

# **Programmieren für Naturwissenschaften**

**Vorlesungsskript (4.1)**

**FS 2025**

**PD Dr. Kaspar Riesen**

**[kaspar.riesen@unibe.ch](mailto:kaspar.riesen@unibe.ch)**

# Inhaltsverzeichnis

<b>I Python Basics</b>	<b>1</b>
<b>1 Einführung und Motivation</b>	<b>2</b>
1.1 Weshalb Python? . . . . .	2
1.2 Ein Erstes Python Programm . . . . .	3
1.2.1 Die Eingebaute Funktion <code>print</code> . . . . .	5
1.2.2 Kommentare . . . . .	8
1.3 Programmentwicklung . . . . .	10
1.3.1 Kompilieren und Interpretieren . . . . .	10
1.3.2 Python Interaktiv Verwenden . . . . .	12
1.3.3 Python im Skript Modus Verwenden . . . . .	17
1.3.4 Programmierfehler . . . . .	20
<b>2 Variablen und Listen</b>	<b>23</b>
2.1 Variablen . . . . .	23
2.1.1 Eingaben Entgegennehmen . . . . .	34
2.2 Listen – die Datenstruktur <code>list</code> . . . . .	36
2.2.1 Listen-Abstraktionen . . . . .	45
<b>3 Schleifen und Bedingungen</b>	<b>47</b>
3.1 Die <code>if</code> -Anweisung . . . . .	48
3.1.1 Boolesche Ausdrücke . . . . .	50
3.1.2 Die <code>if-else</code> Anweisung . . . . .	54
3.1.3 Verschachtelte <code>if</code> -Anweisungen . . . . .	55
3.1.4 Die <code>if-elif</code> -Anweisung . . . . .	57
3.2 Die <code>while</code> -Anweisung . . . . .	59
3.2.1 Endlosschleifen . . . . .	63
3.2.2 Verschachtelte Schleifen . . . . .	63

3.3	Die <code>for</code> -Anweisung . . . . .	65
<b>4</b>	<b>Standardmodule Verwenden</b>	<b>70</b>
4.1	Das Modul <code>random</code> . . . . .	71
4.2	Das Modul <code>math</code> . . . . .	75
4.3	Das Modul <code>statistics</code> . . . . .	78
4.4	Dateien Lesen und Schreiben . . . . .	80
4.4.1	Vom Umgang mit Zeichenketten . . . . .	81
4.4.2	Dateien Lesen . . . . .	86
4.4.3	Dateien Schreiben . . . . .	87
4.4.4	Das Modul <code>csv</code> . . . . .	90
<b>5</b>	<b>Eigene Funktionen Programmieren</b>	<b>92</b>
5.1	Funktionen Definieren . . . . .	93
5.2	Funktionen Aufrufen . . . . .	94
5.3	Parameter an Funktionen Übergeben . . . . .	97
5.4	Verändern von Parametern in Funktionen . . . . .	101
5.5	Die <code>return</code> Anweisung . . . . .	104
5.6	Teilen-und-Beherrschen . . . . .	107
<b>6</b>	<b>Weitere Datenstrukturen</b>	<b>112</b>
6.1	Zweidimensionale Listen . . . . .	112
6.1.1	Kopieren von Listen . . . . .	116
6.2	Wörterbücher – Die Datenstruktur <code>dict</code> . . . . .	118
6.3	Tupel – Die Datenstruktur <code>tuple</code> . . . . .	124
6.4	Mengen – Die Datenstruktur <code>set</code> . . . . .	127
<b>II</b>	<b>Datenprobleme mit Python Lösen</b>	<b>130</b>
<b>7</b>	<b>Daten Sortieren, Filtern und Zusammenfassen</b>	<b>131</b>
7.1	Datenstrukturen: <code>Series</code> und <code>DataFrame</code> . . . . .	132
7.1.1	Daten mit <code>pandas</code> Lesen und Schreiben . . . . .	135
7.2	Daten Sortieren . . . . .	137
7.3	Daten Filtern . . . . .	138
7.4	Daten Zusammenfassen . . . . .	141

<b>8 Daten Durchsuchen</b>	<b>145</b>
8.1 Komplexität von Algorithmen . . . . .	146
8.2 Zugehörigkeitsoperatoren . . . . .	148
8.3 Lineare Suche . . . . .	151
8.4 Binäre Suche . . . . .	152
<b>9 Daten Visualisieren</b>	<b>156</b>
9.1 Liniendiagramme . . . . .	156
9.1.1 Linien und Farben . . . . .	158
9.1.2 Mehrere Datenreihen . . . . .	161
9.1.3 Text . . . . .	163
9.2 Weitere Diagramme . . . . .	165
<b>10 Daten Vergleichen</b>	<b>174</b>
10.1 Minkowski Distanzen . . . . .	175
10.2 Kosinus Ähnlichkeit . . . . .	178
10.3 Jaccard Ähnlichkeit . . . . .	180
10.4 Levenshtein Distanz . . . . .	182
<b>11 Daten Gruppieren</b>	<b>189</b>
11.1 Clustering: Definition und Ziele . . . . .	189
11.2 Hierarchische Clusterings . . . . .	191
11.3 Mit Python ein Clustering Berechnen . . . . .	198
<b>12 Daten Klassifizieren</b>	<b>201</b>
12.1 Lernen eines Klassifikationsmodells . . . . .	201
12.2 Der $k$ -NN Klassifikator . . . . .	206
12.3 Mit Python Daten Klassifizieren . . . . .	210

# **Teil I**

# **Python Basics**

# Kapitel 1

## Einführung und Motivation

### 1.1 Weshalb Python?

Programmieren hat viel mit Lösen von Problemen zu tun. Das allgemeine Ziel der Programmierung ist es, zu gegebenen Problemen ein oder mehrere Programme zu entwickeln, die auf Computern ausführbar sind und dabei die Probleme korrekt, vollständig und möglichst effizient lösen. Die Komplexität der Probleme, die mit Programmen gelöst werden sollen, kann dabei stark variieren: Von sehr einfachen Problemen wie z.B. der Addition zweier Zahlen, bis hin zu sehr komplexen Problemen, wie z.B. der Steuerung von Passagierflugzeugen.

Möchte man ein Programm schreiben, das ein Computer ausführen kann, so muss dies in einer Sprache geschehen, welche eine Maschine verstehen kann – wir benötigen also eine *Programmiersprache*. Programmiersprachen definieren bestimmte Wörter und Symbole zusammen mit einer Menge an Regeln, die genau bestimmen, wie eine Programmiererin<sup>1</sup> die Wörter und Symbole der Sprache zu gültigen *Pro-*

---

<sup>1</sup>Um den Lesefluss nicht zu beeinträchtigen, nutzen wir in diesem Skript meist nur die weibliche Form: *die Programmiererin*. Hiermit sind immer alle Programmier:innen mitgemeint. Umgekehrt nutzen wir meist nur die männliche Form für *der Benutzer* – und auch hier steht dieser Begriff stellvertretend für alle Benutzer:innen.

*grammieranweisungen* (engl. *Programming Statements*) kombinieren kann. Diese Anweisungen werden dann während der Ausführung des Programmes durch den Computer in einer bestimmten Reihenfolge ausgeführt.

Es existieren Dutzende von Programmiersprachen, die z.T. für unterschiedlichste Zwecke definiert worden sind. Diese unterschiedlichen Sprachen besitzen jeweils Vor- und Nachteile, die von vielen Faktoren abhängig sind. In diesem Skript werden wir mit Hilfe der Programmiersprache *Python* das Programmieren erlernen. Obschon es sehr schwierig bzw. unmöglich ist, zu sagen, welche Sprache die *beste* Programmiersprache ist, darf man sagen, dass die Programmiersprache *Python* einige gewichtige Vorteile aufweist:

- Python ist relativ intuitiv und eher leicht zu erlernen
- Python kann für unterschiedlichste Zwecke verwendet werden
- Python ist plattformunabhängig
- Python ist gemäss TIOBE einer der populärsten Programmiersprachen der letzten Jahre<sup>2</sup>

## 1.2 Ein Erstes Python Programm

Programmiersprachen können in verschiedene *Programmierparadigmen* eingeteilt werden. Ein Programmierparadigma entspricht einem bestimmten Programmierstil, der Konzepte für verschiedene Programmieraufgaben festlegt. Python ist eine sogenannte *Multiparadigmen-sprache*. Das bedeutet, Python zwingt Programmiererinnen nicht zu

---

<sup>2</sup><https://www.tiobe.com/tiobe-index/>

einem einzigen Programmierstil, sondern erlaubt es, die für die jeweilige Aufgabe am besten geeigneten Konzepte zu verwenden.

Wir werden Python zunächst zur *imperativen Programmierung* verwenden. Bei der **imperativen Programmierung** wird festgelegt, was in welcher Reihenfolge zu tun ist: *"First do this, next do that and then do this . . . "*

Nachfolgend ist der *Quellcode* (engl. *Sourcecode*) eines ersten, sehr einfachen Python Programmes abgebildet.

```
"""
quote.py
"""

# gibt ein Zitat von Steve Jobs auf der Konsole des Computers aus
print("Steve Jobs:")
print("Es ist besser, ein Pirat zu sein, als der Marine beizutreten.")
```

Wird dieses Programm ausgeführt, werden die folgenden zwei Zeilen auf dem Bildschirm<sup>3</sup> des Computers ausgegeben:

**Steve Jobs:**

**Es ist besser, ein Pirat zu sein, als der Marine beizutreten.**

Obwohl das obige Programm sehr einfach und noch nicht sehr nützlich ist, illustriert dieses Programm einige wichtige Dinge:

- Programmieranweisungen werden in Python in sogenannten *Modulen* definiert. **Module** sind Dateien mit der Dateierweiterung **.py**. Der Name des Moduls ist der Name der Datei. In unserem

---

<sup>3</sup>Etwas genauer: Die Ausgabe der zwei Zeilen erfolgt auf der sogenannten *Konsole* des Computers.

Beispiel heisst das Modul `quote` und ist in einer Datei `quote.py` gespeichert.

- Ein Python Programm kann i.a. aus mehreren Modulen bestehen. Unser erstes Python Programm besteht aber nur aus einem Modul.
- Module enthalten i.d.R. Programmieranweisungen, die das Programm definieren. Ein Modul kann neben Programmieranweisungen andere Dinge enthalten, z.B. Funktionen oder Variablen oder Kommentare. Unser Modul `quote` enthält zwei Programmieranweisungen und zwei Kommentare.
- Kommentare beeinflussen das Programm nicht – diese sollen lediglich das Lesen und Verstehen des Quellcodes für Menschen erleichtern.
- Wird ein Python Modul ausgeführt, so wird – im Prinzip – jede Programmieranweisung im Modul von oben nach unten ausgeführt. Wird das Ende der Datei erreicht, endet das Programm (man sagt, das Programm *terminiert*).
- Wenn das obige Programm ausgeführt wird, dann geschieht folgendes: Die Funktion `print` wird zweimal nacheinander mit unterschiedlichen Parametern aufgerufen. Danach wird das Ende der Datei erreicht und das terminiert unser Python Programm.

### 1.2.1 Die Eingebaute Funktion `print`

Eine Funktion ist eine Gruppierung von Programmieranweisungen, die unter einem bestimmten Namen zusammengefasst werden. Wir rufen eine Funktion auf, wenn wir wollen, dass die Programmieranweisungen

dieser Funktion ausgeführt werden. Wird eine Funktion aufgerufen, so wird der sogenannte *Kontrollfluss* des Programms an diese Funktion abgegeben. Jetzt werden die Programmieranweisungen dieser Funktion ausgeführt. Sind alle Anweisungen, die in der Funktion zusammenfasst sind, ausgeführt, kehrt die Kontrolle zum Punkt des Programmes zurück, wo die Funktion aufgerufen wurde.

Der Quellcode, der ausgeführt wird, wenn die Funktion `print` aufgerufen wird, ist in unserem Programm nicht ersichtlich. Die Funktion `print` ist Teil der sogenannten *Python Standard Library*. Diese Standardbibliothek enthält in Python geschriebener Quellcode, den wir nutzen können. Das heisst, jemand anderes hat die Funktionalität für Ausgaben von Zeichen auf dem Bildschirm in Python programmiert und die nötigen Anweisungen unter dem Namen `print` zusammengefasst. Statt selber diese Funktionalität zu programmieren, rufen wir einfach bestehenden Quellcode aus der Bibliothek auf.

Die Funktion `print` ist ein Beispiel einer sogenannt *eingebauten Funktion* (engl. *Built-in Functions*). Eingebaute Funktionen sind *immer* verfügbar und können direkt aufgerufen werden<sup>4</sup>.

Die Funktion `print` ist so programmiert, dass diese *Parameter* entgegen nehmen kann, nämlich die Daten, die ausgegeben werden sollen. Es gibt Funktionen, die keine Parameter benötigen – die Klammern () sind trotzdem bei jedem Funktionsaufruf nötig. Z.B. kann man die Funktion `print` tatsächlich auch ohne Parameter aufrufen:

---

<sup>4</sup>Hier finden Sie eine aktuelle Liste aller eingebauten Funktionen:  
<https://docs.python.org/3/library/functions.html>

```
print()
```

Diese Anweisung gibt eine leere Zeile ohne Inhalt aus.

Der Funktion `print` kann man auch mehr als einen Parameter mitgeben (durch Kommas getrennt). Die einzelnen Parameter werden dann jeweils mit einem Leerschlag getrennt ausgegeben. Die Anweisung

```
print("Resultat:", 17)
```

erzeugt demnach die Ausgabe `Resultat: 17`

Die Funktion `print` von Python fügt standardmäßig an das Ende jeder Ausgabe einen Zeilenumbruch. Die Ausgabe von

```
print("Achtung!")
print("Fertig!")
print("Los!")
```

ist somit:

Achtung!

Fertig!

Los!

Man kann der Funktion `print` aber ein spezielles Argument `end="d"` mitgeben. Diese Argument bewirkt, dass bei einer Ausgabe das Zeichen `"d"` anstelle des Zeilenumbruchzeichens am Ende der Zeile platziert wird<sup>5</sup>.

<sup>5</sup>Damit der Interpreter unterscheiden kann zwischen einer Zeichenkette, die ausgegeben werden

Die Ausgabe von

```
print("Achtung!", end=" ")
print("Fertig!", end="...")
print("Los!")
```

ist somit:

Achtung! Fertig!...Los!

Wie oben gesehen, ist das Trennzeichen bei einer Ausgabe von mehreren Parametern das Leerzeichen " ". Auch dieses Standardzeichen kann durch ein eigenes Trennzeichen "d" ersetzt werden – mit dem Argument `sep="d"`. Die Ausgabe von

```
print("Achtung!", "Fertig!", "Los!", sep="-->")
```

ist also bspw.

Achtung!-->Fertig!-->Los!

### 1.2.2 Kommentare

Typischerweise enthalten Python Programme an verschiedenen Stellen Kommentare, welche den Sinn und Zweck des Quellcodes in natürlicher Sprache erläutern. Wenn Sie Quellcode lesen, können Sie zwar (meistens) verstehen, *was* dieser tut, aber nicht immer, *warum* er dies tut. Kommentare sind für eine Programmiererin die einzige Möglichkeit soll, und einer Zeichenkette, die als Endzeichen verwendet werden soll, müssen wir den Namen des Argumentes (`end=`) explizit angeben.

ihre Gedanken mitzuteilen. Kommentare sollen Einsicht in die Absichten und Überlegungen der Programmiererin geben.

Gute Kommentare sind tatsächlich essentiell für die Qualität eines Programmes: Der Quellcode eines Programmes muss oftmals modifiziert oder erweitert werden. Kommentare helfen dabei, den Quellcode (auch Jahre nach dessen Entwicklung) schneller und besser zu verstehen (insbesondere – aber nicht nur – wenn dieser von anderen Programmierern oder Programmiererinnen geschrieben wurde).

Wenn Sie nochmals den Quellcode unseres ersten Beispiels betrachten, sehen Sie, dass wir zwei verschiedene Kommentartypen verwenden: *Reguläre Kommentare* und sogenannte *Docstrings*.

Regulärer Python Kommentar beginnt mit einem Hash-Zeichen (#) und geht bis zum Ende der Zeile. Typischerweise verwenden wir Kommentare, um den *nachfolgenden* Code zu erklären:

```
# 5% Fehlertoleranz einbauen  
value = value * 1.05
```

Wenn ein Kommentar in der gleichen Zeile wie eine Anweisung steht, wird er als *Inline-Kommentar* bezeichnet. Auch Inline-Kommentar beginnt mit einem einzelnen Hash-Zeichen:

```
value = value * 1.05 # 5% Fehlertoleranz einbauen
```

Die zweite Kommentarmöglichkeit beginnt und endet mit einem dreifachen Anführungszeichen """ . Solche Kommentare werden als *Docstrings* bezeichnet. Python bietet zwei Arten von *Docstrings* an: einzeilige und mehrzeilige.

```
""" Sortiert die Liste aufsteigend """
```

```
"""
Erhöhen des Wertes gemäss Index
Index 1 - 2 keine Erhöhung
Index 3 - 4 Erhöhung um 5%
Index 4 - 6 Erhöhung um 10%
"""
```

Egal welchen Kommentartyp Sie verwenden: Kommentare sollen prägnant und in ganzen Sätzen formuliert werden, sollen nicht das offensichtliche kommentieren und müssen eindeutig sein.

## 1.3 Programmierung

### 1.3.1 Kompilieren und Interpretieren

*Maschinensprachen* sind Programmiersprachen, in denen die Instruktionen definiert sind, die vom Prozessor eines Computers direkt ausgeführt werden können. Jeder Prozessortyp besitzt seine eigene Maschinensprache. Einzelne Anweisungen einer Maschinensprache können nur sehr einfache Aktionen ausführen (z.B. Kopieren eines Wertes in einen Speicherregister oder Vergleichen eines bestimmten Wertes mit Null). Jeder Befehl der Maschinensprache ist durch einen oder mehrere binäre Zahlenwerte codiert. Eine sinnvolle Folge von solchen binären Zahlencodes bildet der Quellcode, der von einem Computer ausgeführt werden kann. Das Programmieren in Maschinensprache ist eher schwierig, fehleranfällig und vor allem langwierig.

Um eine Anwendung zu programmieren, verwendet man typischerweise eine bestimmte *Hochsprache* (z.B. Java, Ruby oder eben Python).

Hochsprachen ermöglichen das Schreiben von Quellcode in “Englisch-Ähnlichen” Sätzen, die relativ einfach zu verstehen und zu lesen sind (auf jeden Fall einfacher als eine Folge von Binärzahlen). Damit aber ein Programm auf einem Computer ausgeführt werden kann, muss dieses zwingend in der entsprechenden Maschinensprache ausgedrückt sein. Das bedeutet, dass Quellcode, welcher in einer anderen Sprache als Maschinencode verfasst wurde, vor seiner Ausführung in Maschinensprache übersetzt werden muss.

*Kompilierer* (engl. *Compiler*) sind Computerprogramme, die Quellcode, geschrieben in einer bestimmten Sprache *A*, in eine Zielsprache *B* übersetzen können (es braucht unterschiedliche Kompilierer für unterschiedliche Quell- und Zielsprachen). Oftmals ist die Zielsprache die Maschinensprache eines bestimmten Computers. Beachten Sie, dass eine einzige Anweisung in einer Hochsprache (z.B. die Anweisung `print`) vielen – vielleicht Hunderten – von Maschineninstruktionen entsprechen kann. Der Kompilierer erzeugt diese Maschineninstruktionen automatisch aus dem Quellcode.

Python ist eine *interpretierte Sprache*, die *keine* Kompilierung erfordert. *Interpreter* sind ebenfalls Computerprogramme, die eine Abfolge von Anweisungen scheinbar direkt ausführen. Ein Interpreter liest dazu eine oder mehrere Dateien mit Quellcode ein, analysiert diese und führt sie anschliessend Anweisung für Anweisung aus, indem er diese in die Maschinensprache übersetzt, die der Computer ausführen kann. Interpreter sind deutlich langsamer als Kompilierer, bieten jedoch andere Vorteile.

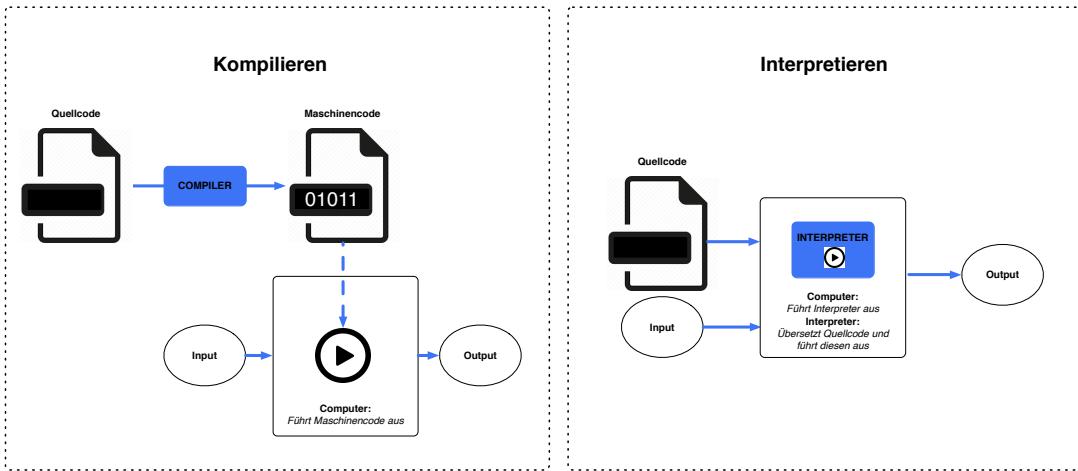


Abbildung 1.1: Kompilieren vs. Interpretieren: Kompilieren: (1) Quellcode wird durch Compiler in eine Maschinensprache übersetzt (2) Maschinencode kann auf Computern ausgeführt werden, welche diese Maschinensprache verstehen. Interpretieren: (1) Computer führt den Interpreter aus (2) Interpreter übersetzt Zeile für Zeile Quellcode in Maschinencode und führt diesen Schritt für Schritt aus

Zu den grössten Vorteilen von interpretiertem Quellcode zählt die Unabhängigkeit von einer vorher festgelegten Maschinensprache. Der von einem Kompilierer für einen Prozessortyp erzeugte Maschinencode läuft nicht auf einem anderen Prozessortyp, ohne neu zu kompilieren. Interpretierter Quellcode hingegen läuft ohne Änderung auf jedem System, auf dem es einen entsprechenden Interpreter gibt.

**Beispiel 1** In Abb. 1.1 werden die Prozesse des Kompilierens und des Interpretierens gegenübergestellt.

### 1.3.2 Python Interaktiv Verwenden

Der Interpreter von Python kann in zwei Modi verwendet werden:

- Interaktiver Modus
- Skript Modus

Die Möglichkeit, den Python Interpreter interaktiv einzusetzen, macht es einfach, mit den Funktionen der Sprache Python zu experimentieren, Wegwerfprogramme zu schreiben oder kleine Teile des Quellcodes während der Programmierung schnell zu testen.

Ist der Python Interpreter installiert, ist es möglich, diesen im interaktiven Modus durch Eingabe des folgenden Befehls in einer Konsole zu starten (unter Mac OS heißt die Konsole *Terminal* und auf Windows Geräten *Eingabeaufforderung*)<sup>6</sup>:

```
python
```

Durch Eingabe von **Control-D** (unter Mac OS) oder **Control-Z** (unter Windows) kann der Interpreter beendet werden (alternativ: folgenden Befehl eingeben und mit der Eingabetaste bestätigen: `quit()`).

Beim Starten des Interpreters wird eine Begrüßungsansage (mit Angabe der Versionsnummer und eines Copyright-Vermerks) ausgegeben, bevor der Interpreter den ersten *primären Prompt* ausgibt:

```
>>>
```

Im interaktiven Modus wird mit dem *primären Prompt* nach der nächsten Anweisung gefragt. Falls sich eine Anweisung über mehrere Zeilen erstrecken sollte, wird mit dem *sekundären Prompt*

```
...
```

---

<sup>6</sup>Den Befehl müssen Sie mit der Eingabetaste bestätigen. Unter Mac OS: `python3` eintippen

Versionsanzeige und Begrüssung

Befehl zum Starten des Python Interpreters

Konsole (Terminal)

Primärer Prompt

Ergebnis einer Anweisung

Anweisungen

```
rza@Kaspars-iMac-2 ~ % python3
Python 3.10.0 (v3.10.0:b494f5935c, Oct 4 2021, 14:59:20) [Clang 12.0.5
(clang-1205.0.22.11)] on darwin
[Type "help", "copyright", "credits" or "license" for more information.
] >>> 1+1
2 >>> quit()
rza@Kaspars-iMac-2 ~ %
```

Abbildung 1.2: Den Python Interpreter im Terminal von Mac OS starten/beenden.

nach Fortsetzungszeilen gefragt.

In den Beispielen im Skript unterscheiden sich Input und Output durch das Vorhandensein oder Fehlen der Prompts (>>> und . . .): Um die Beispiele nachzuvollziehen, müssen Sie alles nach dem Prompt eingeben. Zeilen, die nicht mit einer Prompt beginnen, werden vom Interpreter ausgegeben.

Zum Beispiel kann der Python Interpreter als einfacher Taschenrechner verwendet werden.

**Beispiel 2** In Abb. 1.2 wird der Python Interpreter in einer Konsole gestartet, danach wird mit Python die Addition  $1 + 1$  berechnet und mit dem Befehl `quit()` wird der Interpreter wieder verlassen.

Ein Ausdruck (engl. *Expression*) ist eine Kombination von einem oder mehreren Operatoren und Operanden. Die Syntax von arithmetischen

*Ausdrücken* in Python ist einfach: Die Operatoren `+`, `-`, `*` und `/` funktionieren wie in den meisten anderen Sprachen:

### Beispiel 3

```
>>> 2 + 2  
4  
>>> 5 + 2 * 4  
13  
>>> 10 / 2 - 3  
2.0  
>>> 8 + 12 * 2 - 4  
28
```

Mit dem Operator `**` können Sie Potenzen berechnen (beachten Sie die Verwendung von Inline Kommentaren):

### Beispiel 4

```
>>> 5 ** 2 # 5 hoch 2  
25  
>>> 2 ** 7 # 2 hoch 7  
128
```

Um den Rest einer Division zu berechnen, können Sie den *Modulo-Operator* `%` verwenden. Dieser Operator gibt den Rest zurück, der entsteht, wenn man den ersten Operanden durch den zweiten Operanden dividiert. Zum Beispiel ergibt

`18 % 4 = 2`

da  $18 / 4 = 4$  mit Rest 2. Der Modulo-Operator kann auch auf Gleitkommazahlen angewendet werden. Dies kann insbesondere hilfreich sein, wenn man den Nachkommabereich einer Zahl extrahieren möchte.

