer mit einer kurzen Willkommensnachricht begrüßt werden, in der erklärt wird, mit welchen Eingaben er eine neue Partie starten oder das Spiel wieder beenden kann. Zusätzlich soll die Möglichkeit bestehen, die Spielregeln abzurufen. Bei Fehleingaben innerhalb des Willkommen-Menüs soll der Benutzer darauf hingewiesen werden und weiterhin im Willkommen-Menü verbleiben. Beim Start einer neuen Partie soll zunächst nach den teilnehmenden Spielern und deren Namen gefragt werden. Es kann sich wahlweise um menschliche oder Computer-Spieler handeln. Folgende Kombinationen sind möglich: Mensch-Mensch, Mensch-Computer, Computer-Mensch, Computer-Computer. Letzteres ist die *War Games*[3]-Variante *„A strange game: The only winning move is not to play“*. Während einer Partie müssen die Spielregeln eingehalten werden und nach jedem Zug muss das Spielfeld auf die Siegbedingung untersucht werden. Computer-Spieler sollten sich möglichst intelligent verhalten und den Eindruck erwecken, sie müssten auch „nachdenken“. Sollte ein Spieler gewonnen haben, so ist die Partie beendet und eine Meldung erscheinen, welcher Spieler gesiegt hat. Im Anschluss soll der Benutzer entweder eine weitere Partie mit den gleichen Spielern starten, zurück zum Willkommen-Menü gehen oder das Spiel beenden können. Wenn das Spiel beendet wird, soll zunächst für einige Sekunden ein Abspann auf dem Bildschirm erscheinen.

Der jetzige Stand der Dinge ist, dass uns ein Blocktext mit den Projektanforderungen vorliegt. Er ist nicht vollständig und lässt Interpretationsspielräume offen. Beispielsweise wird wohl jeder seine eigene Vorstellung davon haben, wie der Willkommen- oder der Abspann-Bildschirm aussehen könnte. Es gibt daher nicht *die* richtige oder *die* falsche Implementierung, sondern nur verschiedene Realisierungen der gleichen Anforderung. Widmen wir uns nun dem Sieben-Phasen-Modell.

C.4.1. Zerlegung des Problems in Teilprobleme

Der Projektbeschreibung nach können wir mehrere Teilprobleme ausmachen. Wir haben das eigentliche Spiel *Tic Tac Toe* mit seinen Spielregeln. Dieses Spiel soll im Programm in eine textbasierte

[3] `http://www.imdb.com/title/tt0086567/`

Benutzeroberfläche (engl. *command line interface (CLI)*) eingebettet werden. Das CLI reagiert auf die Eingaben des Benutzers und steuert den Programmfluss. Außerdem sollen Computer-Spieler angeboten werden, die über irgendeine Form künstlicher Intelligenz verfügen. Wir können daher die drei Teilprobleme

- die Benutzeroberfläche
- das eigentliche Spiel Tic Tac Toe
- die künstliche Intelligenz der Computer-Spieler

identifizieren. Im nächsten Schritt ermitteln wir genauer, welche Anforderungen an die einzelnen Teilprobleme gestellt wurden. Der Vorteil dieses Ansatzes ist, dass wir die Teilprobleme jetzt sogar unter mehreren Entwicklern aufteilen können, die parallel weiter arbeiten.

C.4.2. Präzisierung der Teilprobleme

Der Einfachheit halber filtern wir an dieser Stelle einfach die relevanten Sätze aus der obigen Problembeschreibung heraus und erstellen daraus eine Beschreibung der einzelnen Teilprobleme. Schlüsselworte sind *kursiv* hervorgehoben.

Die Benutzeroberfläche

- Das Spiel soll als *Konsolenanwendung* realisiert werden.
- Beim Start des Spiels soll der Benutzer mit einer kurzen *Willkommensnachricht* begrüßt werden, in der erklärt wird, mit welchen Eingaben er eine neue Partie starten oder das Spiel wieder beenden kann. Zusätzlich soll die Möglichkeit bestehen, die Spielregeln abzurufen.
- Bei Fehleingaben innerhalb des *Willkommen-Menüs* soll der Benutzer darauf hingewiesen werden und weiterhin im Willkommen-Menü verbleiben.
- Beim Start einer neuen *Partie* soll zunächst nach den teilnehmenden *Spielern* und deren *Namen* gefragt werden. Es kann sich wahlweise um menschliche oder Computer-Spieler handeln. Folgende Kombinationen sind möglich: Mensch-Mensch, Mensch-Computer, Computer-Mensch, Computer-Computer.
- Sollte ein Spieler *gewonnen* haben, so ist die Partie beendet und eine Meldung erscheinen, welcher Spieler gesiegt hat. Im Anschluss soll der Benutzer entweder eine weitere Partie mit den gleichen Spielern starten, zurück zum Willkommen-Menü gehen oder das Spiel beenden können.
- Wenn das Spiel beendet wird, soll zunächst für einige Sekunden ein *Abspann* auf dem Bildschirm erscheinen.