### Beispiel 5

```
>>> 12.34 % 1  
0.34
```

Die Prioritäten der Python-Operatoren sind wie gewohnt definiert:

1. Potenzierung (\*\*)
2. Multiplikation (\*), Division (/) und Modulo (%)
3. Addition (+) und Subtraktion (-)

Haben die Operatoren die gleiche Priorität, werden diese von links nach rechts ausgewertet. Mit Klammern können Sie Teilausdrücke gruppieren und priorisieren.

### Beispiel 6

```
>>> (5 + 2) * 4  
28  
>>> 10 / (5 - 3)  
5.0  
>>> 8 + 12 * (6 - 2)  
56  
>>> (6 - 3) * (2 + 7) / 3  
9.0
```

Die arithmetischen Operationen sind für ganze Zahlen und für Gleitkommazahlen definiert. Die Resultate arithmetischer Ausdrücke passen sich dabei i.d.R. den Operanden an. Operationen mit gemischten

Operanden (ganze Zahlen und Gleitkommazahlen) konvertieren ganz-zahlige Operanden in Gleitkommazahlen:

### Beispiel 7

```
>>> 4 * 3  
12  
>>> 4 * 3.75 - 1  
14.0
```

Eine Division (/) gibt aber *immer* eine Gleitkommazahl zurück (selbst wenn beide Operanden ganze Zahlen sein sollten). Eine Division, bei der das Resultat auf die nächste ganze Zahl abgerundet werden soll, ist mit dem Operator // möglich. So wird beispielsweise der Ausdruck 11 // 4 zu 2 ausgewertet. Beachten Sie, dass sich der Ausdruck (-11) // 4 zu -3 auswertet, weil -2.75 nach “unten” abgerundet wird.

### Beispiel 8

```
>>> 10 / 2 # Division gibt immer eine Gleitkommazahl zurück  
5.0  
>>> 17 // 3 # Ganzzahldivision verwirft den Bruchteil  
5
```

Die im interaktiven Modus eingegebenen Anweisungen werden *nicht* als Programm gespeichert. Ein Programm, das wir zu einem beliebigen späteren Zeitpunkt (nochmals) ausführen lassen können, müssen wir im *Skript Modus* schreiben.

### 1.3.3 Python im Skript Modus Verwenden

Im Skript Modus schreiben wir Python Programme in Dateien. Diese Programme können dabei mit jedem beliebigen Texteditor geschrieben

werden, der reine Textdateien (also Dateien mit der Endung `.txt`) speichern kann (z.B. `TextEdit`, `TextMate`, `Atom` oder ähnliche Programme). Das Interpretieren von Python Programmen kann dann in der Konsole des Computers durchgeführt werden.

Nehmen wir an, dass der Quellcode in einer Datei `simple.py` im Ordner `/Users/rza/Desktop/` gespeichert ist (`simple.py` enthält eine einzige Zeile Quellcode, nämlich einen Aufruf der eingebauten Funktion `print`). Wir öffnen die Konsole und wechseln in den Ordner, in dem sich die Datei mit dem Quellcode befindet. Dies geschieht mit dem Befehl `cd` (*change directory*):

```
cd /Users/rza/Desktop/
```

Damit sich das Python Programm `simple.py` ausführen lässt, müssen wir nun in der Konsole den Befehl

```
python3 simple.py
```

eingeben und mit der Eingabetaste bestätigen.

Der Python Interpreter liest nun Anweisung für Anweisung aus dem Quellcode, übersetzt diese in Maschinensprache und führt die erzeugten Befehle auf dem Prozessor aus. Ist das Programm fehlerfrei, wird das Programm komplett ausgeführt und danach wird das Programm terminieren.

**Beispiel 9** In Abb. 1.3 ist der gesamte Prozess, der in der Konsole durchgeführt wird, illustriert.

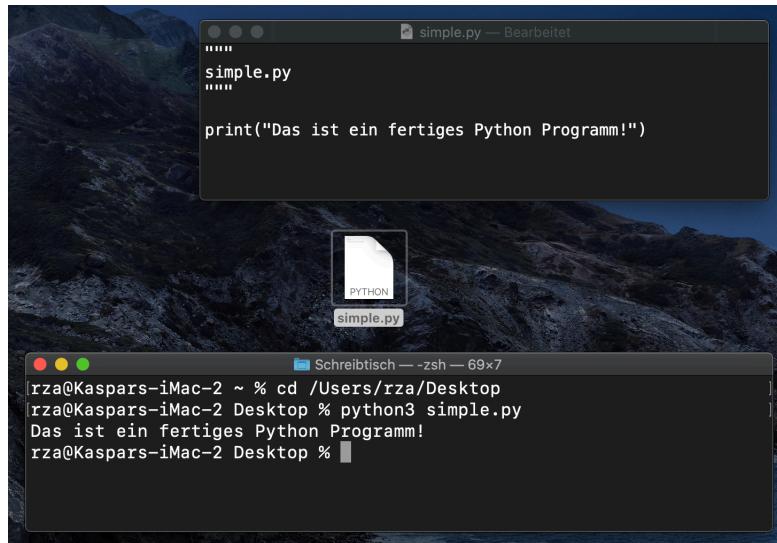


Abbildung 1.3: Python Programme in einer Konsole interpretieren und ausführen.

Eine *Entwicklungsumgebung* beschreibt die Menge von Werkzeugen, die man zum Erstellen, Testen, Modifizieren und Ausführen von Quellcode benötigt. Eine Entwicklungsumgebung heisst *Integrierte Entwicklungsumgebung (IDE)* für *Integrated Development Environment*), wenn diese mehrere Werkzeuge in einer Software vereint. IDEs bieten oftmals graphische Oberflächen und zusätzliche Fähigkeiten, die das Programmieren stark vereinfachen können. Zum Beispiel haben Sie in IDEs die Möglichkeit, Ihre *Python Module* in Paketen zu verwalten, IDEs bieten i.d.R. eine *integrierte Konsole* für Ein- und Ausgaben und IDEs *unterstützen* einem bei der Fehlersuche (um nur einige Vorteile zu nennen).

Es existieren zahlreiche IDEs, die zur Python Programmentwicklung verwendet werden können, wie z.B. *PyCharm* oder *Eclipse*. Wir verwenden in dieser Vorlesung die IDE *PyCharm*<sup>7</sup> (siehe Abb. 1.4).

---

<sup>7</sup>Von PyCharm gibt es eine frei erhältliche Version (*Community Edition*). Sie dürfen eine andere IDE verwenden – beachten Sie hierbei aber, dass wir bei technischen Problemen mit Ihrer IDE ggf. nicht helfen können.

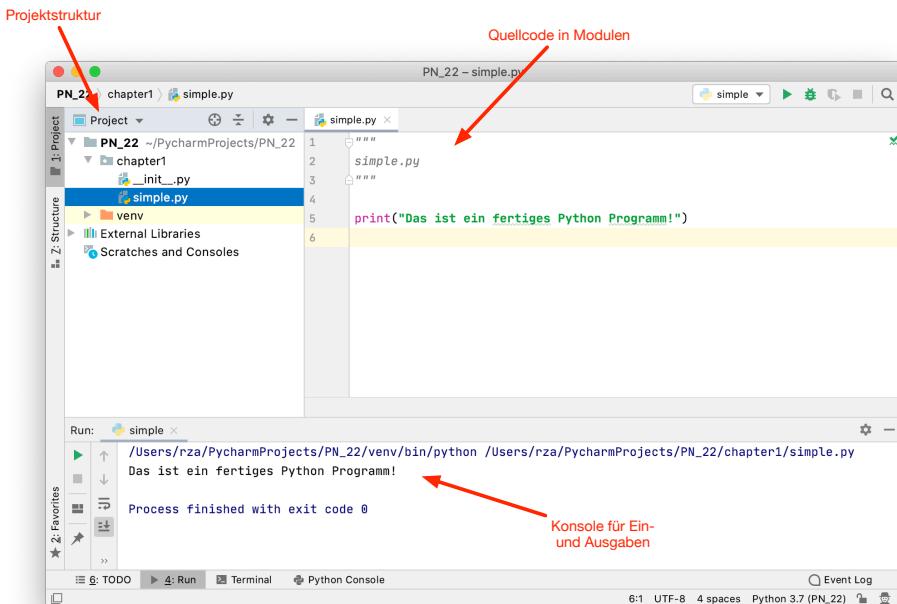


Abbildung 1.4: Python Programme in PyCharm erstellen, organisieren und ausführen.

### 1.3.4 Programmierfehler

Jede Programmiersprache besitzt ihre eigene *Syntax*. Die Syntax einer Programmiersprache definiert, wie die Elemente der Sprache kombiniert werden dürfen, um gültige Programmieranweisungen zu bilden.

Die *Semantik* einer Programmieranweisung definiert, was passieren soll, wenn die Anweisung ausgeführt wird. Die Semantik beschreibt also die Bedeutung von Quellcode. Ein syntaktisch korrektes Programm muss nicht zwingend semantisch korrekt sein – ein Programm wird immer das tun, was wir tatsächlich programmiert haben und nicht das, was wir gemeint haben zu programmieren.

Die Syntax definiert, *wie* ein Programm geschrieben werden darf, die Semantik hingegen definiert, *was* ein geschriebenes Programm tun

```

    /Users/rza/PycharmProjects/PN_22/venv/bin/python /Users/rza/PycharmProjects/PN_
File "/Users/rza/PycharmProjects/PN_22/chapter1/simple.py", line 6
    ^
SyntaxError: unexpected EOF while parsing

```

Abbildung 1.5: Fehler in der Syntax führen zu Fehlerausgaben während der Ausführung des Programmes.



Abbildung 1.6: PyCharm überprüft Teile der Syntax bereits bei der Erstellung eines Programmes.

wird. Dementsprechend kann man zwei Fehlerarten unterscheiden:

- Fehler in der Syntax
- Fehler in der Semantik

Wird ein Python Programm interpretiert, werden alle Regeln der Syntax kontrolliert. Jeder Fehler, der auf das Nichteinhalten einer Syntaxregel zurückzuführen ist, heisst *Syntaxfehler*. Falls ein Python Programm syntaktisch nicht korrekt ist (ein einziger Syntaxfehler reicht hierzu schon aus), erzeugt der Interpreter zur Laufzeit eine Fehlermeldung und das Programm terminiert (siehe Abb. 1.5).

Integrierte Entwicklungsumgebungen überprüfen bereits beim Erstellen von Quellcode Teile der Syntax und markieren entdeckte Fehler (siehe Abb. 1.6).

Die zweite Fehlerart umfasst *logische Fehler* oder *semantische Fehler*. In diesem Fall ist der Quellcode syntaktisch korrekt und das Programm kann ausgeführt werden, aber es produziert fehlerhafte Ausgaben oder

das Programm terminiert nicht ordnungsgemäss. Z.B. berechnet das Programm den Mittelwert falsch oder nach einer bestimmten Eingabe des Benutzers stürzt das Programm einfach ab.

Wir sollten unsere Programme immer ausführlich testen und die erwarteten Ergebnisse mit den tatsächlichen Ergebnissen vergleichen. Ausserdem liegt es an uns, möglichst robuste Programme zu schreiben, welche verhindern, dass das Programm bei einer fehlerhaften oder unerwarteten Manipulation des Benutzers gleich abstürzt.

Wenn logische Fehler beobachtet werden, so muss der Ursprung des Problems im Quellcode gefunden werden – dieser Prozess ist unter dem Namen *Debugging* bekannt und kann manchmal eine langwierige Aufgabe sein.

# Kapitel 2

## Variablen und Listen

### 2.1 Variablen

Bis jetzt haben wir für die Operanden in unseren arithmetischen Ausdrücken direkt Werte angegeben:

#### Beispiel 10

```
>>> 8 + 3  
11
```

Dies ist i.d.R. zu unflexibel. Ferner benötigen wir die Möglichkeit, die Ergebnisse von Ausdrücken speichern zu können. Deshalb führen wir in diesem Kapitel das Konzept von Variablen ein.

Jede *Variable* besitzt einen eindeutigen Namen, den die Programmiererin bei der Programmentwicklung festlegt. Dieser Name wird in der Programmierung *Bezeichner* (engl. *Identifier*) genannt. Eine Variable ist im Wesentlichen ein Platzhalter, der auf einen Speicherort zeigt, der einen bestimmten Wert enthalten kann. Ist eine Variable erstmal definiert, können Sie mit dem Bezeichner der Variablen den zugehörigen Wert auslesen oder den zugehörigen Wert verändern.

Ein gültiger Bezeichner beginnt in Python mit einem Buchstaben A bis Z oder a bis z oder einem Unterstrich \_, gefolgt von null oder mehr Buchstaben, Unterstrichen und Ziffern (0 bis 9).

Beachten Sie, dass Sie keine *reservierten Schlüsselwörter* als Bezeichner verwenden dürfen. Reservierte Schlüsselwörter sind Wörter, die für bestimmte Zwecke der Programmiersprache reserviert sind und somit nicht für andere Zwecke verwendet werden können. Aktuell sind in Python 35 Wörter reserviert. Alle Schlüsselwörter ausser `True`, `False` und `None` werden mit Kleinbuchstaben geschrieben:

<code>False</code>	<code>await</code>	<code>else</code>	<code>import</code>	<code>pass</code>
<code>None</code>	<code>break</code>	<code>except</code>	<code>in</code>	<code>raise</code>
<code>True</code>	<code>class</code>	<code>finally</code>	<code>is</code>	<code>return</code>
<code>and</code>	<code>continue</code>	<code>for</code>	<code>lambda</code>	<code>try</code>
<code>as</code>	<code>def</code>	<code>from</code>	<code>nonlocal</code>	<code>while</code>
<code>assert</code>	<code>del</code>	<code>global</code>	<code>not</code>	<code>with</code>
<code>async</code>	<code>elif</code>	<code>if</code>	<code>or</code>	<code>yield</code>

Python unterscheidet zwischen Gross- und Kleinschreibung (engl. *case sensitive*). Das heisst, `Total`, `total` und `TOTAL` sind in Python verschiedene Bezeichner. Es gibt zahlreiche Konventionen für die Gross- und Kleinschreibung von Bezeichnern<sup>1</sup>.

Für Bezeichner von Variablen verwenden wir die Konvention `lowercase` oder `lowercase_with_underscores` (falls der Bezeichner aus mehreren Worten bestehen sollte). Die gleiche Konvention verwenden wir für

---

<sup>1</sup>Diese Konventionen sind aber *nicht* durch die Programmiersprache Python vorgeschrieben – Konventionen sind Abmachungen zwischen Programmierer:innen, die das Lesen von Quellcode vereinfachen sollen.



Abbildung 2.1: Variablen referenzieren Daten im Speicher.

Bezeichner für Module, Pakete und später auch für eigene Funktionen. Bezeichner sollten zudem kurz (`product` statt `the_current_product`), aussagekräftig (`grade` statt `x`) und eindeutig sein.

Das Gleichheitszeichen (=) wird verwendet, um einer Variablen einen Wert zuzuweisen. Wird eine sogenannte *Zuweisungsoperation* ausgeführt, so wird die rechte Seite des Zuweisungsoperators (=) ausgewertet und das Resultat wird im Speicher an den Platz gelegt, auf den die Variable auf der linken Seite zeigt.

### Beispiel 11

```
>>> pages = 256
>>> x = 8.4
```

Nach Ausführung dieser beiden Zuweisungen zeigen die Variablen `pages` und `x` auf die Werte 256 bzw. 8.4 (siehe auch Abb. 2.1).

Der Datentyp einer Variablen muss – im Gegensatz zu anderen Programmiersprachen – in Python nicht explizit angegeben werden. Der Datentyp einer Variablen leitet sich automatisch aus dem Typ des zugewiesenen Wertes ab.

Python besitzt verschiedene eingebaute Datentypen, die wir als Programmierer:innen nutzen können. Vier einfache Datentypen, die in Python verfügbar sind, sind bspw.:

- `str` repräsentiert Zeichenketten ("Grün", "Hi", "BRAVO", etc.)

- `int` repräsentiert ganze Zahlen (-1, 42, 12345, etc.)
- `float` repräsentiert Gleitkommazahlen (1.0, 3.14, -123.456, etc.)
- `bool` repräsentiert Wahrheitswerte (`True` oder `False`)

Die folgenden Variablen `price` und `title` sind also bspw. vom Typ `float` (Gleitkommazahl) und `str` (Zeichenkette).

```
price = 17.95
title = "Python Grundkurs"
```

Wir können in Python mehreren Variablen gleichzeitig (d.h. auf einer Zeile) Werte zuweisen:

```
pages, price, title = 256, 17.95, "Python Grundkurs"
```

Wir können einer Variablen auch gleichzeitig mehrere Werte zuweisen<sup>2</sup>:

```
info = 256, 17.95, "Python Grundkurs"
```

Zudem erlaubt es Python, mehreren Variablen gleichzeitig den gleichen Wert zuzuweisen:

```
x = y = z = 1
```

Die Bezeichner von Variablen können wir nach deren Definition in unserem Quellcode verwenden – wir sagen, wir *referenzieren* eine Variable. Wird eine Variable referenziert, so wird der Wert, auf den diese Variablen zeigt, verwendet.

---

<sup>2</sup>Hierbei wird ein sogenanntes *Tupel* definiert – Details zum Umgang mit Tupeln folgen in Kapitel 6.

**Beispiel 12** Betrachten Sie das folgende Modul *meaning*:

```
"""
meaning.py
"""

question = "Sinn des Lebens?"
answer = 42

print(question, answer)
```

Wir weisen den Variablen *question* und *answer* je einen Wert zu. Beim Aufruf der Funktion *print* referenzieren wir beide Variablen und das Programm wird somit folgende Ausgabe generieren:

*Sinn des Lebens? 42*

Der Zuweisungsoperator (=) hat die kleinere Priorität als alle arithmetischen Operationen. Die gesamte rechte Seite einer Zuweisung wird also immer zuerst berechnet und danach erst der Variablen auf der linken Seite des Zuweisungsoperators zugewiesen (siehe Abb. 2.2).

**Beispiel 13** Die Ausgabe des folgenden Programmes *area.py* ist demnach *Fläche = 800*.

```
"""
area.py
"""

width = 20
height = 40
area = width * height
print("Fläche =", area)
```

Die rechte und linke Seite einer Zuweisung dürfen die gleiche Variable beinhalten. Das heisst, dass zum Beispiel folgende Anweisung syntaktisch korrekt ist<sup>3</sup>:

<sup>3</sup>Wenn Sie andere Programmiersprachen kennen, kennen Sie vielleicht das Inkrement ++ und das Dekrement --. Diese Operatoren sind in Python nicht vorgesehen.

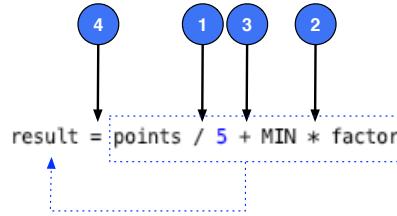


Abbildung 2.2: Die Reihenfolge einer Auswertung: Der Zuweisungsoperator `=` hat die niedrigste Priorität.

```
count = count + 1
```

Zuerst wird 1 zum ursprünglichen Wert von `count` addiert und danach wird das berechnete Resultat in `count` gespeichert (das heisst, der alte Wert von `count` wird überschrieben). Beachten Sie insbesondere den Unterschied zur Anweisung `count + 1`, welche *keine* Auswirkung auf die Variable `count` hat.

### Beispiel 14

```
>>> count = 1
>>> count = count + 1
>>> count
2
>>> count + 1
3
>>> count
2
```

Oftmals müssen auf Variablen Operationen ausgeführt werden und das Resultat dieser Operation soll danach wieder in der gleichen Variablen gespeichert werden. Zum Beispiel möchten wir zur Variablen `num` den Wert der Variablen `count` addieren. Hierzu schreiben wir fol-

gende Anweisung:

```
num = num + count
```

Der Einfachheit halber existieren in Python *Zuweisungsoperatoren*, welche diesen Vorgang erleichtern. Zum Beispiel ist die Anweisung

```
num += count
```

äquivalent zu der obigen Zuweisung.

Es existieren unterschiedliche Zuweisungsoperatoren in Python, unter anderem die folgenden:

- $x += y$  entspricht  $x = x + y$
- $x -= y$  entspricht  $x = x - y$
- $x *= y$  entspricht  $x = x * y$
- $x /= y$  entspricht  $x = x / y$

Die rechte Seite einer Zuweisungsoperation kann ein komplexer Ausdruck sein. Dieser Ausdruck wird zuerst komplett ausgewertet und erst dann mit der linken Seite kombiniert (und in der Variablen gespeichert). Das bedeutet, dass z.B.

```
total += (points - 10) / maximum
```

äquivalent ist zu

```
total = total + ((points - 10) / maximum)
```

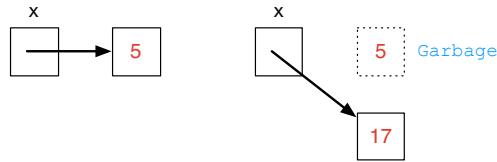


Abbildung 2.3: Werte, die nicht mehr referenziert werden, werden als *Garbage* markiert.

Wenn eine Variable nicht definiert ist, wird der Versuch, sie zu verwenden, zu einem Fehler führen:

### Beispiel 15

```
>>> x # Der Variablen x wurde noch kein Wert zugewiesen
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Variablen können während der Programmausführung verschiedene Werte referenzieren – zu jedem Zeitpunkt wird aber immer nur ein Wert referenziert.

### Beispiel 16

```
>>> x = 5
>>> x
5
>>> x = 17
>>> x
17
```

Sobald ein Wert im Speicher von keiner Variablen mehr referenziert wird, wird dieser Wert vom Python Interpreter automatisch als *Garbage* deklariert und der entsprechende Speicherplatz wird wieder freigegeben (siehe Abb. 2.3).

**Beispiel 17** Wir betrachten das Python Programm `polygon.py`, das den Wert einer Variablen zur Laufzeit des Programmes ändert:

```
"""
polygon.py
"""

sides = 3
print("Anzahl Seiten eines Trigons:")
print(sides)

sides = 12
print("Anzahl Seiten eines Dodekagons:")
print(sides)

print("Anzahl Seiten eines Ikosagons:")
print(sides + 1) # LESEN ändert eine Variable nicht!

print("In der Variablen Sides ist gespeichert:")
print(sides)
```

In diesem Programm wird zunächst der Variablen `sides` der Wert 3 zugewiesen. Danach wird der aktuelle Wert der Variablen `sides` ausgegeben. Die nächste Anweisung überschreibt den gespeicherten Wert der Variablen `sides` mit einer Zuweisung auf den Wert 12. Die ersten Ausgaben des Programmes `polygon` lauten deshalb:

*Anzahl Seiten eines Trigons:*

3

*Anzahl Seiten eines Dodekagons:*

12

Wird eine Variable aber nur verwendet (z.B. innerhalb einer arithmetischen Operation), ohne dass eine Zuweisung stattfindet, so bleibt der Wert der Variablen unverändert: Lesen von Daten verändert die Daten niemals – nur mit Zuweisungen können Sie die Werte von Va-

riablen ändern!

Betrachten Sie zum Beispiel die Anweisung

```
print(sides + 1)
```

Hier wird die Variable `sides` nur gelesen und deshalb bleibt diese unverändert. Die weiteren Ausgaben des Programmes `polygon.py` lauten:

Anzahl Seiten eines Ikosagons:

13

In der Variablen `sides` ist gespeichert:

12

In Python darf man einer Variablen nacheinander Werte eines beliebigen Typs zuweisen (siehe auch Abb. 2.4):

### Beispiel 18

```
>>> message = 99
>>> message
99
>>> message = "Hallo Python"
>>> message
'Hello Python'
```

Mit der in Python eingebauten Funktion `type` können Sie den Datentyp einer Variablen erfragen:

### Beispiel 19

```
>>> message = 99
```

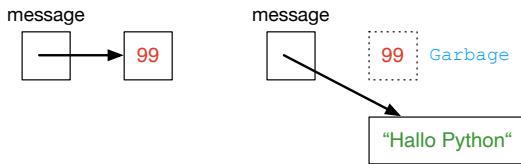


Abbildung 2.4: Eine Variable kann Daten unterschiedlicher Typen referenzieren.

```
>>> type(message)
<class 'int'>
>>> message = "Hallo Python"
>>> type(message)
<class 'str'>
```

Es existieren eingebaute Funktionen zur expliziten Konvertierung von Variablen. So können Sie zum Beispiel mit den Funktionen `int(x)` und `float(x)` den Datentyp der Variablen `x` ändern. Beachten Sie, dass bei der Konvertierung einer Gleitkommazahl in eine ganze Zahl der Nachkommabereich einfach *abgeschnitten* wird.

### Beispiel 20

```
>>> x = 4.9
>>> x = int(x)
>>> x
4
>>> x = 7
>>> x = float(x)
>>> x
7.0
```

Mit der eingebauten Funktion `str(x)` können Sie eine ganze Zahl in eine Zeichenkette konvertieren. Das "Entpacken" einer Zahl aus einer Zeichenkette ist ebenso möglich:

## Beispiel 21

```
>>> x = 4
>>> x = str(x)
>>> type(x)
<class 'str'>
>>> x
'4'
>>> x = int(x)
>>> type(x)
<class 'int'>
>>> x
4
```

### 2.1.1 Eingaben Entgegennehmen

Als nächstes führen wir eine hilfreiche eingebaute Funktion ein – die Funktion `input`. Diese Funktion erwartet als Parameter eine Zeichenkette, welche zunächst ausgegeben wird. Danach wartet das Programm (bzw. die Funktion) auf eine Eingabe des Benutzers über die Tastatur. Eine Eingabe des Benutzers (abgeschlossen mit der Eingabetaste) wird schliesslich von `input` als Zeichenkette zurückgegeben. Diese Rückgabe kann dann z.B. einer Variablen zugewiesen werden.

**Beispiel 22** Betrachten Sie das folgende Programm.

```
"""
echo.py
"""

message = input("Ihre Nachricht: ")
print("Mein Echo: ", message)
```

Eine mögliche Ein- und Ausgabe des Programmes lautet:

*Ihre Nachricht: Hallo zusammen!*

*Mein Echo: Hallo zusammen!*

Die Rückgabe der Funktion `input` ist *immer* eine Zeichenkette. Wollen Sie vom Benutzer eine Zahl entgegennehmen, so müssen Sie die Eingabe konvertieren.

**Beispiel 23** Betrachten Sie das folgende Programm, in dem die Eingabe zunächst in eine ganze Zahl konvertiert, danach verdoppelt und schliesslich ausgegeben wird:

```
"""
double.py
"""

value = input("Ganze Zahl eingeben: ")
value = int(value)

doubled_value = value * 2
print("Das Doppelte der Zahl", value, "=", doubled_value)
```

Mögliche Ein- und Ausgabe:

*Ganze Zahl eingeben: 7*

*Das Doppelte der Zahl 7 = 14*

Durch Verschachtelung der Funktionen können Sie die Rückgabe der Funktion `input` direkt als Parameter an die Funktion `int` weitergeben.