Das eigentliche Spiel Tic Tac Toe

- Während einer Partie müssen die *Spielregeln* eingehalten werden: Zwei Spieler setzen auf einem 3×3-*Spielfeld* abwechselnd ein *Kreuz (X)* bzw. einen *Kreis (O)* in ein leeres Feld.
- Nach jedem Zug muss das Spielfeld auf die *Siegbedingung* untersucht werden. Es gewinnt derjenige, der zuerst drei seiner Symbole in einer horizontalen, vertikalen oder diagonalen Reihe hat. Es gibt drei horizontale, drei vertikale und zwei diagonale Reihen.

Die künstliche Intelligenz der Computer-Spieler

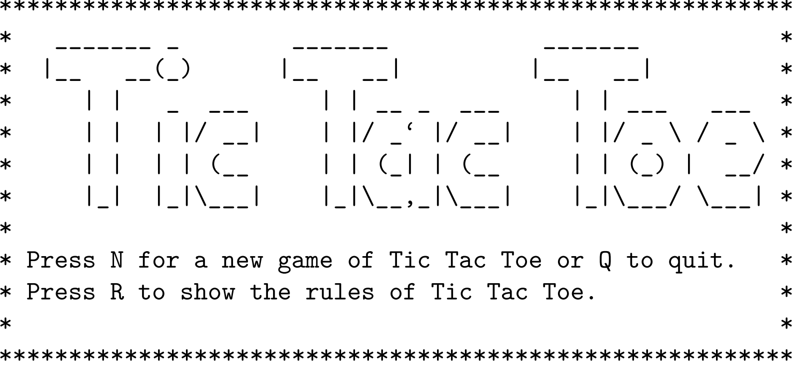
- Computer-Spieler sollten sich *möglichst intelligent* verhalten ...
- und den Eindruck erwecken, sie müssten auch *„nachdenken“*.

C.4.3. Entwickeln einer Lösungsidee für die Teilprobleme und das Gesamtproblem

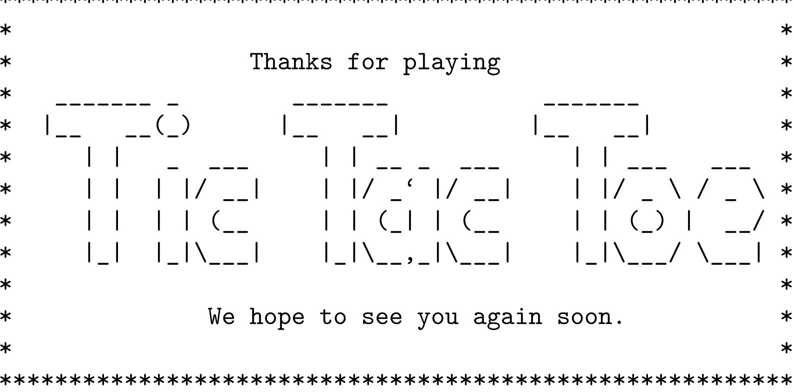
In diesem Schritt überlegen wir uns langsam, wie wir die Programmabläufe modellieren können und welche Programmierkonstrukte wir dafür verwenden können. Wir überlegen uns, *was* wir später programmieren möchten, aber noch nicht zwingenderweise *wie*.

Die Benutzeroberfläche

Die Benutzeroberfläche hat die Aufgabe, die Eingaben des Benutzers entgegenzunehmen und vom Programm berechnete Ergebnisse auszugeben. Sie führt den Benutzer durch unser Tic-Tac-Toe-Programm.
Das Willkommen-Menü könnte beispielsweise so aussehen:



Auf eine ähnliche Weise könnten wir den Abspann darstellen:



Durch die Darstellung des Willkommen-Menüs haben wir uns schon indirekt überlegt, welche Funktionstasten im Programm eine Rolle spielen sollen. `N`, `Q` und `R` haben in diesem Menü eine besondere Bedeutung und lösen bestimmte Aktionen aus. Jeder andere Tastendruck an dieser Stelle stellt eine Fehleingabe dar, die wir mit `Wrong key. Please press N, Q or R.` quittieren. Auf ähnliche Weise können wir uns die anderen Bildschirmausgaben anderen Situationen überlegen.