```
value = int(input("Ganze Zahl eingeben: "))
```

## 2.2 Listen – die Datenstruktur `list`

Bis jetzt zeigen unsere Variablen auf einzelne Werte (z.B. auf Werte vom Typ `int` oder `str`). Python sieht eine Reihe von Datenstrukturen vor, mit denen *mehrere* Werte oder Variablen in einem Behälter zusammengefasst werden können. In diesem Abschnitt betrachten wir hierzu ein erstes Beispiel.

Die eingebaute Datenstruktur `list` ist der allgemeinste Behältertyp und definiert eine Liste mit kommagetrennten Werten (Elementen) innerhalb eckiger Klammern. In Python dürfen Listen Elemente verschiedener Datentypen enthalten<sup>4</sup>.

**Beispiel 24** Wir definieren drei Listen (siehe auch Abb. 2.5):

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
>>> names = ["Maxime", "Emilie", "Noelle", "Eliane"]
>>> names
['Maxime', 'Emilie', 'Noelle', 'Eliane']
>>> info = ["Alicia", 27, 1550.87]
>>> info
['Alicia', 27, 1550.87]
```

Die eingebaute Funktion `range([start,] end[, step])` ist sehr nützlich im Umgang mit Listen<sup>5</sup>. Die Funktion `range` erzeugt arithmetische Bereiche von `0` bis `end` (der angegebene Endpunkt ist nie Teil des

---

<sup>4</sup>Obschon das eher nicht üblich ist.

<sup>5</sup>Die eckigen Klammern bedeuten, dass die Parameter `start` und `step` optional sind, nicht, dass Sie an dieser Stelle eckige Klammern eingeben sollten.

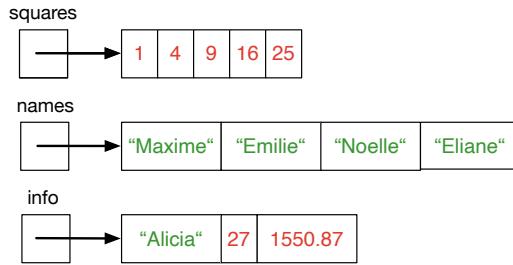


Abbildung 2.5: Drei Listen.

Bereiches). Optional können Sie den Bereich bei einer anderen Zahl **start** beginnen lassen oder eine andere Schrittweite **step** verwenden:

### Beispiel 25 Beispiele von Bereichen:

- `range(5)` erzeugt den Bereich 0, 1, 2, 3, 4
- `range(5, 10)` erzeugt den Bereich 5, 6, 7, 8, 9
- `range(0, 10, 3)` erzeugt den Bereich 0, 3, 6, 9
- `range(-10, -100, -30)` erzeugt den Bereich -10, -40, -70

Mit einer Konvertierung durch die eingebaute Funktion `list`, können Sie aus einem `range` Objekt ein Objekt vom Typ `list` erzeugen:

### Beispiel 26

```
>>> values = list(range(4))
>>> values
[0, 1, 2, 3]
>>> numbers = list(range(1, 10, 2))
>>> numbers
[1, 3, 5, 7, 9]
```

Die eingebaute Funktion `len` kann verwendet werden, um die Anzahl Elemente einer Liste auszulesen, während die eingebauten Funktionen `min` und `max` das Minimum bzw. das Maximum einer Liste ermitteln:

## Beispiel 27

```
>>> vals = [-7, 3, 8, 99, 0]
>>> len(vals)
5
>>> min(vals)
-7
>>> max(vals)
99
```

Mit ganzzahligen *Indizes* innerhalb eckiger Klammern können wir in Listen einzelne Elemente referenzieren. Die Syntax hierzu lautet:

`listen_bezeichner[index]`

Das Zählen beginnt dabei bei 0. Der Index des ersten Elementes in der Liste ist also 0, das zweite Element hat Index 1, und das  $n$ -te Element hat Index  $n-1$ . Negative Indizes kennzeichnen Positionen relativ zum Ende der Liste (der Index -1 kennzeichnet z.B. das letzte Element, -2 das vorletzte Element, etc.)

## Beispiel 28

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares[1] # gibt Wert mit Index 1 zurück
4
>>> squares[-1] # Referenz auf das letzte Element
25
```

Sogenanntes *Slicing* verwendet folgende Syntax:

```
listen_bezeichner[[start]:[end]:[step]]
```

Alle Slicing-Operationen liefern eine Kopie der Liste mit den angeforderten Elementen vom Index `start` bis zum Index `end` (nicht inklusive) mit der gegebenen Schrittweite `step`. Alle drei Parameter sind optional (wiederum durch die eckigen Klammern angegeben). Falls diese nicht angegeben werden, werden folgende Standardwerte verwendet:

- `start`: 0 (bzw. `len` - 1 falls die Schrittweite negativ ist)
- `end`: `len` (bzw. -1 falls die Schrittweite negativ ist)
- `step`: 1

### Beispiel 29

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares[2:4]
[9, 16]
>>> squares[:3]
[1, 4, 9]
>>> squares[2:]
[9, 16, 25]
>>> squares[::-2]
[1, 9, 25]
>>> squares[:]
[1, 4, 9, 16, 25]
```

Slicing-Ausdrücke können negative Indizes (relativ zum Ende der Liste) und negative Schrittweiten enthalten.

### Beispiel 30

```
>>> squares = [1, 4, 9, 16, 25]
```

```
>>> squares[-2:]  
[16, 25]  
>>> squares[:-2]  
[1, 4, 9]  
>>> squares[2::-1]  
[9, 4, 1]  
>>> squares[:2:-1]  
[25, 16]  
>>> squares[::-1]  
[25, 16, 9, 4, 1]
```

Listen unterstützen weitere Operationen wie zum Beispiel die sogenannte *Konkatenation*, mit der wir zwei Listen mit dem Operator + verbinden können (funktioniert nur, wenn beide Operanden Listen sind):

### Beispiel 31

```
>>> squares = [1, 4, 9, 16, 25]  
>>> squares + [36, 49, 64, 81, 100]  
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]  
>>> list1 = [1, 2, 3, 4]  
>>> list1 + 5  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
TypeError: can only concatenate list (not "int") to list  
>>> list1 + [5]  
[1, 2, 3, 4, 5]
```

Beachten Sie, dass im obigen Beispiel die Listen `squares` und `list1` *nicht* verändert werden (denken Sie daran: Lesen ändert eine Variable

niemals!). Um eine Liste anzupassen, können Sie z.B. den Zuweisungsoperator `+=` anwenden:

### Beispiel 32

```
>>> list1 = [1, 2, 3, 4]
>>> list2 = [5, 6, 7, 8]
>>> list1 += list2
>>> list1
[1, 2, 3, 4, 5, 6, 7, 8]
```

Zudem können Sie einzelne Elemente einer Liste leicht ändern:

### Beispiel 33

```
>>> cubes = [1, 8, 27, 65, 125] # falsche Eingabe
>>> cubes[3] = 64 # falschen Wert ersetzen
>>> cubes
[1, 8, 27, 64, 125]
```

Die Zuordnung zu *Slices* ist ebenfalls möglich, was möglicherweise die Grösse der Liste verändert oder den Inhalt einer Liste ganz löscht:

### Beispiel 34

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # ersetzen mehrerer Werte
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # löschen von Werten
>>> letters[2:5] = []
```

```

>>> letters
['a', 'b', 'f', 'g']
>>> # ersetzen der Liste mit einer leeren Liste
>>> letters[:] = []
>>> letters
[]

```

Objekte vom Datentyp `list` bieten neben den Operationen mit Indizes und eckigen Klammern `[]` noch weitere Bearbeitungsmöglichkeiten an: Sogenannte *Methoden* der Klasse `list`. Wir können uns Methoden wie Funktionen vorstellen – im Gegensatz zu eingebauten Funktionen, die direkt aufgerufen werden können (wie z.B. `print` oder `len`), werden Methoden aber immer auf dem Objekt aufgerufen, das Sie bearbeiten wollen. Hierzu verwenden Sie den sogenannten *Punktoperator* wie folgt:

```
listen_bezeichner.methoden_bezeichner([parameter])
```

**Beispiel 35** Die Klasse `list` bietet zum Beispiel die Methode `append` an, mit der man ein Element an das Ende der Liste hinzufügen kann:

```

>>> lst = [1, 2, 3]
>>> lst.append(4)
>>> lst
[1, 2, 3, 4]

```

Nachfolgend sind einige Methoden der Klasse `list` aufgeführt (`lst` sei dabei eine Variable vom Typ `list`, also z.B. `lst = [1, 2, 3]`):

- `lst.append(x)`:

Fügt das Element `x` am Ende der Liste hinzu.

- `lst.insert(i, x):`

Fügt ein Element an einer bestimmten Position ein. Das erste Argument `i` ist der Index des Elements, vor dem eingefügt werden soll, also fügt z.B. `lst.insert(0, x)` das Element `x` am Anfang der Liste ein.

- `lst.remove(x):`

Entfernt das erste Element aus der Liste, dessen Wert gleich `x` ist (lässt einen Fehler aus, wenn es kein solches Element gibt).

- `lst.pop([i])` Entfernt das Element an der angegebenen Position in der Liste und gibt es zurück. Wenn kein Index `i` angegeben wird, entfernt `lst.pop()` den letzten Eintrag in der Liste und gibt ihn zurück.

- `lst.clear():`

Entfernt alle Elemente aus der Liste .

- `lst.index(x):`

Liefert den Index des ersten Elements, dessen Wert gleich `x` ist (lässt einen Fehler aus, wenn es kein solches Element gibt).

- `lst.count(x):`

Liefert die Häufigkeit von `x` in der Liste.

- `lst.sort():`

Sortiert die Liste (funktioniert nur, wenn die Elemente in der Liste auch tatsächlich geordnet werden können).

**Beispiel 36** Betrachten Sie das folgende Programm, das einige der Methoden auf Listenobjekten demonstriert

```

"""
band.py
"""

# Liste definieren
band = ["Lauener", "Mumenthaler", "Fehlmann"]
print("(a)", band)

# Elemente hinzufügen
band.append("Schmid")
band.insert(2, "von Siebenthal")
print("(b)", band)

# Element entfernen
band.pop(1)
print("(c)", band)

# Element ersetzen
band[1] = "Etter"
band[3] = "Stäuble"
print("(d)", band)

# Elemente auslesen
print("(e)", band[0])
print("(f)", band[-1])

# Element suchen
searchname = "Fehlmann"
location = band.index(searchname)
print("(g)", location)

# Grösse der Liste anzeigen
print("(h)", len(band))

# Sortieren der Liste
band.sort()
print("(i)", band)

# Löschen der Liste
band.clear()
print("(j)", band)

```

Die Ausgabe lautet:

- (a) ['Lauener', 'Mumenthaler', 'Fehlmann']
- (b) ['Lauener', 'Mumenthaler', 'von Siebenthal', 'Fehlmann', 'Schmid']

- (c) `['Lauener', 'von Siebenthal', 'Fehlmann', 'Schmid']`
- (d) `['Lauener', 'Etter', 'Fehlmann', 'Stäuble']`
- (e) `Lauener`
- (f) `Stäuble`
- (g) `2`
- (h) `4`
- (i) `['Etter', 'Fehlmann', 'Lauener', 'Stäuble']`
- (j) `[]`

### 2.2.1 Listen-Abstraktionen

Mit *Listen-Abstraktionen* (engl. *List Comprehension*) können mit wenig Code neue Listen erstellt und mit Daten gefüllt werden. Nehmen wir zum Beispiel an, wir wollen eine Liste mit 1000 Nullen erzeugen:

#### Beispiel 37

```
>>> zeros = [0 for x in range(1000)]
```

Die folgende Listen-Abstraktion erzeugt eine Liste von Quadratzahlen:

#### Beispiel 38

```
>>> squares = [x**2 for x in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Folgende Listen-Abstraktion erstellt eine neue Liste aus einer bestehenden Liste mit verdoppelten Werten:

#### Beispiel 39

```
>>> vec1 = [-4, -2, 0, 2, 4]
>>> vec2 = [x*2 for x in vec1]
>>> vec2
[-8, -4, 0, 4, 8]
```

Wir filtern die Liste, um negative Zahlen auszuschliessen (hierzu verwenden wir die `if`-Klausel, die wir erst im nächsten Kapitel genau betrachten):

### Beispiel 40

```
>>> vec1 = [-4, -2, 0, 2, 4]
>>> vec2 = [x for x in vec1 if x >= 0]
>>> vec2
[0, 2, 4]
```

Wir wenden eine eingebaute Funktion auf allen Elementen der Liste an:

### Beispiel 41

```
>>> vec1 = [-4, -2, 0, 2, 4]
>>> vec2 = [abs(x) for x in vec1]
>>> vec2
[4, 2, 0, 2, 4]
```

# Kapitel 3

## Schleifen und Bedingungen

Python Programme führen – solange nichts anderes definiert ist – alle Anweisungen eines Moduls von oben nach unten aus, bis das Ende der Datei erreicht wird. Mit Hilfe von *Bedingungsanweisungen* (engl. *Conditionals*) und *Schleifen* (engl. *Loops*) können wir die Reihenfolge der Ausführungen aber beeinflussen (siehe auch Abb. 3.1).

Schleifen erlauben uns, gewisse Programmieranweisungen mehrfach ausführen zu lassen (ohne diese mehrfach zu programmieren). Bedingungsanweisungen führen einige Programmieranweisungen nur dann aus, wenn bestimmten Bedingungen erfüllt sind.

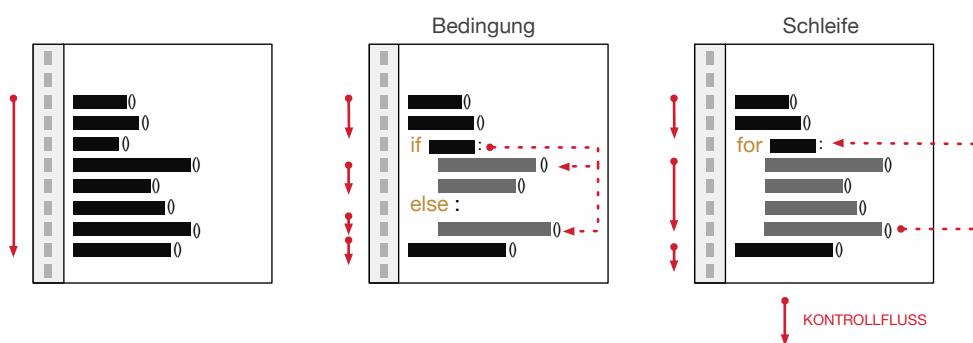


Abbildung 3.1: Schleifen und Bedingungen steuern den Kontrollfluss eines Programmes.

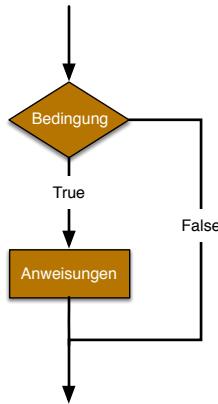


Abbildung 3.2: Die Logik einer `if`-Anweisung.

### 3.1 Die `if`-Anweisung

Die `if`-Anweisung ist ein Beispiel einer Bedingungsanweisungen. Die erste Zeile einer `if`-Anweisung besteht aus dem reservierten Wort `if` gefolgt von einem *Booleschen Ausdruck* und einem Doppelpunkt `:`. Danach folgen beliebig vielen Programmieranweisungen, die mit einem Tabulator (= 4 Leerzeichen) eingerückt sein müssen. Ein Boolescher Ausdruck ist entweder wahr oder falsch (besitzt also den Wert `True` oder `False`).

Wenn der Boolesche Ausdruck wahr ist, so werden die eingerückten Programmieranweisungen ausgeführt und wenn nicht, so werden diese übersprungen und es wird mit der nächsten Anweisung unterhalb der `if`-Anweisung fortgefahrene (siehe Abb. 3.2).

**Beispiel 42** Betrachten Sie folgendes Beispiel einer `if`-Anweisung:

```

if count > 10:
    print("10 überschritten!")

```

Der Boolesche Ausdruck in dieser ***if***-Anweisung ist **count** > 10. Dieser Ausdruck ist entweder wahr oder falsch. Wenn **count** grösser ist als 10, so wird die **print** Anweisung ausgeführt. Andernfalls wird die **print** Anweisung übersprungen und die nächste Anweisung unterhalb der ***if***-Anweisung wird ausgeführt.

Die Anweisung unter der ***if***-Klausel ist eingerückt. Dies bedeutet, dass diese Programmieranweisung zur ***if***-Anweisung gehört (und nur dann ausgeführt wird, wenn die Bedingung wahr ist). Es können beliebig viele Anweisungen zu einer ***if***-Anweisung hinzugefügt werden.

**Beispiel 43** Das folgende Beispielprogramm fällt eine Entscheidung aufgrund des eingegebenen Wertes **hours**.

```
"""
hours_check.py
"""

maximum = 42 # Obergrenze der zulässigen Stunden
payrate = 22.5 # Stundenlohn

hours = int(input("Geleistete Stunden eingeben: "))

if hours > maximum:
    print("Arbeit stoppen")
    hours = maximum # Maximum = 42 Stunden

# Gehalt berechnen und ausgeben
pay = hours * payrate
print("Ihr Gehalt:", pay)
print("Stunden bezahlt:", hours)
```

Wenn der Wert der Variablen **hours** grösser ist, als die im Quellcode definierte Variable **maximum**, wird **Arbeit stoppen** ausgegeben und der Variablen **hours** wird der Wert von **maximum** zugewiesen. Andernfalls werden beide Anweisungen übersprungen. Danach wird das Gehalt berechnet und ausgegeben.

Eine mögliche Ein- und Ausgabe ist z.B.:

*Geleistete Stunden eingeben: 45*

*Arbeit stoppen*

*Ihr Gehalt: 945.0*

*Stunden bezahlt: 42*

### 3.1.1 Boolesche Ausdrücke

Die obigen Beispiele von `if`-Anweisungen basieren auf Booleschen Ausdrücken, die zwei numerische Werte miteinander vergleichen (hierzu haben wir den Operator `>` verwendet). In Python existieren weitere sogenannte *relationale Operatoren*, die den Vergleich zweier Werte/Variablen und die Definition eines Booleschen Ausdrucks ermöglichen:

- `==`: Sind zwei Werte gleich?
- `!=`: Sind zwei Werte ungleich?
- `<`: Ist der erste Wert kleiner als der zweite Wert?
- `>`: Ist der erste Wert grösser als der zweite Wert?
- `<=`: Ist der erste Wert kleiner-gleich dem zweiten Wert?
- `>=`: Ist der erste Wert grösser-gleich dem zweiten Wert?

Diese relationalen Operatoren sind sowohl für Zahlen als auch für Zeichenketten definiert. Die Berechnung eines Booleschen Ausdrucks mit `<` oder `>` auf Zeichenketten basiert auf dem Unicode Zeichensatz. In Unicode wird jedem Zeichen ein numerischer Wert zugewiesen. Glücklicherweise sind die Ziffern sowie die Gross- und Kleinbuchstaben in Unicode aufsteigend sortiert codiert:

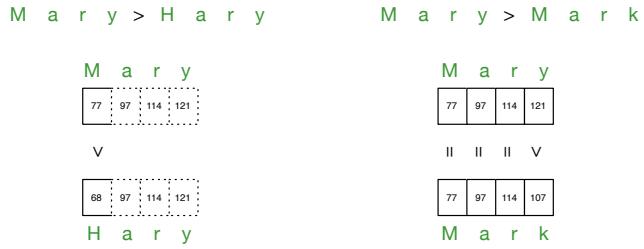


Abbildung 3.3: Zeichenketten werden auf Basis der Unicode Werte einzelner Zeichen von links nach rechts miteinander verglichen.

Zeichen	Unicode Wert
"0" – "9"	48 – 57
"A" – "Z"	65 – 90
"a" – "z"	97 – 122

Bei einem Vergleich von Zeichenketten werden die zugehörigen Codierungen der Zeichen von links nach rechts verglichen bis das erste ungleiche Zeichenpaar gefunden wird oder das Ende der Zeichenkette erreicht wird (in diesen Fall sind die Zeichenketten identisch).

**Beispiel 44** Siehe Abb. 3.3: Die Zeichenkette "**Mary**" ist zum Beispiel grösser als "**Harry**", da der Unicode von "**M**" grösser ist als der von "**H**". Bei den Zeichenketten "**Mary**" und "**Mark**" stimmen die ersten drei Zeichen überein – der Unicode von "**y**" ist dann aber grösser als der Unicode von "**k**" und somit ist die Zeichenkette "**Mary**" grösser als "**Mark**".

Beachten Sie: "**George**" ist zum Beispiel kleiner als "**abraham**", da Grossbuchstaben einen kleineren Unicode Wert als Kleinbuchstaben besitzen. Eine Zeichenkette, die *Präfix* einer anderen Zeichenkette ist, ist immer kleiner als die längere Zeichenkette (z.B. ist "**Haus**" kleiner als "**Hausdach**").

## Beispiel 45

```
>>> "abba" < "zumba"  
True  
>>> "abba" < "aha"  
True  
>>> "George" < "abraham"  
True  
>>> "Haus" < "Hausdach"  
True
```

Das logische `and`, das logische `or` und das logische `not` sind in Python ebenfalls vorhanden (alles reservierte Wörter):

- Der Ausdruck `not a` ist `True`, wenn `a` `False` ist und umgekehrt.
- Der Ausdruck `a and b` ist `True`, wenn `a und b` beide `True` sind und sonst `False`.
- Der Ausdruck `a or b` ist `True`, wenn `a oder b` `oder` beide `True` sind und sonst `False`.

Ein Boolescher Ausdruck kann mit einer *Wahrheitstabelle* beschrieben werden, der alle möglichen `True-False`-Kombinationen für die involvierten Ausdrücke auflistet. Für den Operator `not` gibt es nur zwei mögliche Situationen:

<code>a</code>	<code>not a</code>
<code>False</code>	<code>True</code>
<code>True</code>	<code>False</code>

Beachten Sie, dass der Operator `not` den gespeicherten Wert in `a` nicht verändert (wollen wir den Wert, der in `a` gespeichert ist, verändern, so braucht es eine Zuweisung: `a = not a`).

Für die Operatoren `and` und `or` gibt es vier mögliche Situationen:

a	b	a and b	a or b
<code>False</code>	<code>False</code>	<code>False</code>	<code>False</code>
<code>False</code>	<code>True</code>	<code>False</code>	<code>True</code>
<code>True</code>	<code>False</code>	<code>False</code>	<code>True</code>
<code>True</code>	<code>True</code>	<code>True</code>	<code>True</code>

Der Operator `not` hat die höchste Priorität der drei logischen Operatoren, gefolgt von `and` und von `or`.

**Beispiel 46** Betrachten Sie die folgende `if`-Anweisung:

```
if not complete and count > maximum:  
    print("Alarm")
```

Die Variable `complete` sei hierbei vom eingebauten Datentyp `bool` – ist also entweder `True` oder `False` – genauso ist die Variable `count` entweder grösser als `maximum` oder nicht. Folgende Wahrheitstabelle listet alle möglichen Kombinationen auf:

complete	count > maximum	not complete	not complete and count > maximum
<code>False</code>	<code>False</code>	<code>True</code>	<code>False</code>
<code>False</code>	<code>True</code>	<code>True</code>	<code>True</code>
<code>True</code>	<code>False</code>	<code>False</code>	<code>False</code>
<code>True</code>	<code>True</code>	<code>False</code>	<code>False</code>

Offensichtlich wird diese `if`-Anweisung nur dann `Alarm` ausgeben, wenn `complete` falsch und `count` grösser als `maximum` ist.

Eine weitere Möglichkeit, Boolesche Ausdrücke zu bilden, bietet der `in`-Operator (ebenfalls ein reserviertes Wort). Mit dem `in`-Operator können Sie feststellen, ob ein bestimmtes Element in einer Liste enthalten ist oder nicht. Der Ausdruck

```
val in lst
```

gibt `True` zurück, wenn der Wert `val` in der Liste `lst` enthalten ist und sonst `False`<sup>1</sup>.

### 3.1.2 Die `if-else` Anweisung

Manchmal möchten wir in einem Programm etwas ausführen, wenn eine Bedingung wahr ist und etwas anderes, wenn die Bedingung falsch ist. Hierzu können wir ein `else` zu der `if`-Anweisung hinzufügen.

**Beispiel 47** Betrachten Sie die folgende `if-else` Anweisung: Wenn `count <= maximum` wahr ist, so wird die Variable `count` um 1 erhöht, andernfalls wird `count` der Wert 0 zugewiesen.

```
if count <= maximum:  
    count += 1  
else:  
    count = 0
```

Beachten Sie, dass zur Laufzeit des Programmes nur immer entweder die `if`- oder die `else`-Anweisung ausgeführt wird, aber niemals beide.

**Beispiel 48** Das folgende Beispielprogramm fällt eine Entscheidung aufgrund der Variablen `age`.

```
"""  
age_check.py  
"""  
  
minor = 18  
  
age = int(input("Ihr Alter: "))  
  
if age < minor:  
    print("Jugend ist kein Fehler.")  
else:  
    print("Alter ist kein Verdienst.")  
print("Alter ist eine Zahl.")
```

---

<sup>1</sup>Mit dem Operator `not in` können Sie bestimmen, ob ein Element *nicht* in einer Liste enthalten ist.

Wenn das eingegebene Alter kleiner ist, als die im Quellcode definierte Konstante `minor`, wird `Jugend ist kein Fehler.` ausgegeben. Andernfalls wird diese `print` Anweisung übersprungen und dafür `Alter ist kein Verdienst.` ausgegeben. In beiden Fällen wird danach der Satz `Alter ist eine Zahl.` ausgegeben.

Zwei mögliche Ein- und Ausgaben:

*Ihr Alter: 43*  
*Alter ist kein Verdienst.*  
*Alter ist eine Zahl.*

*Ihr Alter: 15*  
*Jugend ist kein Fehler.*  
*Alter ist eine Zahl.*

### 3.1.3 Verschachtelte `if`-Anweisungen

Die Anweisungen innerhalb einer `if`- oder `else`-Anweisung können wiederum `if`-Anweisungen sein. Solche Konstrukte nennt man *verschachtelte if-Anweisungen*. Verschachtelte `if`-Anweisungen erlauben uns, komplexe Entscheidungen zu treffen, die auf mehreren – verschachtelten – Entscheidungen basieren.

**Beispiel 49** Das Programm `max_of_three.py` verwendet verschachtelte `if`-Anweisungen, um die grösste Zahl aus drei eingegebenen Zahlen zu finden. Eine mögliche Ein und Ausgabe lautet:

*Erste Zahl: -99*  
*Zweite Zahl: 1234.56*

*Dritte Zahl: 17*

*Das Maximum Ihrer Eingaben: 1234.56*

```
"""
max_of_three.py
"""

num1 = float(input("Erste Zahl: "))
num2 = float(input("Zweite Zahl: "))
num3 = float(input("Dritte Zahl: "))

if num1 > num2:
    if num1 > num3:
        maximum = num1
    else:
        maximum = num3
else:
    if num2 > num3:
        maximum = num2
    else:
        maximum = num3

print("Das Maximum Ihrer Eingaben: ", maximum)
```

Betrachten Sie die folgende verschachtelte **if**-Anweisung:

```
if not full:
    if pressure <= maximum:
        print("OK")
    else:
        print("Problem mit dem Druck!")
```

Die Ebene der Einrückung gibt vor, dass die **else**-Anweisung zum zweiten **if** gehört. Wollen wir das obige Beispiel so ändern, dass das **else** zur ersten **if**-Anweisung gehört, kann dies durch eine Anpassung der Einrückung erreicht werden:

```
if not full:
    if pressure <= maximum:
        print("OK")
else:
    print("Problem mit der Füllung!")
```

### 3.1.4 Die `if-elif`-Anweisung

Eine `if-elif`-Anweisung führt dazu, dass der Kontrollfluss einem von mehreren möglichen Pfaden folgt. Das Schlüsselwort `elif` ist eine Kurzform für ein `if` nach einem `else`. Es ermöglicht uns, mehrere Boolesche Ausdrücke ohne Verschachtelungen zu überprüfen.

Wenn die Boolesche Bedingung in der `if`-Klausel falsch ist, wird die Bedingung des nächsten `elif`-Blocks überprüft. Ist die Bedingung hier auch falsch, wird der nächste `elif`-Block überprüft, und so weiter. Wenn alle Booleschen Bedingungen falsch sind, werden die Anweisungen unterhalb vom `else` ausgeführt (der `else` Block ist dabei optional).

**Beispiel 50** Folgendes Code Fragment ist ein Beispiel für eine `if-elif`-Anweisung:

```
if val == "a":  
    a_count += 1  
elif val == "b":  
    b_count += 1  
elif val == "c":  
    c_count += 1  
else:  
    others += 1
```

Wenn in unserem Beispiel der Ausdruck `val == "a"` wahr ist, dann wird `a_count` um eins erhöht (und alles andere übersprungen). Wenn `val == "b"` wahr ist, wird die `if`-Anweisung übersprungen und es wird `b_count` um eins erhöht (analog wird für den Fall `val == "c"` der Kontrollfluss direkt zum zweiten `elif` springen).

Wenn keiner der angegebenen Fälle dem evaluierten Wert entspricht (z.B. wenn `val == "z"`), so springt der Kontrollfluss zum optionalen

*Standardfall* (angegeben mit dem reservierten Wort `else`). Existiert kein `else` Fall, wird in einer solchen Situation *gar keine* Anweisung der gesamten `if-elif`-Anweisung durchgeführt. Es ist meistens eine gute Idee, einen `else` Fall zu programmieren (selbst dann, wenn man eigentlich davon ausgehen darf, dass dieser Fall gar nie eintreten kann).

**Beispiel 51** In folgenden Programm `report.py` wird eine schriftliche Beurteilung auf Basis der erreichten Punkte bestimmt. Die auf eine ganze Zahl gerundete Schulnote `grade` wird zur Überprüfung für die `if-elif`-Anweisung verwendet. Für das Runden verwenden wir die eingebaute Funktion `round(number[, ndigits])`, die eine zu rundernde Zahl `number` und optional eine ganze Zahl `ndigits` entgegen nimmt. Die Funktion rundenet `number` auf `ndigits` Nachkommastellen. Wenn `ndigits` weggelassen wird, wird auf die nächste ganze Zahl gerundet.

```
"""
report.py
"""

max_points = 35

points = int(input("Erreichte Punkte eingeben (0 - 35): "))
grade = round(points / max_points * 5 + 1)

if grade == 6:
    print("Das ist ausgezeichnet.")
elif grade == 5:
    print("Das ist gut.")
elif grade == 4:
    print("Das ist genügend.")
else:
    print("Das ist ungenügend.")
```

Eine mögliche Ein- und Ausgabe des Programmes lautet:

Erreichte Punkte eingeben (0 - 35): 26

Das ist gut.

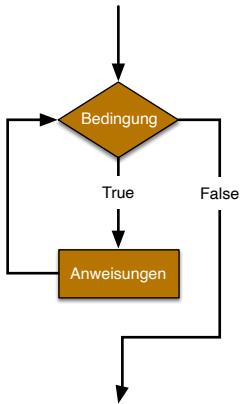


Abbildung 3.4: Die Logik von `while`-Schleifen.

## 3.2 Die `while`-Anweisung

Eine `while`-Anweisung ist eine Schleife, die eine Boolesche Bedingung genau gleich evaluiert wie eine `if`-Anweisung. Wenn die Bedingung wahr ist, so werden die zugehörige(n) Programmieranweisung(en) ausgeführt.

Anders als bei einer `if`-Anweisung wird der Kontrollfluss danach aber nicht zur nächsten Anweisung *unterhalb* der `while`-Anweisung springen, sondern die Boolesche Bedingung wird *nochmals* evaluiert. Falls diese immer noch wahr ist, wird die zugehörige Programmieranweisung nochmals ausgeführt. Dies wird solange fortgesetzt, bis die Bedingung irgendwann falsch wird. Erst dann wird mit der nächsten Anweisung unterhalb der `while`-Anweisung fortgefahrene (siehe Abb. 3.4).

**Beispiel 52** Die folgende Schleife gibt zum Beispiel die Werte von 1 bis 3 aus:

```

count = 1
while count <= 3:
    print(count)
    count += 1

```

Beachten Sie, dass im obigen Beispiel zwei Anweisungen eingerückt sind. Der ganze eingerückte Block wird bei jeder Iteration durch die `while`-Schleife wiederholt.

**Beispiel 53** Das Programm `average` liest eine Reihe von Zahlen vom Benutzer ein, summiert diese und berechnet den Durchschnitt der eingegebenen Zahlen. Hierzu verwenden wir die Variable `total`, um die Benutzereingaben aufzusummieren und die Variable `count`, um die Anzahl Benutzereingaben zu zählen. In der Variablen `value` speichern wir die jeweils aktuelle Benutzereingabe.

```
"""
average.py
"""

total = 0.0 # Summe der Eingaben
count = 0 # Anzahl Eingaben

value = int(input("Ganze, positive Zahl eingeben (-1 zum Beenden): "))

while value != -1:
    count += 1
    total += value
    print("Aktuelle Summe:", total)
    value = int(input("Ganze, positive Zahl eingeben (-1 zum Beenden): "))

if count == 0:
    print("Keine Zahlen eingegeben!")
else:
    mean = total / count
    print("Durchschnitt der Eingaben:", round(mean, 2))
```

Da wir zu Beginn nicht wissen können, wie viele Zahlen der Benutzer eingeben möchte, brauchen wir eine Möglichkeit, dem Programm mitzuteilen, dass der Benutzer seine Eingabe beenden möchte. Im Programm `average` definieren wir hierzu die Zahl `-1` als sogenannten Wächterwert (engl. sentinel value).

Ein Wächterwert muss immer ausserhalb der normalen Eingabemöglichkeiten liegen. Sobald der Benutzer diesen Wächterwert eingibt, soll die Eingabe beendet werden. Umgekehrt soll der Benutzer weitere Eingaben machen können, so lange der zuletzt eingegebene Wert `value` ungleich `-1` ist. Diese Idee programmieren wir mit Hilfe einer `while`-Schleife.

Innerhalb der `while`-Schleife wird zunächst die Variable `count` um eins erhöht, dann wird die letzte Eingabe `value` zu `sum` addiert (und die aktuelle Summe wird ausgegeben) und zuletzt wird eine neue Eingabe vom Benutzer entgegengenommen und der Variablen `value` zugewiesen.

Beachten Sie, dass der Benutzer direkt nach dem Start des Programmes `-1` eingeben könnte. Dies würde in der Anweisung

```
mean = total / count
```

zu einer Division durch `0` führen, die wir aber mit einer `if`-Anweisung abfangen. Bei mindestens einer gültigen Eingabe wird der Durchschnitt der Eingaben gerundet ausgegeben. Eine mögliche Ein- und Ausgabe des Programmes lautet:

Ganze, positive Zahl eingeben (-1 zum Beenden): 4

Aktuelle Summe: 4.0

Ganze, positive Zahl eingeben (-1 zum Beenden): 7

Aktuelle Summe: 11.0

Ganze, positive Zahl eingeben (-1 zum Beenden): 9

*Aktuelle Summe: 20.0*

*Ganze, positive Zahl eingeben (-1 zum Beenden): -1*

*Durchschnitt Ihrer Eingaben: 6.67*

Schleifen und Bedingungen können auch dazu verwendet werden, um Programme robuster zu machen. Ein robustes Programm fängt potentielle Probleme so elegant wie möglich ab. Ungültige Eingaben oder Divisionen durch Null sind typische Probleme, die man mit `if`- oder `while`-Anweisungen abfangen kann.

**Beispiel 54** Das folgende Programm `set.py` demonstriert, wie eine `while`-Schleife dazu verwendet werden kann, den Benutzer zu zwingen eine bestimmte Eingabe zu machen. Der Kontrollfluss bleibt so lange in der `while`-Schleife stecken (jedes Mal mit der Aufforderung eine korrekte Eingabe zu machen), bis der Benutzer einen gültigen Wert eingibt.

```
"""
set.py
"""

values = [4, 2, 5, 1, 8]
print("Aktuelle Werte:", values)

new_value = int(input("Neuen Wert eingeben: "))

while new_value in values:
    print("Dieser Wert ist schon vorhanden.")
    print("Aktuelle Werte:", values)
    new_value = int(input("Neuen Wert eingeben: "))

values.append(new_value)
print("Aktuelle Werte:", values)
```

Mögliche Ein- und Ausgabe des Programmes:

*Aktuelle Werte: [4, 2, 5, 1, 8]*

*Neuen Wert eingeben: 4*

*Dieser Wert ist schon vorhanden.*

*Aktuelle Werte: [4, 2, 5, 1, 8]*

*Neuen Wert eingeben: 7*

*Aktuelle Werte: [4, 2, 5, 1, 8, 7]*

### 3.2.1 Endlosschleifen

Es liegt in unserer Verantwortung, dass die Boolesche Bedingung einer Schleife irgendwann falsch wird. Wenn diese nie falsch wird, so werden die Anweisungen der `while`-Schleife für immer ausgeführt (bzw. bis das ganze Programm abgebrochen wird). Eine solche Situation nennt man *Endlosschleife* und entspricht i.d.R. einem Programmierfehler.

**Beispiel 55 Ein Beispiel einer Endlosschleife:**

```
count = 1
while count <= 1:
    print(count)
    count -= 1
```

Wenn Sie diese oder ähnliche Schleifen programmieren und ausführen, sollten Sie wissen, wie man laufende Programme terminieren kann<sup>2</sup>.

### 3.2.2 Verschachtelte Schleifen

Wie `if`-Anweisungen können auch `while`-Schleifen verschachtelt werden. Dabei gilt: Bei jedem Durchlauf durch die äussere Schleife wird die innere Schleife *komplett* abgearbeitet.

---

<sup>2</sup>Suchen Sie z.B. nach `terminate application in pycharm`.

Wie oft wird z.B. in der folgenden verschachtelten Schleife die Zeichenkette "Hello Again" ausgegeben?

```
count1 = 1
while count1 <= 5:
    count2 = 1
    while count2 < 10:
        print("Hello Again")
        count2 += 1
    count1 += 1
```

Die `print` Anweisung ist innerhalb der inneren Schleife zu finden. Die äussere Schleife wird genau 5 mal ausgeführt (`count1` wächst in Einerschritten von 1 bis 5). Die innere Schleife wird 9 mal ausgeführt (`count2` wächst in Einerschritten von 1 bis 9). Bei jedem Durchlauf der äusseren Schleife wird die innere Schleife komplett abgearbeitet: Deshalb wird "Hello Again"  $5 \times 9 = 45$  mal ausgegeben.

**Beispiel 56** Das folgende Programm `gcd.py` berechnet den grössten gemeinsamen Teiler (ggT) zweier Zahlen:

```
"""
gcd.py
"""

another = "y"

while another == "y" or another == "Y":
    value1 = int(input("Erste Zahl eingeben: "))
    value2 = int(input("zweite Zahl eingeben: "))

    # Euklids Verfahren zur Berechnung des ggT
    while value1 != value2:
        if value1 > value2:
            value1 -= value2
        else:
            value2 -= value1

    gcd = value1
    print("ggT Ihrer Eingaben:", gcd)

    another = input("Weitere Berechnung? (y/n): ")
```

Beachten Sie, dass am Ende der äusseren `while`-Schleife dem Benutzer die Möglichkeit gegeben wird, zu entscheiden, ob er eine weitere Berechnung durchführen möchte oder nicht. Die äussere `while`-Schleife kontrolliert also, wie viele Berechnung durchgeführt werden.

Im Block der äusseren Schleife werden zunächst zwei Zahlen vom Benutzer eingelesen. Mit diesen Eingaben wird danach Euklids Verfahren zur Bestimmung des ggT durchgeführt. Dies erfordert den Einsatz einer inneren `while`-Schleife. In dieser Schleife wird immer die kleinere der beiden Zahlen von der grösseren subtrahiert, bis schliesslich beide gleich gross sind – dies entspricht dann tatsächlich dem ggT der beiden Zahlen (weshalb das so ist, wird hier nicht weiter diskutiert).

Ein mögliche Ein- und Ausgabe des Programmes wäre:

*Erste Zahl eingeben: 36*

*Zweite Zahl eingeben: 24*

*ggT Ihrer Eingaben: 12*

*Weitere Berechnung? (y/n): y*

*Erste Zahl eingeben: 15*

*Zweite Zahl eingeben: 3*

*ggT Ihrer Eingaben: 3*

*Weitere Berechnung? (y/n): n*

### 3.3 Die `for`-Anweisung

Eine `for`-Anweisung iteriert über die Elemente einer beliebigen Sequenz (z.B. einer Liste), in der Reihenfolge, in der sie in der Sequenz

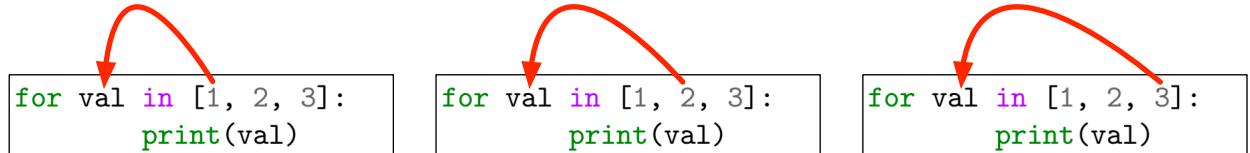


Abbildung 3.5: In jeder Iteration durch die `for`-Schleife zeigt die Laufvariable `val` auf ein anderes Element der Liste.

erscheinen. Die allgemeine Syntax, um zum Beispiel alle Elemente einer Liste `values` auszugeben, lautet:

```

values = [1, 2, 3]
for val in values:
    print(val)

```

Der Variablen `val` – auch *Laufvariable* genannt – wird bei jeder Iteration der nächste Wert aus der Liste `values` zugewiesen (siehe Abb. 3.5).

**Beispiel 57** Die Ausgabe der folgenden `for`-Schleife lautet:

```

cat 3
window 6
defenestrat 12

```

```

words = ["cat", "window", "defenestrat"]
for w in words:
    print(w, (len(w)))

```

Beachten Sie, dass man einer `for`-Schleife beliebig viele Anweisungen hinzufügen kann. Der ganze eingerückte Block wird bei jeder Iteration durch die `for`-Schleife wiederholt.

Die Funktion `range` kann das Schreiben einer `for`-Anweisung vereinfachen:

```

for val in range(1, 4):
    print(val)

```

Um über die Indizes einer Liste zu iterieren, können Sie die Funktionen `range` und `len` wie folgt kombinieren:

```
words = ["cat", "window", "defenestrator"]
for i in range(len(words)):
    print(i, words[i])
```

Die Ausgabe lautet:

```
0 cat
1 window
2 defenestrator
```

In den meisten solchen Fällen ist es jedoch praktischer, die Funktion `enumerate()` zu verwenden. Die Funktion `enumerate()` enthält zusätzlich einen automatischen Zähler mit beliebigem Startwert. Das folgende Code-Fragment

```
words = ["cat", "window", "defenestrator"]
for i, entry in enumerate(words, 1):
    print(i, entry)
```

führt bspw. zur Ausgabe

```
1 cat
2 window
3 defenestrator
```

Auch `for`-Schleifen kann man verschachteln – das Prinzip ist dabei das Gleiche wie bei verschachtelten `while`-Schleifen: Bei jedem Durchlauf durch die äussere Schleife wird die innere Schleife *komplett* abgearbeitet.

Das folgende Code-Fragment

```

for i in range(1, 4):
    for j in range(1, 5):
        print(i * j, end=" ")
    print()

```

führt bspw. zur Ausgabe

```

1 2 3 4
2 4 6 8
3 6 9 12

```

Es gibt Situationen, in denen man den Schleifenablauf beeinflussen möchte.

- Die **break**-Anweisung beendet die aktuelle Schleife vorzeitig und geht direkt in die Zeile nach der Schleife.
- Mit der **continue**-Anweisung wird nur die aktuelle Iteration ausgesetzt, die Schleife wird nicht abgebrochen, sondern mit der nächsten Iteration fortgesetzt.

**Beispiel 58** Mögliche Ein- und Ausgabe des folgenden Programmes:

```

Eintrag eingeben ('Q' zum Beenden): Banana
Eintrag eingeben ('Q' zum Beenden): Apple
Eintrag eingeben ('Q' zum Beenden): Mango
Eintrag eingeben ('Q' zum Beenden): Q
['Banana', 'Apple', 'Mango']

```

```

entries = []
while True:
    inp = input("Eintrag eingeben ('Q' zum Beenden): ")
    if inp == "Q":
        break
    entries.append(inp)
print(entries)

```

**Beispiel 59** Die Ausgabe des folgenden Programmes lautet:

1 3 5 7 9

```
for x in range(10):
    if x % 2 == 0:
        continue
    print(x, end=" ")
```

# Kapitel 4

## Standardmodule Verwenden

Python wird mit einer Bibliothek von Standardmodulen geliefert. Diese Standardbibliothek enthält zahlreiche Module mit Funktionen, welche vielfältige Aufgaben lösen können. Wir besprechen in diesem Kapitel exemplarisch einige Standardmodule – schauen Sie sich bei Bedarf die Dokumentationen anderer Module an:

<https://docs.python.org/3.11/py-modindex.html#cap-m>

Einige der Module der Standardbibliothek sind im Interpreter integriert. Dies ermöglicht den direkten Zugriff auf Funktionen, die zwar nicht zum Kern der Sprache Python gehören, aber dennoch eingebaut sind (dies sind die eingebauten Funktionen, die wir schon oft benutzt haben, z.B. `print`, `round` oder `len`).

I.d.R. müssen wir aber die Standardmodule, die wir verwenden wollen, in unser eigenes Modul importieren. Dies geschieht mit der `import` Anweisung. Folgende Anweisung importiert zum Beispiel das Modul `math` aus der Standardbibliothek:

```
import math
```

Um die Funktionen zu nutzen, die sich im importierten Modul befinden, benötigen wir den Punktoperator und folgende Syntax:

```
modul_bezeichner.funktions_bezeichner([parameter])
```

Um also bspw. die Funktion `sqrt` aus dem importierten Modul `math` aufzurufen (um zum Beispiel die Wurzel aus 2 zu berechnen), programmieren wir

```
result = math.sqrt(2)
```

Alternativ können wir mit `from` auch einzelne Funktionen aus den Modulen importieren. Anschliessend kann diese Funktion direkt aufgerufen werden (ohne Modulbezeichner und Punktoperator). Bspw:

```
from math import sqrt
result = sqrt(2)
```

## 4.1 Das Modul random

In Programmen müssen relativ häufig zufällige Entscheidungen getroffen werden. Das Modul `random` bietet Werkzeuge zur Erzeugung von Zufallszahlen und zur Durchführung von Zufallsauswahlen. Die folgende Liste fasst einige wichtige Funktionen des Moduls `random` zusammen:

- `random.randint(a, b):`  
Liefert eine zufällige ganze Zahl `x`, so dass `a <= x <= b`.
- `random.randrange(a, b[, step]):`

Liefert eine zufällige ganze Zahl  $x$ , so dass  $a \leq x < b$  und  $(x+a) \% \text{step} = 0$ . Der Standardwert für  $\text{step}$  ist 1.

- `random.random()`:

Liefert eine zufällige Gleitkommazahl aus dem Intervall  $[0.0, 1.0[$ .

- `random.uniform(a, b)`:

Liefert eine zufällige Gleitkommazahl aus dem Intervall  $[a, b[$ .

- `random.gauss(mu, sigma)`:

Liefert eine normalverteilte Zufallszahl mit Mittelwert `mu` und Standardabweichung `sigma`.

- `random.shuffle(x)`:

Mischt die Elemente einer Liste `x`.

- `random.choice(x)`:

Liefert ein zufälliges Element aus der Liste `x`.

- `random.sample(x, k)`:

Liefert eine Liste mit `k` Elementen, die aus der Liste `x` ausgewählt wurden (ohne Zurücklegen).

Einige der Funktionen aus dem Modul `random` operieren auf Listen (oder geben eine Liste als Resultat zurück).

## Beispiel 60

```
>>> import random
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
>>> random.shuffle(squares)
```

```

>>> squares
[4, 25, 9, 1, 16]
>>> names = ["Maxime", "Emilie", "Noelle", "Eliane"]
>>> random.choice(names)
'Maxime'

```

Hinweis: Zufallszahlen, die von Funktionen des Moduls `random` erzeugt werden, sind eigentlich *Pseudozufallszahlen*. Das heisst, die Zahlen werden auf Basis eines *Startwertes* (engl. *Seed*) berechnet. Standardmässig wird die Systemzeit für den Startwert verwendet. Man kann aber die Funktion `random.seed(x)` verwenden, um einen eigenen Startwert `x` anzugeben. Dies ist insbesondere hilfreich, wenn Sie eine Simulation, die auf Zufallszahlen basiert, später nochmals mit den genau gleichen Zufallszahlen reproduzieren wollen.

### Beispiel 61

```

>>> import random
>>> numbers = [1, 2, 3, 4, 5, 6]
>>> random.seed(10)
>>> random.sample(numbers, 3)
[5, 1, 4]
>>> random.seed(10)
>>> random.sample(numbers, 3)
[5, 1, 4]

```

**Beispiel 62** Das Programm `guessing_game` demonstriert den Einsatz des Moduls `random`. Die Idee des Programmes ist, dass der Computer eine Zufallszahl zwischen 1 und 100 generiert und der Variablen `computer_pick` zuweist. Danach wird die geratene Zahl des Benutzers eingelesen und der Variablen `user_guess` zugewiesen. Wenn die getippte Zahl des Benutzers der zuvor generierten Zufallszahl entspricht,

wird ein entsprechender Text ausgegeben. Wenn nicht, gibt das Programm dem Benutzer einen Tipp und liest einen neuen Rateversuch ein. Mögliche Ein- und Ausgabe:

*Ich denke an eine Zahl zwischen 1 und 100*

*Rate: 50*

*Falsch!*

*Die gesuchte Zahl ist grösser!*

*Rate: 75*

*Falsch!*

*Die gesuchte Zahl ist kleiner!*

*Rate: 62*

*Richtig geraten!*

*Anzahl Rateversuche: 3*

```
"""
guessing_game.py
"""

import random

bound = 100
computer_pick = random.randint(1, bound)
print("Ich denke an eine Zahl zwischen 1 und", bound)
user_guess = int(input("Rate: "))
guess_count = 1

while user_guess != computer_pick:
    print("Falsch geraten.")
    if user_guess > computer_pick:
        print("Die gesuchte Zahl ist kleiner.")
    else:
        print("Die gesuchte Zahl ist grösser.")
    user_guess = int(input("Rate: "))
    guess_count += 1

print("Richtig geraten!")
print("Anzahl Rateversuche:", guess_count)
```

## 4.2 Das Modul math

Das Modul `math` bietet eine Vielzahl mathematischer Funktionen sowie einige nützliche Konstanten an<sup>1</sup>. Einige wichtige Funktionen des Moduls `math` sind:

- `math.ceil(x)`:  
Liefert die kleinste ganze Zahl grösser oder gleich  $x$ .
- `math.floor(x)`:  
Liefert die grösste ganze Zahl kleiner oder gleich  $x$ .
- `math.comb(n, k)`:  
Liefert die Anzahl der Möglichkeiten,  $k$  Elemente aus  $n$  Elementen auszuwählen (ohne Zurücklegen, ohne Beachtung der Reihenfolge).
- `math.perm(n, k)`:  
Liefert die Anzahl der Möglichkeiten,  $k$  Elemente aus  $n$  Elementen auszuwählen (ohne Zurücklegen, mit Beachtung der Reihenfolge).
- `math.fabs(x)`:  
Liefert den Absolutwert von  $x$ .
- `math.exp(x)`:  
Gibt  $e^x$  zurück, wobei  $e$  der Euler'schen Zahl entspricht.
- `math.log(x[, base])` Mit einem Argument wird der natürliche Logarithmus von  $x$  (zur Basis  $e$ ) zurückgegeben. Mit zwei Argumenten wird der Logarithmus von  $x$  zur angegebenen Basis  $base$  zurückgegeben.

---

<sup>1</sup>Diese Funktionen können nicht für komplexe Zahlen verwendet werden; verwenden Sie die gleichnamigen Funktionen aus dem Modul `cmath`, wenn Sie Unterstützung für komplexe Zahlen benötigen.

- `math.pow(x, y)`: Gibt  $x^y$  zurück (im Gegensatz zum eingebauten `**`-Operator konvertiert `math.pow()` beide Argumente `x` und `y` in Gleitkommazahlen. Verwenden Sie den Operator `**`, um genaue ganzzahlige Potenzen zu berechnen).
- `math.sqrt(x)`: Gibt die zweite Wurzel von `x` zurück.
- `math.cos(x)`: Liefert den Kosinus eines Winkels `x` (analog dazu: `math.sin(x)`, `math.tan(x)`).
- `math.acos(x)`: Liefert den Winkel zurück, dessen Cosinus `x` ist (analog dazu: `math.asin(x)`, `math.atan(x)`).

Neben mathematischen Funktionen speichert das Modul `math` auch wichtige mathematische Konstanten:

- `math.pi`: Die mathematische Konstante  $\pi = 3.141592\dots$
- `math.e`: Die mathematische Konstante  $e = 2.718281\dots$
- `math.inf`: Eine Gleitkomma-Unendlichkeit (für negative Unendlichkeit verwenden Sie `-math.inf`).

Die Werte, welche die Funktionen des Moduls `math` zurückgeben, können direkt in arithmetischen Ausdrücken weiterverwendet werden.