Das eigentliche Spiel Tic Tac Toe

Wir benötigen ein Spielfeld mit neun Feldern, die entweder leer oder von Spieler 1 bzw. Spieler 2 belegt sind. Das Ende der zweiten Runde könnte auf dem Bildschirm etwa so aussehen

```
_X_
OO_
_X_
```

und in Python durch die folgende Liste realisiert werden, wobei die ersten drei Einträge für die erste Zeile stehen, die nächsten drei für die zweite und die letzten drei für die dritte Zeile des Spielfeldes:
`field = ["_", "X", "_", "O", "O", "_", "_", "X", "_"]`.
Solange das Spiel noch nicht vorbei ist, sind die Spieler abwechselnd am Zug. Hierfür müssen wir uns im Programm merken, wer gerade dran ist. Der aktuelle Spieler kann dann einen Zug machen, d. h. ein Feld auswählen, wo er sein Symbol hinsetzen möchte. Das Programm muss gewährleisten, dass dieses Feld noch nicht besetzt ist. Wählt der Spieler ein besetztes Feld, wird der Zug abgelehnt und er wird erneut um einen gültigen Zug gebeten. Hierbei ist zu unterscheiden, ob ein menschlicher oder ein Computer-Spieler zieht. Der menschliche Spieler gibt seinen Zug über eine Tastatur-Eingabe bekannt, der Computer-Spieler kann sich im Rahmen seiner künstlichen Intelligenz selbst entscheiden. Außerdem benötigen wir im Programm eine Funktion, die nach jedem Zug das Spielfeld, d. h. die Liste `field` darauf überprüft, ob drei X bzw. drei O in einer Reihe stehen.

Die künstliche Intelligenz der Computer-Spieler

Die Anforderungen sind sehr vage formuliert. Wir wissen nur, dass der Computer-Spieler „möglichst intelligent“ sein soll und „nachdenken“ muss. Um uns zu überlegen, wie der Computer-Spieler agiert, betrachten wir das menschliche Spielerverhalten. Wenn wir am Zug sind, schauen wir auf das Spielfeld und sehen die möglichen freien Felder. Für eines dieser Felder müssen wir uns entscheiden. Die einfachste, aber nicht unbedingt klügste Variante ist, wenn wir unser Symbol einfach auf das erste freie Feld setzen, welches wir sehen. In einer ersten Implementierung könnte sich der Computer-Spieler genauso verhalten. Das ist zwar nicht „möglichst intelligent“, sondern „ziemlich doof“, aber es erfüllt seinen Zweck: Der Computer kann spielen. Später können wir das immernoch verbessern. Um das Nachdenken zu simulieren, können wir die Computer-Spieler-Funktion künstlich verlangsamen, etwa indem wir ein Pause-Intervall von ein paar Sekunden einfügen und auf dem Bildschirm eine Meldung wie `<Player (CPU)> is thinking` ausgeben.

C.4.4. Entwurf eines groben Programmgerüsts

Im vorherigen Schritt haben wir versucht, die drei Teilprobleme um detailliertere Beschreibungen zu ergänzen. Nun müssen wir uns langsam der konkreten Implementierung in der Programmiersprache Python nähern. Bevor wir jedoch den Texteditor auspacken und anfangen, den Code zusammenzuhacken, malen wir uns den Programmablauf auf einem Blatt Papier auf. Welche Verarbeitungsschritte gibt es und wie hängen sie zusammen? Dafür können wir uns an *Programmablaufplänen (PAP)*[4] orientieren. Unseren vorherigen Überlegungen zufolge könnten wir den in Abb. C.1 dargestellten Programmablauf erhalten.

Natürlich müssen wir die einzelnen Aktionen noch genauer beschreiben. Beispielsweise können wir die Aktion „Tic Tac Toe spielen“ ebenfalls in ähnlicher Weise darstellen.