**Beispiel 63** Die folgende Programmieranweisung

```
math.sqrt(1 - math.pow(math.sin(alpha), 2))
```

entspricht zum Beispiel der Formel

$$\sqrt{1 - (\sin \alpha)^2}$$

**Beispiel 64** Nachfolgend ist ein interaktives Programm gezeigt, das die Lösungen einer quadratischen Gleichung der Form

$$ax^2 + bx + c$$

findet. Das Programm verwendet hierzu die Lösungsformel

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} .$$

```
"""
roots.py
"""

import math

a = int(input("Koeffizient a eingeben: "))
b = int(input("Koeffizient b eingeben: "))
c = int(input("Koeffizient c eingeben: "))

discriminant = math.pow(b, 2) - 4 * a * c
if discriminant < 0:
    print("Keine reelle Lösung")
else:
    x1 = (-b + math.sqrt(discriminant)) / (2 * a)
    x2 = (-b - math.sqrt(discriminant)) / (2 * a)
    print("x1 =", round(x1, 2), "; x2 =", round(x2, 2))
```

Eine mögliche Ein- und Ausgabe des Programmes wäre bspw:

```
Koeffizient a eingeben: 5
Koeffizient b eingeben: 3
Koeffizient c eingeben: -4
x1 = 0.64 ; x2 = -1.24
```

## 4.3 Das Modul statistics

Das Modul `statistics` bietet Funktionen zur Berechnung von statistischen Masszahlen auf Daten an.

- `statistics.mean(data)` und `statistics.geometric_mean(data)`: Ermittelt den arithmetischen Mittelwert bzw. das geometrische Mittel der Daten.
- `statistics.median(data)`: Ermittelt den Median der Daten (ungerade Anzahl Datenpunkte: mittlerer Wert; gerade Anzahl Datenpunkte: Mittelwert der beiden mittleren Werte). Wenn man auf jeden Fall ein *bestehendes* Element des Datensatzes ermitteln möchte, kann man die Funktionen `median_low` oder `median_high` verwenden. Diese Funktionen geben den kleineren bzw. grösseren der beiden mittleren Werte zurück.
- `statistics.mode(data)`: Ermittelt den einzelnen, häufigsten Datenpunkt aus diskreten oder nominalen Daten. Wenn es mehrere Modi mit der gleichen Häufigkeit gibt, wird der erste zurückgegeben, der in den Daten gefunden wurde. Mit der Funktion `statistics.multimode(data)` erhalten Sie eine Liste mit allen Modi der Daten.
- `statistics.pvariance(data)` und `statistics.pstdev(data)`: Ermittelt die Populationsvarianz bzw. die Populationsstandardabweichung (die Quadratwurzel der Populationsvarianz) der Daten.
- `statistics.quantiles(data, n=x)`: Teilt die Daten mit gleicher Wahrscheinlichkeit in `x` Intervalle

und liefert eine Liste von  $x - 1$  Schnittpunkten, die die Intervalle trennen. Setzen wir z.B.  $n=100$  erhalten wir *Perzentile*: Die 99 Schnittpunkte, welche die Daten in 100 gleich grosse Mengen unterteilen. Beachten Sie, dass Sie bei der Angabe des Parameters  $n$  den Namen des Parameters explizit mit angeben müssen:

```
statistics.quantiles(data, n=4)
```

Das kennen wir bereits von der eingebauten Funktion `print` (mit den speziellen Argumenten `end=` und `sep=`). Wir kommen später im Skript auf dieses Phänomen zurück.

Hinweis: Mit den Funktionen `pvariance` und `pstdev` behandeln wir unsere Daten so, als ob es sich um die *gesamte* Population handelt. Sind die Daten aber nur eine *Stichprobe* aus der gesamten Population sind die Funktionen `variance()` und `stdev` in der Regel eine bessere Wahl. Hierbei werden die Daten so behandelt, als wären sie eine Stichprobe und nicht die gesamte Population (die Varianz und Standardabweichung werden aus der Stichprobe geschätzt).

**Beispiel 65** In folgendem Programm werden einige der Funktionen des Moduls `statistics` demonstriert.

```
"""
statistics_demo.py
"""

import statistics

data = [5, 3, 5, 9, 1, 2, 14, 4]

print("Daten (unsortiert)", data)
print("Mittelwert der Daten:", statistics.mean(data))
print("Standardabweichung der Daten:", statistics.stdev(data))
print("Median der Daten:", statistics.median(data))
print("Unterer Median der Daten:", statistics.median_low(data))
print("Modus der Daten:", statistics.mode(data))
print("Quartile der Daten:", statistics.quantiles(data, n=4))
```

*Die Ausgabe des Programmes:*

*Daten (unsortiert) [5, 3, 5, 9, 1, 2, 14, 4]  
Mittelwert der Daten: 5.375  
Standardabweichung der Daten: 4.240535680446853  
Median der Daten: 4.5  
Unterer Median der Daten: 4  
Modus der Daten: 5  
Quartile der Daten: [2.25, 4.5, 8.0]*

Hinweis: Die Funktionen `mode` und `multimode` können auch auf nicht-numerischen Daten (z.B. Text) angewendet werden.

### Beispiel 66

```
>>> data = "aabbbbccddddeeffffgg"  
>>> statistics.mode(data)  
'b'  
>>> statistics.multimode(data)  
['b', 'd', 'f']
```

## 4.4 Dateien Lesen und Schreiben

Für die Bearbeitung von Dateien existieren zahlreiche Standardmodule in der Standardbibliothek von Python. Für das einfache Lesen und Schreiben von Dateien ist aber in Python tatsächlich *kein* zusätzliches Modul nötig. Die eingebaute Funktion `open` kann – wie auch `print` oder `round` – direkt verwendet werden. Die Funktion `open` gibt ein Dateiobjekt zurück und wird mit zwei Parametern aufgerufen:

```
file = open(filename, mode)
```

Das erste Argument `filename` ist eine Zeichenkette, die den Dateinamen enthält. Das zweite Argument `mode` ist eine weitere Zeichenkette, welche den Modus beschreibt, mit dem die Datei verwendet werden soll. Einige erlaubte Modi sind:

- `"r"` – Lesemodus: Datei kann *nur* gelesen werden
- `"w"` – Schreibmodus: Datei wird *nur* geschrieben (eine vorhandene Datei mit dem gleichen Namen wird dabei *überschrieben*)
- `"a"` – Anhängmodus: Neue Daten werden an das Ende einer bestehenden Datei *angehängt*
- `"r+"` - Lesen und Schreiben sind erlaubt

Tipp: Öffnen Sie Dateien in einer `with`-Umgebung, so dass Dateien *immer* ordnungsgemäss geschlossen werden:

```
with open('data.txt') as file:
```

#### 4.4.1 Vom Umgang mit Zeichenketten

Wenn wir Inhalte aus Dateien lesen, erhalten wir immer Objekte vom Typ `str` zurück. Umgekehrt können wir ausschliesslich Zeichenketten in Dateien schreiben. Bevor wir Dateien lesen und schreiben, lohnt es sich deshalb, die Datenstruktur `str` etwas genauer zu betrachten.

Auf Objekten vom Datentyp `str` können Sie – wiederum mit Hilfe des Punktoperators – verschiedene Methoden aufrufen:

```
zeichenketten_bezeichner.methoden_bezeichner()
```

In folgender Liste sind einige der Methoden zusammengefasst (`st` sei dabei eine Variable vom Typ `str`; bspw. `st = "Test"`):

- `st.find(s)`:

Gibt die Position der gesuchten Zeichenkette `s` in `st` aus (-1, falls sich `s` nicht in der Zeichenkette `st` befinden sollte).

- `st.endswith(suffix)` und `st.startswith(prefix)`:

Gibt den Wahrheitswert `True` zurück, wenn die Zeichenkette mit dem angegebenen Suffix endet bzw. mit dem Präfix startet, ansonsten `False`.

- `st.count(sub)`:

Liefert die Anzahl der nicht überlappenden Vorkommen des Teilstrings `sub` in der Zeichenkette `st`.

- `st.lower()` und `st.upper()`:

Liefert eine Kopie der Zeichenkette `st` zurück, in der alle Grossbuchstaben in Kleinbuchstaben bzw. alle Kleinbuchstaben in Grossbuchstaben umgewandelt sind.

- `st.replace(old, new)`:

Gibt eine Kopie der Zeichenkette `st` zurück, bei der alle Vorkommen von `old` durch `new` ersetzt werden.

- `st.strip([chars])`:

Gibt eine Kopie der Zeichenkette `st` mit entfernten Zeichen `chars`

zurück. Wird der Parameter weggelassen, werden führende und abschliessende Leerzeichen entfernt.

- `st.split(sep)`:

Gibt eine Liste der Wörter aus der Zeichenkette `st` zurück, wobei `sep` als Begrenzungszeichen verwendet wird.

Ist eine Variable vom Typ `str` definiert, so kann weder dessen Grösse noch können einzelne Zeichen verändert werden. Man sagt: Zeichenketten sind unveränderliche Objekte (engl. *immutable*). Es gibt aber Methoden, die eine Kopie der Zeichenkette zurückgeben, welche das Resultat von Veränderungen an der ursprünglichen Zeichenkette sind.

### Beispiel 67

```
>>> name1 = "kaspar"
>>> name2 = name1.upper()
>>> name1 # Zeichenketten sind unveränderlich
'kaspar'
>>> name2
'KASPAR'
```

Zeichenketten dürfen beliebig lang sein. Wird eine Zeichenkette im Quellcode aber mit einem Zeilenumbruch getrennt, führt dies zu einem Syntaxfehler. Mit anderen Worten: Zeichenketten dürfen das Spezialzeichen “Zeilenumbruch” nicht enthalten.

Zeichenketten können mit dem Konkatenationsoperator `+` (den wir schon auf Listen angewendet haben) über mehrere Zeilen ”verklebt” werden. Die Ausgabe von:

```
print("py" +
      "thon")
```

ist bspw. `python`.

Das Zeilenumbruchzeichen ist nicht das einzige Spezialzeichen, das nicht in einer Zeichenkette enthalten sein darf. Beachten Sie, zum Beispiel folgende Anweisung:

```
print("Er sagte: \"Hallo.\"")
```

Die zweiten Anführungszeichen werden als Ende der Zeichenkette interpretiert (was zu einem Syntaxfehler führt). Um dieses (und ähnliche) Probleme zu bewältigen, bietet Python eine Reihe sogenannter *Escape Sequenzen* an, mit denen Spezialzeichen in Zeichenketten integriert werden können. *Escape Sequenzen* starten immer mit dem Zeichen `\`.

Beispiele von solchen Sequenzen:

- `"\t"`: Tabulator
- `"\n"`: Zeilenumbruch
- `"\'"`: Einfache Anführungszeichen
- `"\""`: Doppelte Anführungszeichen
- `"\\\""`: Backslash Zeichen

Ähnlich wie die Indexierung auf Listen, funktioniert das *Indexieren auf Zeichenketten*. Der Index des ersten Zeichens in einer Zeichenkette ist 0, das zweite Zeichen hat Index 1, etc. Indizes können auch negative Zahlen sein, um von rechts zu zählen (der Index -1 kennzeichnet das letzte Zeichen, -2 kennzeichnet das vorletzte Zeichen, etc.)

## Beispiel 68

```
>>> word = "Python"
```

```

>>> word[0] # Zeichen an Position 0
'P'
>>> word[5] # Zeichen an Position 5
'n'
>>> word[-1] # letztes Zeichen
'n'
>>> word[-2] # zweitletztes Zeichen
'o'

```

Neben der Indizierung wird auch das *Slicing* unterstützt. Während die Indizierung verwendet wird, um *einzelne* Zeichen zu erhalten, ermöglicht das *Slicing* das Erhalten von Teilzeichenketten:

```

>>> word[0:2] # Zeichen von 0 (inklusive) bis 2 (exklusive)
'Py'
>>> word[2:5] # Zeichen von 2 (inklusive) bis 5 (exklusive)
'tho'

```

Diese Indizes haben nützliche Standardwerte: Ein weggelassener erster Index ist auf 0 voreingestellt, ein weggelassener zweiter Index auf die Grösse der zu schneidenden Zeichenkette.

```

>>> word[:2] # Zeichen vom Anfang bis 2 (exklusive)
'Py'
>>> word[4:] # Zeichen von 4 (inklusive) bis zum Ende
'on'
>>> word[-2:] # von der zweitletzten Position bis zum Ende
'on'

```

#### 4.4.2 Dateien Lesen

Ein zurückgegebenes Dateiobjekt `file_object` der Funktion `open` bietet Methoden an, mit denen Daten aus `file_object` gelesen oder in `file_object` geschrieben werden können<sup>2</sup>. Um zum Beispiel den Inhalt einer Datei zu lesen, rufen wir die Methode `read` auf einem vorher geöffneten Dateiobjekt `file_object` auf:

```
file_object.read()
```

Die Methode `read` liest die Datei ein und gibt den *gesamten* Inhalt der Datei als einzige (möglicherweise sehr lange) Zeichenkette zurück.

Um Zeilen einzeln aus einer Datei `file_object` zu lesen (um die so gelesenen Zeichenketten einzeln zu bearbeiten), können wir eine `for`-Schleife programmieren:

```
for line in file_object:
```

**Beispiel 69** In folgendem Modul `webpages` wird das Lesen einer Datei demonstriert. In der Datei `webpages.txt` seien auf jeder Zeile gültige URLs auf Webseiten gespeichert:

```
https://scholar.google.com
https://dblp.org
https://www.researchgate.net
https://orcid.org
```

---

<sup>2</sup>Wir haben dieses Prinzip schon auf Objekten vom Typ `list` und `str` angewendet.

Mit einer `for`-Schleife iterieren wir über alle Zeilen der Datei und lesen diese einzeln ein. Für jede Zeile der Datei schneiden wir das Zeilenumbruchzeichen mit Hilfe der Methode `strip` ab. Danach verwenden wir die eingelesene Zeile als Parameter für die Funktion `open_new_tab` des importierten Moduls `webbrowser`. Diese Funktion öffnet die übergebene URL in einem neuen Tab des Browsers.

```
"""
webpages.py
"""

import webbrowser

with open("webpages.txt", "r") as file:
    for url in file:
        url = url.strip("\n")
        webbrowser.open_new_tab(url)
```

#### 4.4.3 Dateien Schreiben

Zum Schreiben von Daten müssen Sie ebenfalls zunächst ein Dateiobjekt öffnen (mit der eingebauten Funktion `open` im Modus `"w"` oder `"a"`). Danach können Sie mit der Methode `write` Daten in das Dateiobjekt schreiben. Der Aufruf

```
file_object.write("Hallo")
```

schreibt z.B. die Zeichenkette `Hallo` in die geöffnete Datei. Hinweis: Die Methode `write` kann – im Gegensatz zur eingebauten Funktion `print` – nur genau ein Argument entgegennehmen.

**Beispiel 70** In folgendem Modul `write_poem` öffnen wir eine Datei `poem.txt` im Modus `"w"` (falls es diese Datei noch nicht geben sollte, wird sie nun erstellt). In das geöffnete Dateiobjekt schreiben wir mit

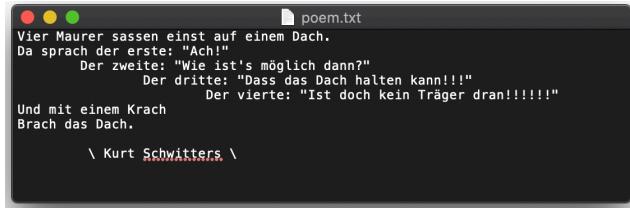


Abbildung 4.1: Die erzeugte Datei poem.txt.

der Methode `write` eine formatierte Zeichenkette. Die erzeugte Datei ist in Abb. 4.1 ersichtlich.

```

"""
write_poem.py
"""

with open("poem.txt", "w") as file:
    file.write("Vier Maurer sassen einst auf einem Dach.\n" +
               "Da sprach der erste: \"Ach!\"\n" +
               "\tDer zweite: \"Wie ist's möglich dann?\"\n" +
               "\t\tDer dritte: \"Dass das Dach halten kann!!!\"\n" +
               "\t\t\tDer vierte: \"Ist doch kein Träger dran!!!!!!\"\n" +
               "Und mit einem Krach\n" +
               "Brach das Dach. \n\n\t \\ Kurt Schwitters \\ ");

```

Das an die Methode `write` übergebene Argument muss zwingend eine Zeichenkette sein. Wollen wir numerische Daten in eine Datei schreiben, müssen wir diese vorher in eine Zeichenkette konvertieren. Ferner ist die direkte Konkatenation von Zahlen und Zeichenketten nicht möglich. Das heisst, dass folgende Anweisung syntaktisch nicht korrekt ist:

```
"Mein Alter: " + 43
```

Mit anderen Worten: Zahlen müssen vor der Konkatenation zwingend in eine Zeichenkette konvertiert werden.

Der einfachste Weg zur Konvertierung bietet die eingebaute Funktion `str`.

**Beispiel 71** In folgendem Programm werden Zeichenketten mit (in Zeichenketten konvertierten) Zahlen konkateniert und ausgegeben:

```
"""
info.py
"""

with open("info.txt", "w") as file:
    age = 43
    height = 1.76
    file.write("Ich bin " + str(age) +
               " Jahre alt und " + str(height) + "m gross.")
```

Eine andere Möglichkeit, Zahlen in Zeichenketten aufzunehmen, bietet die Methode `s.format(args)`, die Sie mit dem Punktoperator auf Zeichenketten `s` aufrufen können. Die Zeichenkette, auf der diese Methode aufgerufen wird, kann sogenannte *Ersatzfelder* enthalten. Ein Ersatzfeld besteht aus einem ganzzahligen Index `i` innerhalb geschweifter Klammern (z.B. "`{0}`"). Die Methode `s.format` gibt eine Kopie der Zeichenkette zurück, bei der jedes Ersatzfeld durch die Werte der entsprechenden Argumente der Methode ersetzt wird. Das heisst, das Ersatzfeld "`{0}`" wird mit dem ersten Argument ersetzt, Ersatzfeld "`{1}`" mit dem zweiten, usw.

**Beispiel 72** In folgendem Beispiel werden die Ersatzfelder "`{0}`" und "`{1}`" mit den Werten aus `age` und `height` ersetzt:

```
"""
info.py
"""

with open("info.txt", "w") as file:
    age = 43
    height = 1.76
    file.write("Ich bin {0} Jahre alt und {1} m gross.".format(age, height))
```

#### 4.4.4 Das Modul csv

Das so genannte *CSV-Format (Comma Separated Values)* ist das gängigste Import- und Exportformat für Tabellenkalkulationen und Datenbanken. Das Standardmodul `csv` bietet Funktionalitäten zum einfachen Lesen und Schreiben von Tabellendaten im CSV-Format.

Mit der Funktion

```
readCSV = csv.reader(file_object, delimiter=',')
```

erhalten wir bspw. ein Reader-Objekt, das über die Zeilen der angegebenen csv-Datei `file_object` iterieren kann (mit dem Parameter `delimiter` können wir das Trennzeichen zwischen einzelnen Werten einer Zeile festlegen, z.B. ein Komma oder ein Leerschlag). Zum Iterieren über diese Zeilen kann man eine `for`-Schleife wie folgt programmieren:

```
for row in readCSV:
```

Jede aus der csv-Datei gelesene Zeile `row` wird als Liste von Zeichenketten zurückgegeben. Das heisst, wir können via Index auf einzelne Elemente

```
entry_1 = row[1]
```

oder mit einer `for`-Schleife auf alle Elemente einer Zeile zugreifen:

```
for entry in row:
```

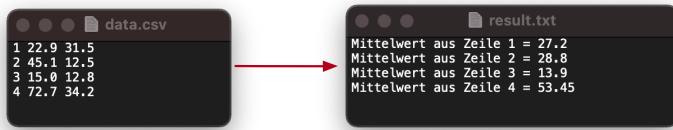


Abbildung 4.2: Aus der Datei `data.csv` wird die Datei `result.txt` erzeugt.

**Beispiel 73** In folgendem Modul laden wir numerische Daten aus einer Datei `data.csv` (siehe Abb. 4.2). Aus jeder Zeile `row` aus `data.csv` erzeugen wir eine Liste, welche die einzelnen Elemente der Zeile enthält. Mit einer `for`-Schleife iterieren wir über diese Listen. Wir berechnen den Mittelwert der Einträge an Position 1 und 2 und schreiben diesen inklusive der Zeilennummer in die Ausgabedatei.

```
"""
compute_mean.py
"""

import csv

out_file = open("result.txt", "w")
with open('data.csv') as in_file:
    readCSV = csv.reader(in_file, delimiter=' ')
    for row in readCSV:
        mean = (float(row[1]) + float(row[2])) / 2
        out_file.write("Mittelwert aus Zeile {} = {}\n".format(row[0], mean))
```

# Kapitel 5

## Eigene Funktionen Programmieren

Beim Programmieren startet man häufig mit der Frage, *was* ein Programm erledigen können muss (man spezifiziert das Problem oder die Probleme, die gelöst werden sollen). Hat man das "was" identifiziert, man muss sich entscheiden, *wie* wir die nötigen Funktionalitäten programmieren.

All unsere bisherigen Programme bestehen jeweils aus einem Modul, in das wir die Anweisungen programmieren, die nacheinander ausgeführt werden sollen (dies entspricht dem Paradigma der imperativen Programmierung). In unseren Programmieranweisungen haben wir hierbei drei Arten von Funktionen (bzw. Methoden) aufgerufen:

- Eingebaute Funktionen
- Funktionen aus Standardmodulen
- Methoden, die man auf Objekten aufruft

**Beispiel 74** *In folgendem Modul `sum_of_square_roots` lösen wir das Problem, die Summe der Quadratwurzeln zweier Zahlen zu berechnen und auszugeben (und verwenden dabei alle Arten von Funktionen/Methoden):*

```

"""
sum_of_square_roots.py
"""

import math

val1 = float(input("Erste Zahl eingeben: ")) # eingebaute Funktionen
val2 = float(input("Zweite Zahl eingeben: "))

res = round(math.sqrt(val1) + math.sqrt(val2), 2) # Funktion aus Standardmodul
print("Summe der Quadratwurzeln: {}".format(res)) # Methode auf str-Objekt

```

Obschon man mit den Funktionen und Methoden der Standardbibliothek recht viele Probleme beim Programmieren lösen kann, ist der Kern der Programmierung das Schreiben eigener Funktionen, die spezifische Probleme lösen.

Die sogenannte *prozedurale Programmierung* ergänzt das Paradigma der imperativen Programmierung mit der Idee, dass wir unser gesamtes Programm in überschaubare Teile zerlegen können. In Python nennt man diese Teile *Funktionen* (je nach Programmiersprache werden diese Teile auch *Unterprogramm*, *Routine* oder *Prozedur* genannt). Innerhalb von Funktionen fassen wir bestimmte Anweisungen zusammen. Bei Bedarf können wir diese Funktionen dann aufrufen (und ihnen dabei ggf. Parameter übergeben und von ihnen Ergebnisse zurück erhalten).

## 5.1 Funktionen Definieren

Eine *Funktion* ist eine Gruppierung von Programmieranweisungen, die unter einem bestimmten Namen zusammengefasst werden. Mit dem Schlüsselwort **def** führen wir eine Funktionsdefinition ein. Darauf fol-

gen der Funktionsname<sup>1</sup>, die in Klammern gesetzte Liste der Parameter und ein Doppelpunkt : (die Klammern sind auch dann nötig, wenn der Funktion keine Parameter mitgegeben werden sollen).

```
def function_name(parameter):
```

Die Anweisungen, die in der Funktion zusammengefasst werden, beginnen auf der nächsten Zeile und müssen eingerückt werden und zwar mit einem Tabulator (= 4 Leerzeichen). Alles was nicht mehr eingerückt ist, gehört nicht zu dieser Funktion.

**Beispiel 75** Folgendes Modul definiert eine Funktion `print_quote()` (ohne Parameter), die ein Zitat auf der Konsole ausgibt.

```
"""
quote.py
"""

# gibt ein Zitat von Steve Jobs auf der Konsole des Computers aus
def print_quote():
    print("Steve Jobs:")
    print("Es ist besser, ein Pirat zu sein, als der Marine beizutreten.")
```

Wird das obige Modul ausgeführt, passiert nichts, da die Funktion `print_quote` nirgends aufgerufen wird.

## 5.2 Funktionen Aufrufen

Funktionsaufrufe einer eigenen Funktion können wir im gleichen Modul platzieren, in dem sich die aufgerufene Funktion befindet. In diesem Fall ist für den Aufruf nur der Bezeichner der Funktion nötig.

---

<sup>1</sup>Wir halten uns an die Konvention `lowercase` oder `lower_case_with_underscore` für Funktionsbezeichner.

**Beispiel 76** In folgendem Modul definieren wir eine Funktion, die wir dann auch gleich aufrufen.

```
"""
print_something.py
"""

def print_something():
    print("something")

print_something() # Aufruf der Funktion
```

Wenn ein Programm grösser und komplexer wird, sollten wir es zur leichteren Wartung in mehrere Module aufteilen. Die in verschiedenen Modulen definierten Funktionen können wir dann aufrufen, indem wir das entsprechende Modul importieren (wie wir das auch schon bei den Standardmodulen gemacht haben). Der Befehl hierzu lautet:

```
import modul_bezeichner
```

wobei `modul_bezeichner` ein Bezeichner einer beliebigen Python Datei ist, deren Funktionen Sie verwenden möchten. Ist ein Modul importiert, können Sie alle Funktionen des importierten Moduls aufrufen. Hierzu ist jetzt aber neben dem Bezeichner der Funktion auch der Name des Moduls und der Punkt-Operator nötig:

```
modul_bezeichner.funktions_bezeichner()
```

Sie können auch nur einzelne Funktionen eines Moduls importieren:

```
from modul_bezeichner import funktions_bezeichner
```

Jetzt kann die Funktion `funktions_bezeichner` wie eine eingebaute Funktion (also direkt) aufgerufen werden.

Sollte sich das zu importierende Modul in einem von Ihnen definierten Package befinden, müssen Sie den entsprechenden Bezeichner des Paketes zur Import-Anweisung hinzufügen. Mit dem reservierten Wort `as` können wir den Bezeichner des importierten Moduls umbenennen:

```
import package_bezeichner.modul_bezeichner as m
```

Aufrufe von Funktionen erfolgen dann durch `m.funktions_bezeichner()`.

**Beispiel 77** In Modul `quote.py` definieren wir zwei Funktionen, die dann in Modul `main.py` aufgerufen werden (nachdem wir das Modul `quote.py` importiert haben).

```
"""
quote.py
"""

# gibt ein Zitat von Steve Jobs auf der Konsole des Computers aus
def print_quote_of_steve():
    print("Steve Jobs:")
    print("Es ist besser, ein Pirat zu sein, als der Marine beizutreten.")

# gibt ein Zitat von Michael Jordan auf der Konsole des Computers aus
def print_quote_of_michael():
    print("Michael Jordan:")
    print("Manche wollen es, manche wünschen es und andere verwirklichen es.")
```

```
"""
main.py
"""

import quote as q

q.print_quote_of_michael()
q.print_quote_of_steve()
q.print_quote_of_michael()
```

## 5.3 Parameter an Funktionen Übergeben

Wird eine Variable innerhalb einer Funktion definiert (einer Variablen wird also innerhalb einer Funktion ein Wert zugewiesen), so kann auch nur innerhalb *dieser* Funktion auf sie zugegriffen werden. Solche Variablen nennt man *lokale Variablen*<sup>2</sup>. Verschiedene Funktionen können lokale Variablen mit dem gleichen Bezeichner definieren. Da jede Funktion nur ihre lokalen Variablen sieht, aber nicht die der anderen Funktionen, können hierbei keine Missverständnisse auftreten.

Ein *tatsächlicher Parameter* oder *Argument* ist ein Wert, der einer Funktion mitgegeben wird, wenn diese aufgerufen wird. Rufen wir zum Beispiel die Funktion `sqrt` aus dem Modul `math` auf

```
math.sqrt(17)
```

wäre der Wert 17 das Argument oder der tatsächliche Parameter.

Die Parameterliste im Funktionskopf definiert mit Bezeichnern die so genannten *formalen Parameter*. Die Bezeichner der formalen Parameter können wir innerhalb der Funktion als lokale Variablen verwenden, wenn wir die Werte der entsprechenden Argumente benötigen. Der Kopf der Funktion `sqrt` sieht also bspw. so aus

```
def sqrt(val):
```

wobei `val` dem formalen Parameter entspricht.

---

<sup>2</sup>Python sieht auch den Gebrauch von *globalen Variablen* vor. Diesen Variablentypen betrachten wir hier allerdings nicht.

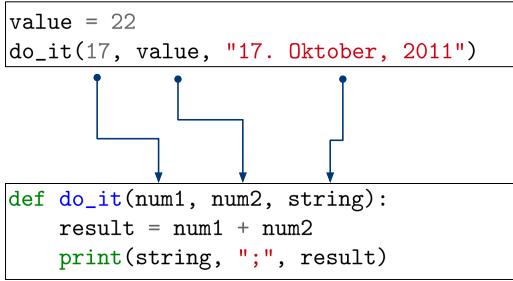


Abbildung 5.1: Parameterübergabe durch einen Funktionsaufruf: Alle Argumente werden den formalen Parametern zugewiesen.

Bei einem Funktionsaufruf wird jedes Argument dem entsprechenden formalen Parameter zugewiesen. Das heisst, das erste Argument wird dem ersten formalen Parameter zugewiesen, das zweite Argument dem zweiten formalen Parameter, etc. Deshalb müssen beim Aufruf die tatsächlichen Parameter und die formalen Parameter übereinstimmen.

**Beispiel 78** In Abb. 5.1 ist die Zuweisung der tatsächlichen zu den formalen Parametern illustriert. Beim Aufruf der Funktion `do_it` werden die tatsächlichen Parameter `17`, `value` und `"17. Oktober, 2011"` an die Funktion übergeben. Die tatsächlichen Parameter werden nun den formalen Parametern zugewiesen. Es geschieht also folgendes:

```

num1 = 17
num2 = 22
string = "17. Oktober, 2011"

```

Das Konzept von parametrisierten Funktionen ermöglicht uns das Erstellen von flexiblem Quellcode. Das heisst, wir können Quellcode definieren, der im Prinzip immer das Gleiche tut, dabei aber die Anweisungen bei jedem Aufruf mit unterschiedlichen Operanden ausführt.

**Beispiel 79** Die Funktion `add` im Modul `arithmetic` kann mit unterschiedlichen tatsächlichen Parametern aufgerufen werden:

```
"""
arithmetic.py
"""

def add(op1, op2):
    result = op1 + op2
    print("{0} + {1} = {2}".format(op1, op2, result))
```

```
"""
main.py
"""

import arithmetic

# Aufruf der gleichen Funktion mit unterschiedlichen Werten
arithmetic.add(3, 4)
arithmetic.add(-7, 3)
arithmetic.add(17, 17)
```

Manchmal ist es nötig oder nützlich, bei der Definition einer Funktion Standardwerte für einen oder mehrere formale Parameter anzugeben. Für formale Parameter, die einen Standardwert besitzen, muss beim Aufruf nicht zwingend ein tatsächlicher Parameter mitgegeben werden (das heisst, wir können die Funktion mit weniger Argumenten aufrufen, als formale Parameter definiert sind).

**Beispiel 80** Die Funktion `print_student` in folgendem Modul definiert drei formale Parameter `name`, `major`, `semester`. Dem zweiten und dritten Parameter sind mit einer Zuweisungsoperation bereits im Funktionskopf Standardwerte zugewiesen:

```
major="Informatik", semester=1
```

Dies ermöglicht es, die Funktion mit einem, zwei oder drei tatsächlichen Parametern aufzurufen. Werden für die vordefinierten formalen Para-

meter keine Argumente übergeben, werden nun einfach die Standardwerte verwendet.

```
"""
student_printer.py
"""

def print_student(name, major="Informatik", semester=1):
    print(name, "studierte", major, "in Semester:", semester)

print_student("Jakob")
print_student("Mark", "Pharmazie")
print_student("Mike", "Mathematik", 3)

print_student("John", semester=5)
print_student("Sandra", major="Geologie")
```

Die ersten drei Zeile der Ausgabe lauten:

*Jakob studiert Informatik in Semester: 1*

*Mark studiert Pharmazie in Semester: 1*

*Mike studiert Mathematik in Semester: 3*

Wir können die Bezeichner der formalen Parameter beim Aufruf auch explizit angeben:

```
print_student(name="Sandra", major="Geologie")
```

Dies ermöglicht dann zum Beispiel, den Standardwert von *major* zu übernehmen und dafür den formalen Parameter *semester* gemäss Argument zu definieren. Die Anweisung:

```
print_student("John", semester=5)
```

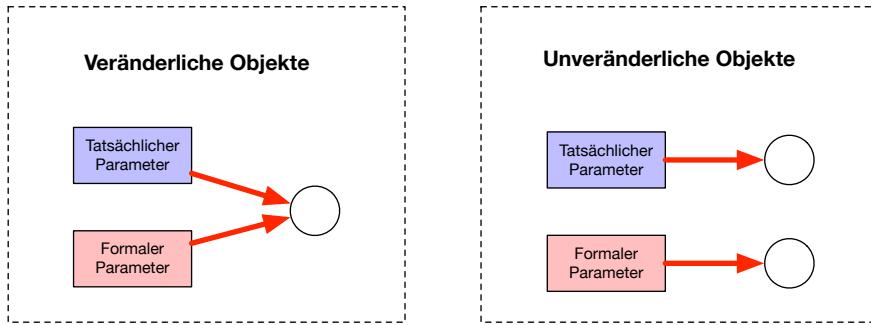


Abbildung 5.2: Tatsächliche und formale Parameter: Unterschied zwischen veränderlichen und unveränderlichen Objekten.

führt somit zur Ausgabe:

*John studiert Informatik in Semester: 5*

Hinweis: Wir haben bereits Funktionen mit Standardwerten verwendet bzw. die Standardwerte mit eigenen Argumenten überschrieben: Bspw. die Funktion `print` (mit den formalen Parametern `end` und `sep`) oder die Funktion `quantiles` des Moduls `statistics` (mit dem formalen Parameter `n`).

## 5.4 Verändern von Parametern in Funktionen

Python unterscheidet bei der Parameterübergabe zwischen *veränderlichen* und *unveränderlichen Objekten* (siehe Abb. 5.2):

- Wenn ein veränderliches Objekt als Parameter an eine Funktion übergeben wird, werden der formale und der tatsächliche Parameter *Aliase* voneinander (beide Parameter zeigen auf das gleiche Objekt). Wenn wir nun innerhalb der Funktion den formalen Parameter ändern, so ändern wir auch den Status des tatsächlichen Parameters ausserhalb der Funktion.

- Wird hingegen ein unveränderliches Objekt als Parameter an eine Funktion übergeben, dürfen wir uns die tatsächlichen und die formalen Parameter als separate, unabhängige Kopien vorstellen<sup>3</sup>. Das heisst, Änderungen, die am formalen Parameter gemacht werden, haben *keinen* Einfluss auf den tatsächlichen Parameter. Springt der Kontrollfluss aus der Funktion zurück, besitzt der tatsächliche Parameter immer noch den gleichen Wert wie vor dem Funktionsaufruf.

Beispiele für unveränderliche Objekte sind z.B. Zahlen (`int` oder `float`) und Zeichenketten (`str`), während Listen (`list`) ein Beispiel für veränderliche Objekte darstellen.

**Beispiel 81** Das folgende Programm illustriert die Unterschiede von veränderlichen und unveränderlichen Objekten bei der Parameterübergabe an Funktionen (siehe auch Abb. 5.3).

```
"""
parameters.py
"""

def change(f1, f2, f3):
    print("Werte bei Start von change:", f1, f2, f3)
    f1 = 0
    f2[0] = 0
    f3 = [0]
    print("Werte am Ende von change:", f1, f2, f3)

value = 99
list1 = [99]
list2 = [99]

print("Werte vor Aufruf von change:", value, list1, list2)
change(value, list1, list2)
print("Werte nach Aufruf von change:", value, list1, list2)
```

---

<sup>3</sup>Hinweis: Technisch gesehen ist das nicht ganz korrekt.

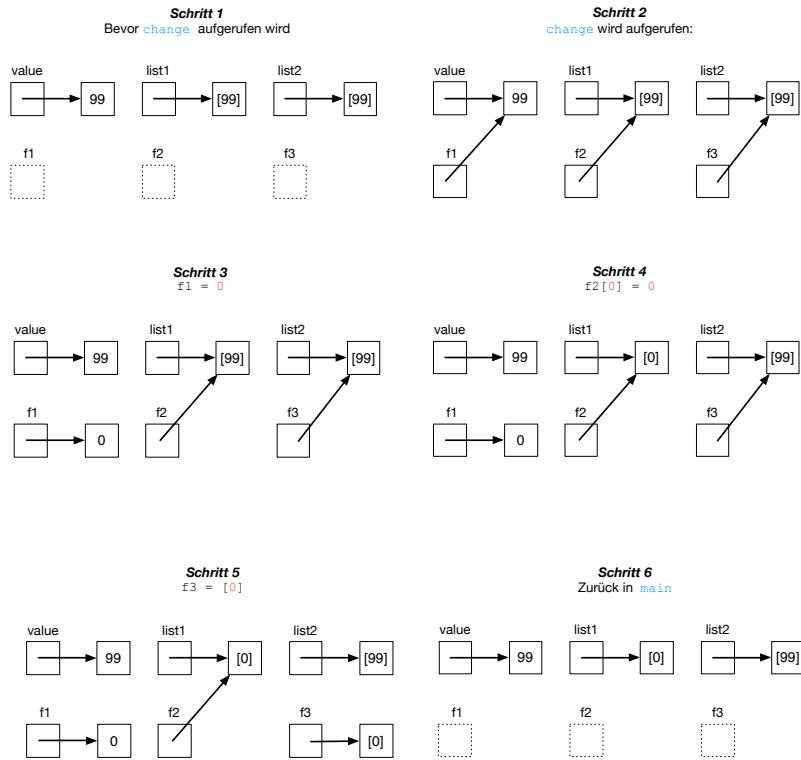


Abbildung 5.3: Veränderung der tatsächlichen und formalen Parameter.

Der erste Parameter, der an `change` übergeben wird, ist ein unveränderliches Objekt vom Typ `int`. Die zwei weiteren Parameter sind Objekte vom veränderlichen Typ `list`. Beim Aufruf der Funktion `change` werden nun die drei tatsächlichen Parameter `value`, `list1` und `list2` in die formalen Parameter `f1`, `f2` und `f3` kopiert.

Innerhalb der Funktion `change` werden die drei formalen Parameter verändert. Der Variablen `f1` wird ein neuer Wert zugewiesen, der Listeninhalt der Variablen `f2` wird verändert und der letzten Variablen `f3` wird ein neues Listen Objekt zugewiesen. Danach kehrt der Kontrollfluss zurück zu `main` und die Werte der tatsächlichen Parameter werden erneut ausgegeben.

Das Programm führt zu folgender Ausgabe:

Werte vor Aufruf von `change`: 99 [99] [99]

Werte bei Start von `change`: 99 [99] [99]

Werte am Ende von `change`: 0 [0] [0]

Werte nach Aufruf von `change`: 99 [0] [99]

Die Variable `value` wird also nicht verändert, da die Änderung nur an der Kopie `f1` vorgenommen wird. Der letzte Parameter `list2` referenziert immer noch das Originalobjekt mit der Liste [99]. Die einzige sichtbare Änderung ist die, die am zweiten tatsächlichen Parameter `list1` vorgenommen wurde. Die Änderung an `f2` wirkt sich auch auf Objekt `list1` aus, da diese Aliase voneinander sind.

## 5.5 Die `return` Anweisung

Wir können Funktionen als eine Art *Black-Box* betrachten, denen wir einen oder mehrere Werte übergeben (die Parameter) und die uns einen oder mehrere Werte zurückgeben (die Rückgabe) (siehe Abb. 5.4).

Python unterscheidet zwischen zwei Arten von Funktionen:

- Funktionen *ohne echte* Rückgabe
- Funktionen *mit* Rückgabe, welche einen (oder mehrere) Wert(e) an den aufrufenden Teil des Programms zurückgeben, wenn diese abgearbeitet sind

Tatsächlich geben in Python auch Funktionen ohne Rückgabe einen



Abbildung 5.4: Im Allgemeinen erwarten Funktionen Parameter und erzeugen eine Rückgabe.

Wert zurück, nämlich `None` (ein reserviertes Wort).

Eine Funktion mit *echter* Rückgabe beinhaltet eine `return`-Anweisung im Funktionsrumpf. Eine `return`-Anweisung besteht aus dem reservierten Wort `return` gefolgt von einem Wert oder einem Ausdruck, der zurückgegeben werden soll.

**Beispiel 82** Das folgende Programm `addition.py` definiert eine Funktion, welche zwei Parameter erwartet und eine Rückgabe erzeugt.

```

"""
addition.py
"""

def add(val1, val2):
    result = val1 + val2
    return result

r1 = add(7, 9)
r2 = add(-8, 13)
r3 = add(r1, r2)

print("Resultat 1 =", r1)
print("Resultat 2 =", r2)
print("Resultat 3 =", r3)

```

Wird eine `return`-Anweisung in einer Funktion ausgeführt, so wird der definierte Ausdruck an die aufrufende Funktion zurückgegeben. Die Rückgabe einer Funktion kann dann entgegengenommen und verwendet werden<sup>4</sup>. Z.B. kann man die Rückgabe ausgeben:

---

<sup>4</sup>Hinweis: Die Rückgabe einer Funktion kann aber auch ignoriert werden. Das heisst, die

```
print("Die Summe lautet:", add(7, 8))
```

Oder man weist die Rückgabe einer Variablen zu:

```
r1 = add(7, 9)
```

Hinweis: *Zurückgeben* ist nicht das gleiche wie *Ausgeben!* Eine Funktion, die etwas ausgibt, zeigt mit Hilfe der Funktion `print` eine Zeichenkette in der Konsole an. Eine Funktion, die mit `return` etwas zurückgibt, gibt einen Wert zurück ohne diesen anzuzeigen.

Wir können in Funktionen auch `return`-Anweisungen *ohne* Rückgabewert definieren:

```
return
```

Wird in einer Funktion dieses "leere" `return` erreicht, kehrt der Kontrollfluss *sofort* aus der Funktion zurück (mit der Rückgabe `None`). Die Anweisungen, die unterhalb einer `return`-Anweisung programmiert sind, werden nicht ausgeführt (in diesem Fall bedeutet `return` also eher "Zurückkehren" als "Zurückgeben"). Dies gilt übrigens auch für `return`-Anweisungen *mit* Rückgabe – die Funktion wird auch hier *sofort* verlassen.

In Python kann eine Funktion mehrere Werte gleichzeitig zurückgeben, indem nach der `return`-Anweisung mehrere durch Komma getrennte Rückgabe muss nicht zwingend verarbeitet werden.

Werte aufgezählt werden. Folgende Anweisung gibt bspw. drei Werte vom Typ `int`, `float` und `str` zurück:

```
return 1, 2.0, "Drei"
```

Wenn Sie eine solche Funktion in einer Zuweisungsanweisung aufrufen, können Sie die zurückgegebenen Werte einzelnen Variablen zuweisen.

**Beispiel 83** Die Funktion `basic_operations` berechnet die Summe, die Differenz, das Produkt und den Quotienten der formalen Parameter `op1` und `op2` und gibt alle Resultate zurück. Die Rückgaben werden den Variablen `r1`, `r2`, `r3`, `r4` zugewiesen.

```
"""
arithmetic.py
"""

def basic_operations(op1, op2):
    s = op1 + op2
    d = op1 - op2
    p = op1 * op2
    q = op1 / op2
    return s, d, p, q

r1, r2, r3, r4 = basic_operations(4, 6)
print(r1, r2, r3, r4)
```

Die Ausgabe lautet:

10 -2 24 0.6666666666666666

## 5.6 Teilen-und-Beherrschen

Wenn wir komplexe Probleme – die aus verschiedenen Teilproblemen bestehen – mit einer einzigen Funktion lösen wollen, führt dies unwei-

gerlich zu grossen und unübersichtlichen Funktionen. Als Grundsatz gilt: Jede Funktion sollte *genau eine* Aufgabe oder Verantwortung übernehmen. Das heisst, wenn wir komplexes Verhalten modellieren, lohnt sich das Aufteilen der gesamten Funktionalität in mehrere kleine Funktionen, die typischerweise leichter zu erstellen sind. Diese in der Informatik weit verbreitete Strategie ist bekannt unter dem Namen *Teilen-und-Beherrschen*. Die Vorteile dieser Strategie sind:

- Quellcode ist besser verständlich
- Quellcode ist wiederverwendbar
- Quellcode ist leichter zu testen
- Quellcode kann in Teams und somit schneller erstellt werden

Wir betrachten in diesem Abschnitt ein Beispiel, das die Idee von Teilen-und-Beherrschen illustrieren soll. Wir schreiben ein Programm, das *Englisch* in sogenanntes *Pig Latin* übersetzen soll. Pig Latin ist eine Sprache, in der jedes Wort eines Satzes nach folgenden Regeln modifiziert wird.

- Bei Wörtern, die mit einem Konsonanten beginnen, wird der erste Buchstabe des Wortes ans Ende gestellt und mit einem *ay* ergänzt. Das Wort *happy* wird somit zu *appyhay* oder das Wort *birthday* wird zu *irthdaybay*.
- Doppelkonsonanten wie bspw. “ch” oder “st” am Anfang eines Wortes werden zusammen ans Ende des Wortes gerückt und mit einem *ay* ergänzt (*grapefruit* wird zu *apefruitgray*).
- Wörter die mit einem Vokal beginnen, werden mit einem *yay* am Ende des Wortes ergänzt (*enough* wird zu *enoughyay*).

Wir identifizieren folgende Aufgaben und Verantwortungen, die wir programmieren müssen:

- Einen Satz  $s$  vom Benutzer einlesen und die Übersetzung ausgeben
- Satz  $s$  in Pig Latin übersetzen. Besteht aus der Teilaufgabe:
  - Jedes einzelne Wort  $w$  aus  $s$  in Pig Latin übersetzen. Besteht aus den Teilaufgaben:
    - \* Überprüfen, ob  $w$  mit Vokal beginnt
    - \* Überprüfen, ob  $w$  mit Doppelkonsonanten beginnt

Wir teilen die Aufgaben und Verantwortungen auf mehrere Funktionen auf. Wir starten mit dem Modul `piglatin.py`. Wir erfragen zunächst vom Benutzer einen Englischen Satz, übersetzen diesen und geben die Übersetzung anschliessend aus. Dieses Modul übernimmt also die erste Verantwortung aus unserer Liste. Die ganze Arbeit des Übersetzens erledigt das Modul `translator`, das wir importieren und dessen Funktion `translate_sentence` wir aufrufen.

```
"""
piglatin.py
"""

import translator

decision = "y"
while decision == "y":
    # Satz einlesen
    sentence = input("Geben Sie einen Englischen Satz ein"
                     "(ohne Interpunktionszeichen): ")
    # Satz übersetzen und ausgeben
    translation = translator.translate_sentence(sentence)
    print("Übersetzung: ", translation)
    decision = input("Weitere Übersetzung? (y/n): ")
```

Das Modul `translator` bietet mit der Funktion `translate_sentence` den fundamentalen Service an, einen ganzen Satz als Parameter entgegenzunehmen und diesen in Pig Latin zu übersetzen. Die Funktion `translate_sentence` ersetzt hierzu die Eingabe `sentence` zunächst mit einer Kopie aus Kleinbuchstaben. Danach wird aus `sentence` mit der Methode `split` eine Liste aus einzelnen Wörtern generiert.

Für jedes Wort aus dieser Liste wird nun die Hilfsfunktion `translate_word` aufgerufen. Diese Funktion übersetzt das einzelne Wort in Pig Latin und gibt das Resultat zurück. Jedes übersetzte Wort wird zusammen mit einem Leerschlag zur Zeichenkette `result` konkateniert und letztlich zurückgegeben.

Die Funktion `translate_word` verwendet zwei weitere Hilfsfunktionen, nämlich `starts_with_vowel` und `starts_with_blend`. Diese beiden Funktionen geben `True` zurück, wenn das übergebene Wort mit einem Vokal bzw. mit einem Doppelkonsonanten beginnt. Je nach Wort wird also eine der drei Regeln zur Modifikation eines einzelnen Wortes angewendet.

Eine mögliche Ein- und Ausgabe:

```
Geben Sie einen Englischen Satz ein (ohne Interpunktionszeichen):  
It is fun to translate English to Pig Latin  
Übersetzung: ityay isyay unfay otay anslatetray englishyay  
otay igpay atinlay  
Weitere Übersetzung? (y/n): n
```

```

"""
translator.py
"""

# Übersetzt sentence in Pig Latin
def translate_sentence(sentence):
    result = ""
    sentence = sentence.lower()
    word_list = sentence.split()
    for word in word_list:
        result += translate_word(word) + " "
    return result

# Übersetzt word in Pig Latin
def translate_word(word):
    result = ""
    if starts_with_vowel(word):
        result = word + "ay"
    else:
        if starts_with_blend(word):
            result = word[2:] + word[:2] + "ay"
        else:
            result = word[1:] + word[:1] + "ay"
    return result

# Überprüft, ob word mit einem Vokal beginnt
def starts_with_vowel(word):
    vowels = "aeiou"
    start_letter = word[0]
    if vowels.find(start_letter) == -1:
        return False
    else:
        return True

# Überprüft, ob word mit einem Doppelkonsonanten beginnt
def starts_with_blend(word):
    if len(word) < 2:
        return False
    vowels = "aeiou"
    return word[0] not in vowels and word[1] not in vowels

```

# Kapitel 6

## Weitere Datenstrukturen

### 6.1 Zweidimensionale Listen

Bis hierhin haben wir nur *eindimensionale Listen* verwendet. *Zweidimensionale Listen* besitzen – wie der Name impliziert – Werte in zwei Dimensionen. Man kann sich zweidimensionale Listen als Tabellen mit Zeilen und Spalten vorstellen (siehe Abb. 6.1).

In zweidimensionalen Listen benötigen wir zwei Indizes, um einen Wert anzusprechen (ein Index für die Zeile und ein zweiter Index für die Spalte). Folgende Anweisung weist der Variablen `value` den Wert zu, der in der Tabelle `table` in der zweiten Zeile und der dritten Spalte steht (das wäre in unserem Beispiel der Wert 3):

```
value = table[1][2]
```

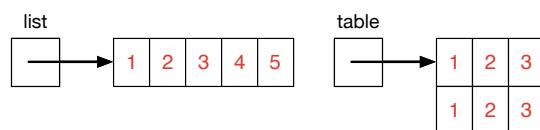


Abbildung 6.1: Ein- vs. zweidimensionale Listen.

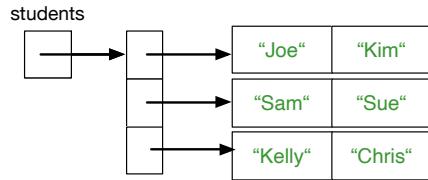


Abbildung 6.2: Eine zweidimensionale Liste ist eine Liste aus Listen.

Obschon die Vorstellung einer Tabelle nützlich sein kann, ist dies nicht ganz korrekt: Eine zweidimensionale Liste ist eigentlich *eine Liste aus Listen*. Das heisst, die einzelnen Elemente in der Liste sind wiederum Listen. Dies erkennt man auch daran, wie man zweidimensionale Listen definiert: Mit einem ersten Klammerpaar [] für die Definition der äusseren oder ersten Liste, welche dann beliebig viele Listen als Elemente enthält.

**Beispiel 84** Eine zweidimensionale Liste `students` kann bspw. folgendermassen erstellt werden (siehe auch Abb. 6.2):

```
students = [["Joe", "Kim"], ["Sam", "Sue"], ["Kelly", "Chris"]]
```

Deshalb gibt zum Beispiel

```
len(students)
```

den Wert 3 zurück (Anzahl Elemente in der Liste `students`), während

```
len(students[0])
```

den Wert 2 zurückgibt (Anzahl Elemente in der Liste, die sich an Position 0 in der Liste `students` befindet).

**Beispiel 85** In folgendem Programm `twodimensional` wird eine Liste `table` erstellt, welche wiederum 4 Listen mit Zahlen enthält. Der Benutzer erhält danach die Möglichkeit einen Eintrag in der Liste `table` anzupassen.

```
"""
twodimensional.py
"""

table = [[1, 3, 5], [3, 6, 1], [4, 9, 8], [1, 1, 2]]
print("Aktuelle Daten:", table)

row = int(input("Welche Zeile wollen Sie ändern: "))
col = int(input("Welche Spalte wollen Sie ändern: "))
new = int(input("Neuer Wert: "))

table[row][col] = new
print("Neue Daten:", table)
```

```
Aktuelle Daten: [[1, 3, 5], [3, 6, 1], [4, 9, 8], [1, 1, 2]]
Welche Zeile wollen Sie ändern: 2
Welche Spalte wollen Sie ändern: 1
Neuer Wert: -1
Neue Daten: [[1, 3, 5], [3, 6, 1], [4, -1, 8], [1, 1, 2]]
```

Verschachtelte `for`-Schleifen eignen sich sehr gut, um über die Inhalte zweidimensionaler Listen zu iterieren. Mit der äusseren Schleife besuchen wir die Listen innerhalb der Liste (die Zeilen). Mit der inneren Schleife besuchen wir dann jedes Element der inneren Listen.

**Beispiel 86** Folgendes Code-Fragment gibt bspw. alle Elemente der Liste `table` aus:

```
1 2
3 4
5 6
```

```

table = [[1, 2], [3, 4], [5, 6]]

for row in table:
    for element in row:
        print(element, end="\t")
    print() # jede innere Liste auf einer neuen Zeile anzeigen

```

**Beispiel 87** In folgendem Programm wird die initial leere Liste `table` mit einer verschachtelten `for`-Schleife iterativ mit Zufallszahlen gefüllt. Bei jeder Iteration der äusseren Schleife wird zuerst eine neue leere Liste `row` erstellt. In der inneren Schleife wird `row` dann mit Zufallszahlen zwischen 1 und 10 gefüllt. Die fertige Liste `row` wird schliesslich zu `table` hinzugefügt.

```

"""
random_table.py
"""

import random

table = []
for i in range(3):
    row = []
    for j in range(3):
        row.append(random.randint(1, 10))
    table.append(row)

print(table)

```

Eine mögliche Ausgabe wäre bspw.:

`[[7, 1, 2], [2, 3, 6], [1, 10, 8]]`

**Beispiel 88** Für dieses Beispiel nehmen wir an, dass wir Resultate von vier Sportmannschaften zur Verfügung haben. Die Resultate sind in der zweidimensionalen Liste `results` zusammengefasst. Jede Zeile in `results` enthält die Anzahl Siege (Spalte 0), die Anzahl Unentschieden (Spalte 1) und die Anzahl Niederlagen (Spalte 2) für jedes

der vier Teams. Das heisst, `results` besitzt vier Zeilen und drei Spalten.