C.4.5. Schrittweise Verfeinerung

Nun kommt es zum Feinschliff unserer Ideen. Wie nennen wir die Funktionen? Welche Parameter gibt es? Welche Funktionen gehören thematisch zusammen? Welche Variablen benötigen wir? Wie

[4] http://de.wikipedia.org/wiki/Programmablaufplan

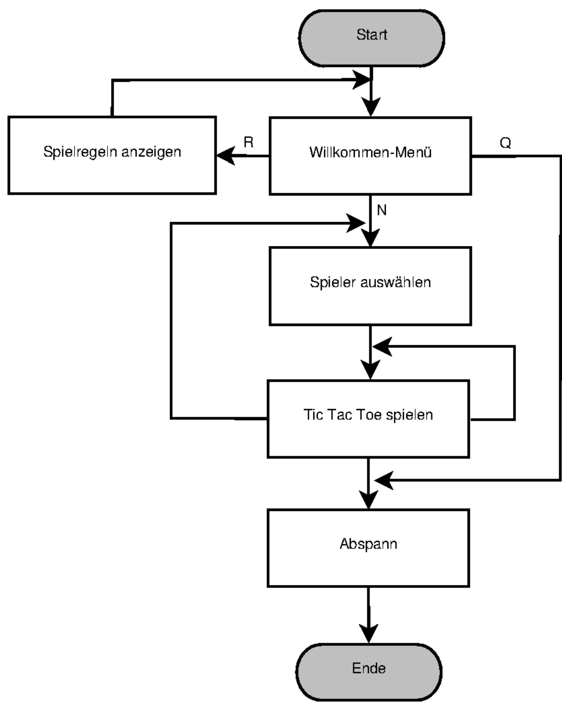


Abbildung C.1.: Programmablauf von Tic Tac Toe

prüfen wir, ob das Spiel zu Ende ist, wenn wir das Spielfeld als Liste in Python gegeben haben? Hier bewegen wir uns gedanklich schon auf der Computer-Ebene. Wir überlegen uns, wie der Algorithmus aussieht und was das für den Programmcode bedeutet.

C.4.6. Umsetzung in eine Programmiersprache

Jetzt können wir endlich anfangen zu programmieren. Wir packen unsere Entwicklungsumgebung aus und legen los. Dabei halten wir uns selbstverständlich an die Programmierkonventionen und kommentieren unseren Code, damit wir ihn auch noch eine Weile später noch verstehen können. Wenn wir in den vorherigen Schritten gut gearbeitet haben und schon über ein wenig Programmiererfahrung verfügen, müssen wir in dieser Phase nicht mehr mit dem Problem Tic Tac Toe kämpfen. Unser jetziger Gegner heißt Python. Wahrscheinlich werden uns häufig Fehlermeldungen begegnen, während wir programmieren. Davon sollten wir uns nicht einschüchtern lassen. Es ist sinnvoll, die Fehlermeldungen aufmerksam zu lesen. Was steht da überhaupt? Wenn wir damit nicht weiter kommen, weiß vielleicht ein anderer Programmierer Rat, der die Fehlermeldung schon kennt. Wenn kein potenzieller Antwortgeber in der Nähe sitzt, dann ist eine Internet-Recherche ratsam. Irgendwo auf der Welt wird jemand sehr wahrscheinlich über das gleiche Problem gestolpert sein, der daraufhin in irgendeinem Forum oder Q&A-Portal seine Frage gestellt hat, die dort hoffentlich beantwortet wurde und von uns gefunden werden kann.

C.4.7. Testen und Korrigieren von Fehlern

In der Regel läuft diese Phase teilweise schon parallel zur tatsächlichen Implementierung in Python. Gerade am Anfang testen wir ständig aus, ob unser Code das tut, was wir beabsichtigen. Das ist einerseits gut, da wir Fehler früh erkennen können und ein Gefühl für die Programmiersprache bekommen, kann aber auch ein Hindernis sein, wenn wir uns frühzeitig in kleinen Fehlern verrennen, die den Gesamtfortschritt behindern. Es ist legitim, einen Fehler zunächst nicht zu

korrigieren, wenn er kein essentielles Problem für den Programmablauf darstellt. Um solche Fehler niedriger Priorität können wir uns später kümmern, wenn die groben Trümmer zur Seite geräumt sind.
Wichtig in dieser Phase ist, dass wir das implementierte Programm noch einmal mit den anfangs gestellten Anforderungen vergleichen. Tut es genau das, was in der Anforderung spezifiziert wurde? Dies herauszufinden ist die Aufgabe des abschließenden Software-Tests. Leider hat die Sache einen Haken. Wir können durch die Tests nicht beweisen, dass das Programm fehlerfrei ist. Wir können durch geeignete Testfälle nur die Anwesenheit von Fehlern beweisen.