Das folgende Programm `league` berechnet mit einer `for`-Schleife die Anzahl Punkte für jedes der vier Teams (Siege ergeben in diesem Beispiel je drei und Unentschieden je einen Punkt). Für jedes Element in der Liste `teams` wird dann die berechnete Punktzahl gespeichert.

```
"""
league.py
"""

win_points = 3
tie_points = 1

# Anzahl Siege, Unentschieden und Niederlagen pro Team
results = [ [1, 2, 0],
            [2, 0, 1],
            [1, 1, 1],
            [1, 0, 2] ]

teams = [ ["FCQ", 0], ["FCW", 0], ["FCE", 0], ["FCR", 0] ]

for i in range(len(teams)):
    points = results[i][0] * win_points
    points += results[i][1] * tie_points
    teams[i][1] = points

print(teams)
```

Schliesslich wird die Liste `teams` ausgegeben:

```
[['FCQ', 5], ['FCW', 6], ['FCE', 4], ['FCR', 3]]
```

### 6.1.1 Kopieren von Listen

Listen sind veränderliche Objekte und wenn wir veränderliche Objekte kopieren, entstehen *Aliase*. Das heisst, zwei Variablen zeigen auf das gleiche Objekt. Wenn wir somit die eine Variable ändern, wird die

andere Variable auch geändert.

### Beispiel 89

```
>>> list1 = [1, 2, 3, 4]
>>> list2 = list1 # Alias
>>> list2[1] = 999
>>> list2
[1, 999, 3, 4]
>>> list1
[1, 999, 3, 4]
```

Für veränderliche Objekte wird aber manchmal auch eine *echte* Kopie benötigt, so dass man *eine* der Kopien ändern kann, ohne die *andere* auch gleich zu ändern. Es gibt verschiedene Möglichkeiten, wie wir eine echte Kopie einer Liste erhalten können:

- Wir verwenden die *Slicing* Operation, um eine Kopie der gesamten Liste zu erhalten:

### Beispiel 90

```
>>> list1 = [1, 2, 3, 4]
>>> list2 = list1[:]
>>> list2[1] = 999
>>> list1
[1, 2, 3, 4]
>>> list2
[1, 999, 3, 4]
```

- Wir verwenden die Funktion `copy` des Standardmoduls `copy`.

### Beispiel 91

```
>>> import copy
```

```

>>> list1 = [1, 2, 3, 4]
>>> list2 = copy.copy(list1)
>>> list2[1] = 999
>>> list1
[1, 2, 3, 4]
>>> list2
[1, 999, 3, 4]

```

## 6.2 Wörterbücher – Die Datenstruktur `dict`

Ein weiterer nützlicher Datentyp, der in Python eingebaut ist, ist das Wörterbuch (`dict`). Wörterbücher werden in anderen Sprachen ”assoziative Speicher” oder ”assoziative Arrays” genannt. Im Gegensatz zu Listen, die durch einen Bereich von Zahlen indiziert werden, werden Wörterbücher durch Schlüssel indiziert, die ein beliebiges unveränderliches Objekt sein können (z.B. können Zeichenketten oder Zahlen als Schlüssel verwendet werden).

Am besten stellt man sich ein Wörterbuch als eine Menge von Schlüssel-Wert-Paaren vor, mit der Anforderung, dass die Schlüssel eindeutig sind (innerhalb eines Wörterbuchs). Ein Klammerpaar erzeugt ein leeres Wörterbuch: `{}`. Durch Platzieren einer durch Kommas getrennten Liste von Schlüssel-Wert-Paaren innerhalb geschweifter Klammern werden dem Wörterbuch anfänglich Elemente hinzugefügt. Die Schlüssel und die Werte werden dabei jeweils mit einem Doppelpunkt getrennt.

### Beispiel 92

```

>>> my_dict = {}
>>> my_dict = {1: "Apple", 2: "Ball"}

```

```

>>> my_dict
{1: 'Apple', 2: 'Ball'}
>>> my_dict = {"Name": "John", "Nr": 2567}
>>> my_dict
{'Name': 'John', 'Nr': 2567}

```

Die Hauptoperationen auf einem Wörterbuch sind das Speichern eines Wertes mit einem beliebigen Schlüssel und das Extrahieren des Wertes, bei einem gegebenen Schlüssel. Der Zugriff auf einen Wert aus einem Wörterbuch funktioniert ähnlich zu einem Zugriff in einer Liste. Statt dem Index gibt man aber innerhalb eckiger Klammern den Schlüssel an.

Wenn Sie in einem Wörterbuch einen Wert unter Verwendung eines Schlüssels speichern, der bereits verwendet wird, wird der alte Wert, der mit diesem Schlüssel verbunden war, überschrieben. Wenn Sie einen neuen Schlüssel verwenden, wird ein neues Schlüssel-Wert-Paar erstellt:

### Beispiel 93

```

>>> my_dict = {"Name": "John", "Alter": 26}
>>> my_dict
{'Name': 'John', 'Alter': 26}
>>> my_dict["Alter"] = 27
>>> my_dict
{'Name': 'John', 'Alter': 27}
>>> my_dict["Raum"] = 123
>>> my_dict
{'Name': 'John', 'Alter': 27, 'Raum': 123}

```

Zu Lesen von Daten aus einem Wörterbuch können die Schlüssel entweder innerhalb eckiger Klammern oder als Parameter in der Methode `get` verwendet werden. Wenn wir die eckigen Klammern verwenden, wird ein Fehler ausgelöst, falls ein Schlüssel nicht im Wörterbuch gefunden wird. Umgekehrt gibt die Methode `get` `None` zurück, wenn der Schlüssel nicht gefunden wird:

### Beispiel 94

```
>>> my_dict = { 'Name': 'Jack', 'Alter': 26}
>>> my_dict["Name"]
'Jack'
>>> a = my_dict.get("Adresse")
>>> print(a)
None
>>> my_dict["Adresse"]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'Adresse'
```

Wir können einen bestimmten Eintrag in einem Wörterbuch entfernen, indem wir die Methode `pop` verwenden. Diese Methode entfernt einen Eintrag mit dem angegebenen Schlüssel und gibt den Wert zurück. Mit der Methode `clear` können alle Elemente auf einmal entfernt werden.

### Beispiel 95

```
>>> squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
>>> squares.pop(4)
16
>>> squares
{1: 1, 2: 4, 3: 9, 5: 25}
```

```
>>> squares.clear()  
>>> squares  
{}
```

Wir können auch testen, ob ein Schlüssel in einem Wörterbuch verwendet wird oder nicht, indem wir das reservierte Wort `in` verwenden. Beachten Sie, dass dieser Test auf Mitgliedschaft nur für die Schlüssel aber nicht für die Werte gilt.

### Beispiel 96

```
>>> squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}  
>>> 1 in squares  
True  
>>> 25 in squares  
False
```

Mit `for`-Schleifen und den Methoden `keys()`, `values()` und `items()` können Schlüssel, Werte oder Schlüssel-Wert-Paare zusammen durchlaufen werden.

### Beispiel 97 *Gegeben sei ein Wörterbuch*

```
squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

*Die Ausgabe des folgenden Code-Fragmentes lautet 1 2 3 4 5.*

```
for var in squares.keys():  
    print(var, end=" ")
```

*Mit folgendem Code erhalten wir als Ausgabe 1 4 9 16 25.*

```
for var in squares.values():  
    print(var, end=" ")
```

Mit folgendem Code erhalten wir als Ausgabe (1, 1) (2, 4) (3, 9) (4, 16) (5, 25) (dies sind sogenannte Tupel, welche wir im nächsten Abschnitt betrachten).

```
for var in squares.items():
    print(var, end=" ")
```

**Beispiel 98** In folgendem Modul `data_manager` sind zwei Funktionen programmiert. Die Funktion `read_file` definiert ein neues Wörterbuch und füllt dieses mit dem gelesenen Inhalt aus der Datei `capitals.csv` mit folgendem Inhalt:

*England;London  
USA;Washington DC  
Schweiz;Bern*

Die Schlüssel sind dabei jeweils die Namen der Länder und die Werte die zugehörigen Hauptstädte. Die Funktion `input_country` erstellt im als Parameter übergebenen Wörterbuch `country_capital` einen neuen Wert für den Schlüssel `country`.

```
"""
data_manager.py
"""

import csv

# Generiert aus einer Datei mit Ländern und Hauptstädten ein Wörterbuch
def read_file():
    country_capital = {}
    with open("capitals.csv") as capital_file:
        csv_file = csv.reader(capital_file, delimiter=";")
        for row in csv_file:
            country = row[0]
            capital = row[1]
            country_capital[country] = capital
    return country_capital
```

```
# Fügt in das Land-Hauptstadt Wörterbuch einen neuen Eintrag ein
def input_country(country_capital, country):
    capital = input("Hauptstadt von {} eingeben: ".format(country))
    country_capital[country] = capital
```

Das Modul `capital_manager` nutzt die beiden Funktionen aus dem Modul `data_manager`:

```
"""
capital_manager.py
"""

import data_manager

country_capital_dict = data_manager.read_file()

while True:
    country = input("Land eingeben: ")
    capital = country_capital_dict.get(country)
    if capital is not None:
        print("Hauptstadt: ", capital)
    else:
        decision = input("Land noch nicht erfasst. Land erfassen? (y/n) ")
        if decision == "y":
            data_manager.input_country(country_capital_dict, country)
    another = input("Weitere Abfrage? (y/n): ")
    if another == "n":
        break
```

Eine mögliche Ein- und Ausgabe sieht folgendermassen aus:

```
Land eingeben: Schweiz
Hauptstadt: Bern
Weitere Abfrage? (y/n): y
Land eingeben: Schweden
Land noch nicht erfasst. Land erfassen? (y/n) y
Hauptstadt von Schweden eingeben: Stockholm
Weitere Abfrage? (y/n): y
Land eingeben: Schweden
Hauptstadt: Stockholm
Weitere Abfrage? (y/n): n
```

## 6.3 Tupel – Die Datenstruktur `tuple`

Ein *Tupel* besteht – ähnlich wie Listen – aus einer Anzahl von Werten, die durch Kommas getrennt sind. Tupel können auf verschiedene Weisen definiert werden:

- Mit einem Klammerpaar zur Kennzeichnung des leeren Tupels:  
()
- Trennen der Elemente durch Komma: a, b, c oder (a, b, c)
- Verwendung der eingebauten Funktion `tuple` zur Konvertierung einer Liste oder einer Zeichenkette in ein Tupel.

### Beispiel 99

```
>>> t1 = () # leeres Tupel
>>> t1
()
>>> t2 = (1, 2, "a")
>>> t2
(1, 2, 'a')
>>> t3 = tuple([1, 2, 3]) # Liste -> Tupel
>>> t3
(1, 2, 3)
>>> t4 = tuple("abc") # Zeichenkette -> Tupel
>>> t4
('a', 'b', 'c')
```

Hinweis: Die Kommas und nicht die Klammern bilden das Tupel. Die Klammern sind optional, ausser im Fall eines leeren Tupels oder wenn

sie zur Vermeidung syntaktischer Mehrdeutigkeit erforderlich sind<sup>1</sup>.

Obwohl Tupel ähnlich wie Listen erscheinen mögen, werden sie oft in verschiedenen Situationen und für verschiedene Zwecke verwendet:

- Tupel sind *unveränderliche* Folgen von Elementen, auf die durch *Entpacken* (siehe weiter unten) zugegriffen wird.
- Listen sind *veränderliche* Folgen von Elementen und Zugriffe auf Elemente erfolgen typischerweise über Indizes.

Tupel unterstützen die Indexierung, einige eingebaute Funktionen wie z.B. `len` oder `min`, sowie einige Methoden, die Sie bereits von Listen kennen (z.B. die Methode `index()`):

### Beispiel 100

```
>>> t = 12345, 54321, "Hallo!"  
>>> t[1]  
54321  
>>> len(t)  
3  
>>> t.index(54321)  
1
```

Alle Methoden, die ein Tupel verändern würden (z.B. `append` oder `remove`) sind hingegen nicht erlaubt (Tupel sind eben unveränderlich).

Die Zuweisung `t = 12345, 54321, "Hallo!"` ist ein Beispiel für eine sogenannte *Tupelverpackung*: Die Werte 12345, 54321 und "Hallo!"

---

<sup>1</sup>Zum Beispiel ist `f(a, b, c)` ein Funktionsaufruf mit drei Argumenten, während `f((a, b, c))` ein Funktionsaufruf mit einem 3-Tupel als einziges Argument ist.

werden zusammen in das Tupel `t` gepackt. Auch die umgekehrte Operation ist möglich und wird *Tupelentpackung* genannt. Das Entpacken eines Tupels erfordert, dass sich auf der linken Seite des Zuweisungsoperators so viele Variablen befinden, wie Elemente im Tupel vorhanden sind.

### Beispiel 101

```
>>> t = 12345, 54321, "Hallo!"  
>>> x, y, z = t # Tupelentpackung  
>>> y  
54321  
>>> z  
'Hallo!'
```

Das folgende Beispiel kombiniert Listen und Tupel, indem wir eine Liste aus Tupeln erstellen.

**Beispiel 102** In Modul `read_measurements` lesen wir die folgenden Daten ein (aus einer Datei `measurements.txt`).

```
17.12.01, 113, A  
13.01.02, 167, A  
16.02.02, 145, B
```

Aus jeder Zeile aus der Datei erzeugen wir ein Tupel, das wir der Liste `measurement_list` hinzufügen. Wir erzeugen also eine Liste aus Tupel.

Die Ausgabe des Programmes lautet:

```
[('17.12.01', 113, 'A'), ('13.01.02', 167, 'A'), ('16.02.02',  
145, 'B')]
```

```
"""  
read_measurements.py  
"""  
  
measurement_list = []  
  
with open("measurements.txt") as file:  
    for line in file:  
        line = line.strip("\n")  
        measurements = line.split(", ")  
        t = measurements[0], int(measurements[1]), measurements[2]  
        measurement_list.append(t)  
  
print(measurement_list)
```

## 6.4 Mengen – Die Datenstruktur `set`

Python enthält einen eingebauten Datentypen für Mengen: `set`. Ein `set` ist eine ungeordnete Sammlung von Elementen *ohne* doppelte Elemente. Geschweifte Klammern können zum Erstellen von Mengen verwendet werden<sup>2</sup>. Zu den grundlegenden Verwendungszwecken gehören die Prüfung der Mitgliedschaft (mit dem Schlüsselwort `in`) und die automatische Eliminierung doppelter Einträge.

### Beispiel 103

```
>>> basket = { 'apple', 'orange', 'apple', 'pear', 'orange',  
'banana'}  
>>> print(basket) # eine Menge enthält niemals Duplikate  
{'orange', 'banana', 'pear', 'apple'}  
>>> 'orange' in basket # schneller Test auf Mitgliedschaft  
True
```

---

<sup>2</sup>Um eine leere Menge zu erzeugen, müssen wir aber `set` verwenden, und nicht etwa `{}` was ein leeres Wörterbuch (`dict`) erzeugen würde.

```
>>> 'crabgrass' in basket  
False
```

Mengen sind veränderliche Objekte – aber Sie dürfen keine veränderlichen Elemente enthalten. Da Mengen nicht geordnet sind, hat die Indizierung auf Mengen keine Bedeutung. Es ist somit in einem `set` nicht möglich, auf ein oder mehrere Elemente der Menge über Indizierung oder via *Slicing* zuzugreifen.

Wir können ein einzelnes Element mit der Methode `add` – und mehrere Elemente mit der Methode `update` zur Menge hinzufügen. In allen Fällen werden Duplikate vermieden. Ein bestimmtes Element kann mit der Methode `discard` aus einer Menge entfernt werden.

### Beispiel 104

```
>>> my_set = {1,3}  
>>> my_set.add(2)  
>>> my_set  
{1, 2, 3}  
>>> my_set.update([2, 3, 4])  
>>> my_set  
{1, 2, 3, 4}  
>>> my_set.discard(3)  
>>> my_set  
{1, 2, 4}
```

Die Datenstruktur `set` unterstützt auch mathematische Operationen wie *Vereinigung*, *Schnittmenge* oder die *Differenz* (siehe Abb. 6.3). Die Vereinigung wird in Python entweder mit dem Operator `|` oder mit der Methode `union`, die Schnittmenge mit dem Operator `&` oder

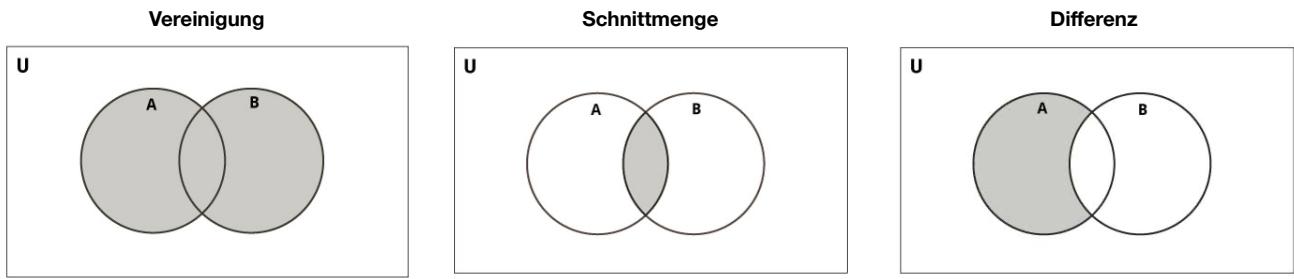


Abbildung 6.3: Vereinigung, Schnittmenge und die Differenz von Mengen  $A$  und  $B$  illustriert.

mit der Methode `intersection` und die Differenz mit dem Operator `-` oder mit der Methode `difference` durchgeführt.

### Beispiel 105

```

>>> A = {1, 2, 3, 4, 5}
>>> B = {4, 5, 6, 7, 8}
>>> C = A | B
>>> C
{1, 2, 3, 4, 5, 6, 7, 8}
>>> B.union(A)
{1, 2, 3, 4, 5, 6, 7, 8}
>>> C = A & B
>>> C
{4, 5}
>>> A.intersection(B)
{4, 5}
>>> C = A - B
>>> C
{1, 2, 3}
>>> A.difference(B)
{1, 2, 3}

```

## Teil II

# Datenprobleme mit Python Lösen

# Kapitel 7

## Daten Sortieren, Filtern und Zusammenfassen

Der *Python Package Index (PyPI)*<sup>1</sup> ist ein *Repository* von Software für die Programmiersprache Python. Entwickler von Python-Modulen können sich bei PyPI registrieren und ihre Pakete dort hochladen.

Die Suche, Installation und Verwaltung externer Pakete ist in PyCharm sehr einfach (siehe auch Abb. 7.1): Starten Sie PyCharm und folgen Sie im Menü PyCharm folgendem Pfad:

```
Preferences... → Project: Name des Projektes → Project  
Interpreter
```

Klicken Sie danach links auf die Schaltfläche + und suchen Sie nach dem gewünschten Paket. Wählen Sie das Paket aus und klicken Sie dann auf die Schaltfläche **Install Package**.

In diesem Kapitel verwenden wir das Paket **pandas** zur einfachen Datenanalyse (z.B. filtern und sortieren von Daten). Um das Paket **pandas** zu laden und mit ihm zu arbeiten, importieren Sie das Paket

---

<sup>1</sup><https://pypi.org>

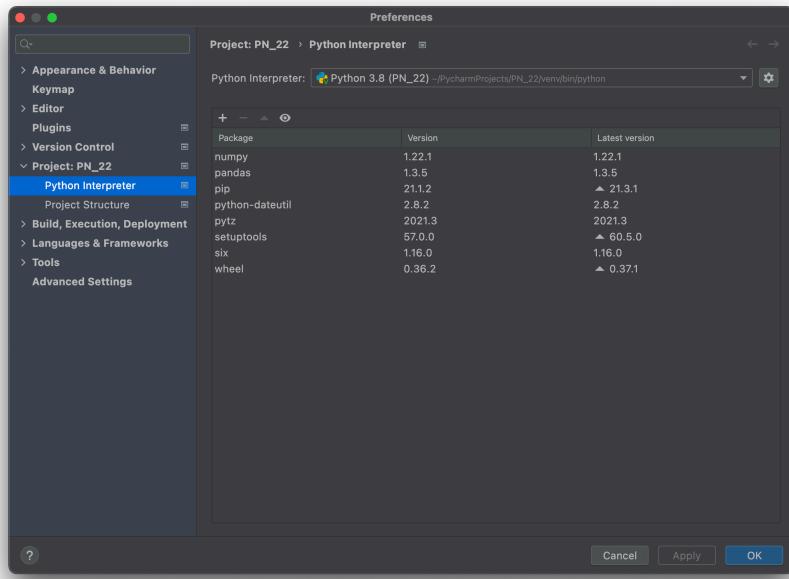


Abbildung 7.1: In der IDE PyCharm nach einem Paket suchen und dieses installieren.

(der unter Programmierer:innen vereinbarte Alias für `pandas` ist `pd`).

```
import pandas as pd
```

Im folgenden gehen wir bei den Beispielen jeweils davon aus, dass `pandas` installiert und unter dem Alias `pd` importiert ist.

## 7.1 Datenstrukturen: Series und DataFrame

Pandas verwendet `DataFrame` und `Series` Objekte zur Verwaltung von Daten (siehe Abb. 7.2). `Series` ist eine eindimensional beschriftete Spalte, welche jeden Datentyp aufnehmen kann (bspw. ganze Zahlen, Zeichenketten, Gleitkommazahlen, usw.). Die Zeilenbeschriftungen werden als *Index* bezeichnet. Mit der Methode `Series` kann eine solche Spalte `s` erstellt werden:

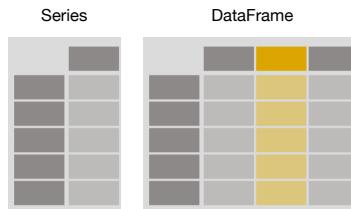


Abbildung 7.2: Series und DataFrame.

```
s = pd.Series(data)
```

wobei `data` zum Beispiel ein Wörterbuch `dict` sein kann:

```
data = {"b": 1, "a": 0, "c": 2}
s = pd.Series(data)
```

Das `Series` Objekt `s` kann man sich jetzt folgendermassen vorstellen:

```
b    1
a    0
c    2
```

Man kann der Methode `Series` einen Index in Form einer Liste als Parameter mitgeben. Wenn ein Index übergeben wird, werden die Werte in den Daten, die den Bezeichnungen im Index entsprechen, herausgezogen. Fehlende Werte erhalten den Eintrag `NaN`<sup>2</sup>.

```
d = {"b": 1, "a": 0, "c": 2}
s = pd.Series(d, index=["b", "d", "a"])
```

Das `Series` Objekt `s` kann man sich jetzt folgendermassen vorstellen (beachten Sie den fehlenden Eintrag für `c`, da dieser Schlüssel im Index nicht vorkommt):

---

<sup>2</sup>`NaN` (*not a number*) ist die Standardmarkierung für fehlende Daten in pandas.

```
b      1.0  
d      NaN  
a      0.0
```

Ein **DataFrame** ist eine zweidimensionale Datenstruktur, die Daten verschiedener Typen in mehreren Spalten speichern kann. Ein **DataFrame** ist vergleichbar mit einem Tabellenblatt einer Tabellenkalkulation.

Es gibt viele Möglichkeiten, selber ein **DataFrame** zu erstellen. Wir zeigen im Folgenden die Möglichkeit mit einem Python Wörterbuch **dict**. Hierbei entsprechen die Schlüssel im **dict** jeweils den Bezeichnungen der Spalten, und die Werte des **dict** sind jeweils Listen, welche Sie in der entsprechenden Spalte speichern wollen. Die Ausgaben von

```
df = pd.DataFrame({'col1': ['C', 'A', 'B', 'A', 'D', 'A'],  
                   'col2': [2, 8, 9, 1, 7, 4],  
                   'col3': [0, 1, 9, 4, 1, 3]})  
print(df)
```

ist demnach:

	col1	col2	col3
0	C	2	0
1	A	8	1
2	B	9	9
3	A	1	4
4	D	7	1
5	A	4	3

Beachten Sie: Da wir keinen Index angeben, werden die Zeilen mit den ganzen Zahlen 0, 1, 2, ... indexiert. Wir können auch hier einen Index manuell als liste definieren (den Index also z.B. als Parameter definieren: `index=["Z1", "Z2", "Z3", "Z4", "Z5", "Z6"]`). Alternativ machen wir aus einer der Spalten einen Index:

Abbildung 7.3: Datei mit Passagierdaten der Titanic.

```
df = df.set_index('col3')
```

Das DataFrame `df` sieht nun so aus:

	col1	col2
col3		
0	C	2
1	A	8
9	B	9
4	A	1
1	D	7
3	A	4

### 7.1.1 Daten mit pandas Lesen und Schreiben

Das Paket `pandas` stellt die Funktion `read_csv()` zur Verfügung, um eine csv-Datei in ein DataFrame einzulesen.

**Beispiel 106** Für das folgende Beispiel laden wir eine Datei `titanic.csv` (siehe Abb. 7.3) mit `pandas` ein. Die Datei `titanic.csv` enthält Angaben von 891 Passagieren der Titanic – unter anderem ist das Überleben

des Unglücks, die Reiseklasse oder der Ticketpreis erfasst:

```
titanic = pd.read_csv("titanic.csv")
```

Wenn ein `DataFrame` mit `print` ausgegeben wird, werden standardmäßig die ersten und letzten fünf Zeilen der Daten angezeigt. Um die ersten  $N$  Zeilen eines `DataFrame` anzuzeigen, verwenden wir die Methode `head( $N$ )` mit der gewünschten Anzahl Zeilen als Argument.

Die Ausgabe von `print(titanic)` bzw. `print(titanic.head(3))` lautet:

```
PassengerId  Survived  Pclass   ...   Fare Cabin Embarked
0            1         0       3    ...  7.2500   NaN      S
1            2         1       1    ...  71.2833  C85      C
2            3         1       3    ...  7.9250   NaN      S
3            4         1       1    ...  53.1000  C123     S
4            5         0       3    ...  8.0500   NaN      S
...          ...
886          887       0       2    ...  13.0000  NaN      S
887          888       1       1    ...  30.0000  B42      S
888          889       0       3    ...  23.4500  NaN      S
889          890       1       1    ...  30.0000  C148     C
890          891       0       3    ...  7.7500  NaN      Q

[891 rows x 12 columns]
PassengerId  Survived  Pclass   ...   Fare Cabin Embarked
0            1         0       3    ...  7.2500   NaN      S
1            2         1       1    ...  71.2833  C85      C
2            3         1       3    ...  7.9250   NaN      S
```

Wir können berechnete `DataFrame` Objekte mit der Methode `to_excel()` jederzeit als Excel Datei speichern (ggf. müssen Sie hierzu noch das Paket `openpyxl` installieren).

	A	B	C	
1	col1	col2	col3	
2	C	2	0	
3	A	8	1	
4	B	9	9	
5	A	1	4	
6	D	7	1	
7	A	4	3	

	A	B	C	D	
1		col1	col2	col3	
2	0	C	2	0	
3	1	A	8	1	
4	2	B	9	9	
5	3	A	1	4	
6	4	D	7	1	
7	5	A	4	3	

Abbildung 7.4: Ein DataFrame als Excel Datei gespeichert (ohne bzw. mit Index).

**Beispiel 107** In folgendem Beispiel wird das DataFrame `df` in eine Excel Datei `example.xlsx` geschrieben. Der Blattname wird als `"Sheet"` definiert und durch die Einstellung `index=False` oder `index=True` werden die Bezeichnungen der Zeilen (also der Index) entweder nicht oder doch im Blatt aufgenommen (siehe Abb. 7.4).

```
df = pd.DataFrame({'col1': ['C', 'A', 'B', 'A', 'D', 'A'],
                   'col2': [2, 8, 9, 1, 7, 4],
                   'col3': [0, 1, 9, 4, 1, 3]})

df.to_excel("example.xlsx", sheet_name="Sheet", index=False)
```

## 7.2 Daten Sortieren

Sortierung dient der Umgruppierung eines vorliegenden DataFrame. Die Tabellendaten im DataFrame werden bei einer Sortierung *nicht* verändert<sup>3</sup>. Bei einer Sortierung werden die vorliegenden Dateneinträge lediglich entsprechend den festgelegten Sortierkriterien zeilenweise umgruppiert.

Mit der Methode `sort_values` können wir die Zeilen in einem DataFrame nach einer oder mehreren Spalten sortieren. Hierzu wird der Methode `sort_values` eine Liste mit Spaltennamen als Parameter `by` mitgegeben.

<sup>3</sup>Hinweis: Wollen Sie das DataFrame `df` dauerhaft verändern, müssen Sie das Resultat der Sortierung wiederum der Variablen `df` zuweisen.

```

df = pd.DataFrame({'col1': ['C', 'A', 'B', 'A', 'D', 'A'],
                   'col2': [2, 8, 9, 1, 7, 4],
                   'col3': [0, 1, 9, 4, 1, 3]})

      col1  col2  col3
0     C     2     0
1     A     8     1
2     B     9     9
3     A     1     4
4     D     7     1
5     A     4     3

```

```

df.sort_values(by=['col1'])   df.sort_values(by=['col1', 'col2'])   df.sort_values(by='col2', ascending=False)

      col1  col2  col3      col1  col2  col3      col1  col2  col3
1     A     8     1      3     A     1     4      2     B     9     9
3     A     1     4      5     A     4     3      1     A     8     1
5     A     4     3      1     A     8     1      4     D     7     1
2     B     9     9      2     B     9     9      5     A     4     3
0     C     2     0      0     C     2     0      0     C     2     0
4     D     7     1      4     D     7     1      3     A     1     4

```

Abbildung 7.5: Zeilen in einem DataFrame auf- oder absteigend nach einer oder mehreren Spalten sortieren.

ben. Standardmässig werden die Daten aufsteigend sortiert – mit dem Parameter `ascending=False` werden die Daten absteigend sortiert.

**Beispiel 108** In Abb. 7.5 wird das DataFrame `df` dreimal umsortiert.

### 7.3 Daten Filtern

In diesem Abschnitt zeigen wir, wie man bestimmte Spalten oder Zeilen aus einem DataFrame filtern kann und wie man gleichzeitig bestimmte Zeilen und Spalten aus einem DataFrame auswählen kann (siehe Abb. 7.6).

Um eine bestimmte Spalte aus einem DataFrame auszuwählen, verwenden wir die Spaltenbezeichnung innerhalb eckiger Klammern [] (ähnlich zur Auswahl eines Wertes mit einem Schlüssel innerhalb eines Wörterbuches `dict`).

Um mehrere Spalten auszuwählen, verwenden wir eine Liste von Spaltennamen innerhalb der Auswahlklammern [] (die inneren eckigen

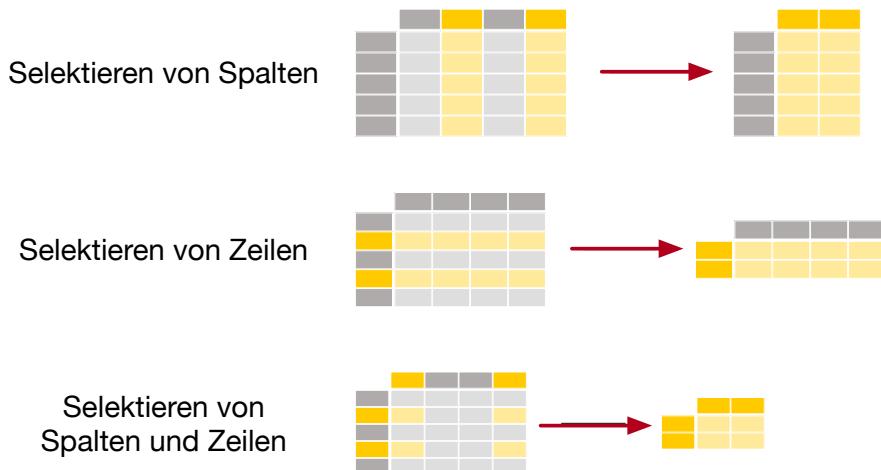


Abbildung 7.6: Selektion von Spalten und Zeilen.

Klammern definieren dabei eine Liste mit Spaltennamen, während die äusseren Klammern verwendet werden, um die Daten aus einem `DataFrame` auszuwählen).

**Beispiel 109** In folgendem Beispiel wählen wir einmal nur die Spalte `"Age"` und einmal die beiden Spalten `["Age", "Sex"]`<sup>4</sup>.

```
# Auswahl einzelner Spalten
ages = titanic["Age"] # Series Objekt
ages_sex = titanic[["Age", "Sex"]] # DataFrame Objekt
```

Wenn wir eine einzelne Spalte eines `DataFrame` auswählen, ist das Ergebnis ein Objekt vom Typ `Series`, wird mehr als eine Spalte selektiert, ist das Resultat wiederum ein `DataFrame` Objekt.

Auf einem `DataFrame` Objekt kann man nicht nur Spalten sondern auch Zeilen selektieren. Um Zeilen auszuwählen, definieren wir eine Boolesche Bedingung innerhalb der Auswahlklammern `[]`.

---

<sup>4</sup>Dieses und die weiteren Beispiele führen Beispiel 106 fort. Das Laden der Datei in ein `Dataframe` wird also nicht nochmals gezeigt.

Die Bedingung `titanic["Age"] > 35` innerhalb der Auswahlklammern prüft zum Beispiel, für welche Zeilen die Spalte "Age" einen Wert grösser als 35 hat. Wenn wir mehrere Bedingungen kombinieren möchten, muss jede Bedingung von Klammern () umgeben sein. Ferner erlaubt `pandas` die Verwendung der Wörter `or` und `and` nicht. Stattdessen müssen wir den Operator | (für `or`) und den Operator & (für `and`) verwenden.

**Beispiel 110** In folgendem Beispiel selektieren wir in `above_35` alle Passagiere, die älter sind als 35 Jahre. In `class_23` werden alle Passagiere der zweiten oder dritten Klasse selektiert.

```
# Filtern von Zeilen
above_35 = titanic[titanic["Age"] > 35]
class_23 = titanic[(titanic["Pclass"] == 2) | (titanic["Pclass"] == 3)]
```

Das Paket `pandas` erlaubt auch, eine Teilmenge von Zeilen *und* Spalten gleichzeitig zu bilden. Nehmen wir zum Beispiel an, wir interessieren uns für die Namen der Passagiere, die jünger als 10 Jahre sind. Für solche Operationen stellt `pandas` den Operator `loc` zur Verfügung, der vor die Auswahlklammern gestellt werden kann.

Bei der Verwendung von `loc` definieren wir dann in den Auswahlklammern [] vor dem Komma die Boolesche Bedingungen zur Auswahl der gewünschten Zeilen und nach dem Komma definieren wir die Spalten, die wir auswählen möchten:

```
dataframe_bezeichner.loc[Zeilenfilter, Spaltenfilter]
```

**Beispiel 111** Mit folgender Auswahl definieren wir ein `DataFrame`

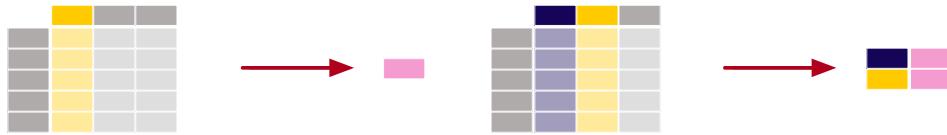


Abbildung 7.7: Zusammenfassung einzelner oder mehrerer Spalten.

*Objekt, das den Namen, das Alter und den Status **Survived** aller überlebenden Passagiere unter 10 Jahren anzeigt:*

```
# Filtern von Zeilen und Spalten
survived_below_ten = titanic.loc[(titanic["Age"] < 10) &
                                  (titanic["Survived"] == 1),
                                  ["Name", "Age", "Survived"]]

print(survived_below_ten.head(3))
```

Ausgabe:

	Name	Age	Survived
10	Sandstrom, Miss. Marguerite Rut	4.0	1
43	Laroche, Miss. Simonne Marie Anne Andree	3.0	1
58	West, Miss. Constance Mirium	5.0	1

## 7.4 Daten Zusammenfassen

Es stehen verschiedene zusammenfassende Statistiken zur Verfügung, die auf einer oder mehreren Spalten mit numerischen Daten angewendet werden können (siehe Abb. 7.7). Diese Operationen schliessen im Allgemeinen fehlende Daten aus und werden standardmässig zeilenübergreifend durchgeführt.

Wenn wir z.B. das maximale Alter der Passagiere wissen möchten, können wir dies tun, indem wir die Spalte "**Age**" auswählen und danach die Methode **max()** auf dem **Series** Objekt anwenden (weitere Methoden, die wir auf **Series** und **DataFrame** Objekten anwenden

können, sind bspw. `min`, `count`, `mean`, `median`, etc.).

Die Methode `describe()` bietet einen schnellen Überblick über die numerischen Daten in einem `DataFrame` oder `Series` Objekt (nicht numerische Daten werden von der Methode `describe()` standardmäßig nicht berücksichtigt).

**Beispiel 112** In folgendem Beispiel rufen wir die Methoden `max()` und `describe()` auf einem `Series` Objekt auf.

```
ages = titanic["Age"]
print("Max Age:", ages.max(), sep="\t")
print(ages.describe())
```

Die Ausgabe lautet:

```
Max Age:    80.0
count      714.000000
mean       29.699118
std        14.526497
min        0.420000
25%       20.125000
50%       28.000000
75%       38.000000
max       80.000000
```

Die Berechnung einer bestimmten Statistik (z.B. das Durchschnittsalter) für jede Kategorie in einer Spalte (z.B. männlich/weiblich in der Spalte Geschlecht) ist ein gängiges Muster. Die Methode `groupby` wird verwendet, um diese Art von Operationen zu unterstützen, und folgt dem Paradigma *Aufteilen-Anwenden-Kombinieren* (siehe Abb. 7.8):

- Aufteilen der Daten in Gruppen
- Unabhängige Anwenden einer Funktion auf jede Gruppe

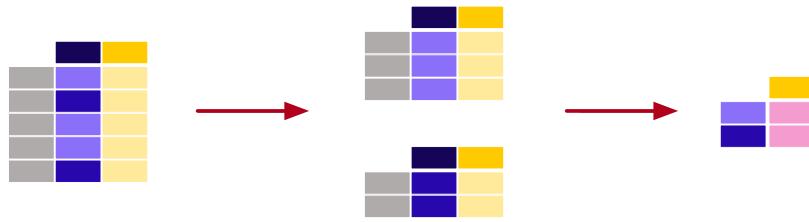


Abbildung 7.8: Das Paradigma *Aufteilen-Anwenden-Kombinieren*.

- Kombinieren der Ergebnisse in einer Datenstruktur

Die Schritte Anwenden und Kombinieren werden in `pandas` typischerweise zusammen ausgeführt.

**Beispiel 113** Zum Beispiel könnten wir an der Überlebensrate der männlichen und weiblichen Passagiere der Titanic interessiert sein. Da uns der Durchschnitt der Spalte "`Survived`" für beide Geschlechter interessiert, wird zunächst eine Unterauswahl für diese beiden Spalten getroffen (`["Sex", "Survived"]`). Anschliessend wird die Methode `groupby()` auf das Geschlecht angewendet, um eine Gruppe pro Kategorie zu bilden. Schliesslich wird mit `.mean()` der Durchschnitt der Spalten pro Geschlecht berechnet und zurückgegeben.

```
survived_m_f = titanic[["Sex", "Survived"]].groupby("Sex").mean()
print(survived_m_f)
```

Ausgabe:

	<i>Survived</i>
<i>Sex</i>	
<i>female</i>	0.742038
<i>male</i>	0.188908

Im vorherigen Beispiel haben wir zuerst explizit zwei Spalten ausgewählt. Wird dies nicht gemacht, wird der Mittelwert auf jeder Spalte berechnet, die numerische Spalten enthält. Die Anweisung

```
print(titanic.groupby("Sex").mean())
```

führt somit beispielsweise zur Ausgabe:

	PassengerId	Survived	Pclass	...	SibSp	Parch	Fare
female	431.028662	0.742038	2.159236	...	0.694268	0.649682	44.479818
male	454.147314	0.188908	2.389948	...	0.429809	0.235702	25.523893

Eine Gruppierung kann nach mehreren Spalten gleichzeitig vorgenommen werden. Hierzu geben wir die Spaltennamen als Liste an die Methode `groupby()` weiter. Die Anweisung

```
print(titanic[["Sex", "Pclass", "Survived"]].groupby(["Sex", "Pclass"]).mean())
```

führt somit beispielsweise zur Ausgabe:

		Survived
Sex	Pclass	
female	1	0.968085
	2	0.921053
	3	0.500000
male	1	0.368852
	2	0.157407
	3	0.135447

# Kapitel 8

## Daten Durchsuchen

Die Suche nach Elementen mit bestimmten Eigenschaften in einer Menge ist – ähnlich zur Sortierung von Elementen – eine häufige Aufgabe in vielen Anwendungsgebieten (z.B. Suche nach einem bestimmten Messwert in einer Datenreihe).

**Definition 1 (Suchproblem)** *Das elementare Suchproblem ist folgendermassen definiert:*

- **Eingabe:** Eine Folge von  $n$  Elementen  $\langle a_1, a_2, \dots, a_n \rangle$ , das gesuchte Element  $x$
- **Ausgabe:** Der Index  $i$ , der die Position des gesuchten Elementes  $x$  in der Folge  $\langle a_1, a_2, \dots, a_n \rangle$  angibt ( $a_i = x$ ). Es wird z.B. -1 zurückgegeben, falls  $a_i \neq x$  für alle  $i \in \{1, \dots, n\}$ .

Die Werte, die wir durchsuchen, werden als *Schlüssel* bezeichnet. Sie können davon ausgehen, dass der Schlüssel Bestandteil einer komplexeren Datenstruktur ist (z.B. die Personalnummer eines Mitarbeiters). Wir abstrahieren diese strukturierten Daten zu sogenannten *Satellitendaten* und analysieren die Suchalgorithmen auf Eingaben, die nur aus Zahlen bestehen.

## 8.1 Komplexität von Algorithmen

Heutzutage sind Rechner zwar sehr schnell, aber sie sind nicht unendlich schnell. Mit anderen Worten: Rechenzeit ist eine beschränkte Ressource, welche mit Vernunft einzusetzen ist. Die *Effizienz eines Algorithmus* beschreibt, wie sparsam dieser mit den Ressourcen umgeht, die er zur Lösung eines gegebenen Problems beansprucht.

Verschiedene Algorithmen, die zur Lösung ein und desselben Problems entwickelt werden, können sich häufig dramatisch in ihrer Effizienz unterscheiden. Diese Unterschiede können viel grösser sein, als die durch Hardware bedingten Unterschiede.

**Definition 2 (Komplexität eines Algorithmus)** *Unter Komplexität eines Algorithmus versteht man seinen maximalen Ressourcenverbrauch, der in der Regel in Abhängigkeit von der Grösse  $n$  der Eingabe angegeben wird:*

*Komplexität eines Algorithmus = Funktion  $f(n)$  der Eingabegrösse  $n$*

Bei der Komplexitätsanalyse interessiert i.a. nicht der *exakte* Aufwand, den ein Programm auf einem bestimmten Computer verursacht. Das bedeutet, für die Angabe der Komplexität eines Algorithmus spielt die verwendete Hardware oder die konkrete Implementation in einer bestimmten Programmiersprache keine Rolle. Zudem unterscheidet sich der tatsächliche Zeitbedarf verschiedener Algorithmen für das gleiche Problem für kleine Eingabegrössen oftmals nur marginal.

Bei Komplexitätsanalysen kommt es vielmehr darauf an, wie die Laufzeit wächst, wenn die Eingabegrösse  $n$  grösser wird (die sogenannte

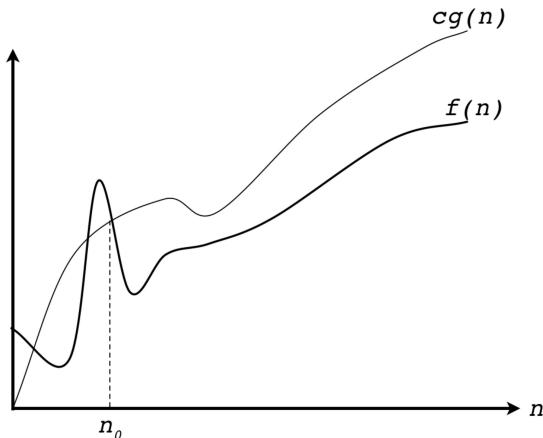


Abbildung 8.1: Für alle Eingaben grösser als  $n_0$  ist der Aufwand  $f(n)$  des Algorithmus höchstens um einen konstanten Faktor  $c$  grösser als die beschränkenden Funktion  $g$ :  $f \in O(g)$

*Wachstumsrate* der Laufzeit). Wenn also beispielsweise doppelt so viele Daten verarbeitet werden müssen, verdoppelt sich dann die Komplexität  $f(n)$  des Algorithmus oder vervierfacht sie sich sogar? Dies gibt Aufschluss über die *Skalierbarkeit eines Algorithmus*, also wie gut sich der Algorithmus auch für wachsende Probleme eignet.

Zur Angabe der Wachstumsrate eines Algorithmus verwendet man oftmals die *Landau-Notation*. Die für uns wichtigste Landau-Notation ist die *O-Notation*, die eine Angabe oberer Schranken ermöglicht<sup>1</sup>.

**Definition 3 (O-Notation)** *f und g sind Funktionen der Eingabegrösse n. Wir schreiben  $f \in O(g)$ , wenn positive Konstanten c und  $n_0$  existieren, sodass*

$$f(n) \leq cg(n)$$

*für alle  $n > n_0$ .*

---

<sup>1</sup>Es existieren auch noch die  $\Theta$  und die  $\Omega$ -Notationen, welche untere- und obere Schranken gleichzeitig ( $\Theta$ ), resp. untere Schranken ( $\Omega$ ) ermöglichen. Wir gehen hier nicht weiter auf diese Landau-Notationen ein.

Mit anderen Worten: Ist die Laufzeit  $f$  eines Algorithmus in  $O(g)$ , dann gilt folgendes: Für alle Eingabegröße  $n > n_0$  ist der Aufwand  $f(n)$  nicht wesentlich grösser (höchstens um eine konstanten Faktor  $c$ ) als die beschränkende Funktion  $g(n)$  (siehe Abbildung 8.1).

Durch die Landau Notation lassen sich Algorithmen entsprechend ihrer Komplexität in bestimmte *Komplexitätsklassen* zusammenfassen. Folgende Tabelle zeigt eine Übersicht über die wichtigsten Komplexitätsklassen.

Notation	Bedeutung	Beschreibung	Wachstum
$f \in O(1)$	beschränkt	$f$ überschreitet einen konstanten Wert nicht (unabhängig von der Grösse $n$ der Eingabe).	Sublineares Wachstum: Der Aufwand wächst langsamer als die Problemgrösse.
$f \in O(\log n)$	logarithmisch	$f$ wächst um einen konstanten Betrag, wenn sich die Eingabegrösse $n$ verdoppelt.	
$f \in O(n)$	linear	$f$ wächst auf das Doppelte, wenn sich die Eingabegrösse $n$ verdoppelt.	“Ungefähr” lineares Wachstum: Der Aufwand wächst ungefähr mit der Problemgrösse.
$f \in O(n \log n)$	log-linear	wächst ungefähr auf das Doppelte, wenn sich die Eingabegrösse $n$ verdoppelt	
$f \in O(n^2)$	quadratisch	$f$ wächst ungefähr auf das Vierfache, wenn sich die Eingabegrösse $n$ verdoppelt	Polynomiales Wachstum: Der Aufwand wächst als Potenz der Problemgrösse (analog dazu $O(n^3), O(n^4), \dots$ ).
$f \in O(2^n)$	exponentiell	$f$ wächst ungefähr auf das Doppelte, wenn sich die Eingabegrösse $n$ um eins erhöht	Exponentielles Wachstums (selbst kleine Eingaben praktisch nicht mehr lösbar)

## 8.2 Zugehörigkeitsoperatoren

Fast jede Programmiersprache hat ihre eigene Implementierung eines grundlegenden Suchalgorithmus, in der Regel als Funktion, die einen booleschen Wert von **True** oder **False** zurückgibt, je nachdem ob ein Objekt in einer bestimmten Sammlung von Objekten gefunden wird oder nicht.

In Python ist die einfachste Art, nach einem Objekt zu suchen, die Verwendung der Zugehörigkeitsoperatoren `in` und `not in`:

- `in` - Gibt `True` zurück, wenn das angegebene Element ein Teil der Sammlung ist.
- `not in` - Gibt `True` zurück, wenn das angegebene Element *nicht* Teil der Sammlung ist.

Die Operatoren `in` und `not in` können zum Beispiel auf Listen (`list`) oder Tupel (`tuple`) angewendet werden. Die Suche mit den Zugehörigkeitsoperatoren entspricht dabei einer sogenannten linearen Suche und ist deshalb nicht wirklich schnell (die Laufzeit liegt in  $O(n)$  – siehe Abschnitt 8.3). Wenden wir die Operatoren `in` und `not in` hingegen auf einer Menge (`set`) an, ist die Suche sehr schnell – die Laufzeit liegt in  $O(1)$ .

Die Laufzeit zur Überprüfung, ob ein bestimmtes Element in einem `set` enthalten ist, überschreitet also einen konstanten Wert nicht und zwar unabhängig davon, wie gross die zu Grunde liegende Menge tatsächlich ist. Wie kann das sein?

Ein `set` wird mit Hilfe von sogenannten *Hash-Tabellen* implementiert. Eine Hash-Tabelle verwendet eine *Hash-Funktion*  $f : \mathcal{E} \rightarrow [0, m-1] \subset \mathbb{N}_0$ , um die Position  $f(e)$  in der Tabelle aus dem zu speichernden Element  $e \in \mathcal{E}$  zu berechnen ( $m$  entspricht der Grösse der Tabelle). Im Idealfall ordnet die Hash-Funktion jedem Element eine eindeutige Position zu (d.h.  $f(e) \neq f(e')$  wenn  $e \neq e'$  und  $e, e' \in \mathcal{E}$ ), aber die meisten Hash-Tabellenentwürfe verwenden eine unvollkommene Hash-Funktion, die zu *Hash-Kollisionen* führen kann, wenn die

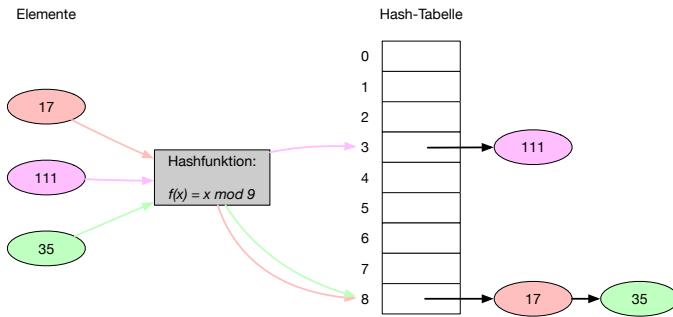


Abbildung 8.2: Drei Elemente 17, 111 und 35 werden an den Positionen  $f(17) = 8$ ,  $f(111) = 3$  und  $f(35) = 8$  in die Hash-Tabelle gelegt.

Hash-Funktion dieselbe Position für mehr als ein Element erzeugt. Solche Kollisionen werden in der Regel auf irgendeine Weise aufgelöst (z.B. werden die Elemente an der entsprechenden Position in einer Liste verwaltet).

**Beispiel 114** Folgende Funktion liefert bspw. einen Hashwert  $f(x)$  im Intervall  $[0, m - 1]$  für eine ganze Zahl  $x$  und eine Hashtabelle der Grösse  $m$ .

$$f(x) = x \bmod m$$

In Abb. 8.2 wird für die Elemente 17, 111 und 35 der Hashwert für eine Tabelle der Grösse  $m = 9$  berechnet (nämlich  $f(17) = 17 \bmod 9 = 8$ ,  $f(111) = 3$  und  $f(35) = 8$ ). Die Elemente werden dann an die entsprechenden Positionen in der Hash-Tabelle gelegt.

Bei der Zugehörigkeitsoperation muss nun im Grunde nur geprüft werden, ob sich das gesuchte Element  $x$  an der durch seinen Hashwert  $f(x)$  bestimmten Position befindet. Das heisst, die Laufzeit dieses Vorgangs ist nicht von der Grösse der Menge abhängig (bei Listen hingegen muss die gesamte Liste durchsucht werden, was mit zunehmender Grösse der Liste langsamer wird).

Zusammenfassend: Wird ein Element  $e$  mit `s.add(e)` zu einem `set`  $s$  hinzugefügt, wird die Position  $f(e)$  von  $e$  in der Hash-Tabelle berechnet. Wird nun mit  $e \text{ in } s$  überprüft, ob sich das Element  $e$  im `set`  $s$  befindet, wird wiederum der Hashwert  $f(e)$  von  $e$  berechnet und nachgeschaut, ob sich in der Hash-Tabelle an Position  $f(e)$  das Element  $e$  befindet.

Die Operatoren `in` und `not in` reichen aus, wenn es nur darum geht, herauszufinden, ob eine Element in einer gegebenen Sammlung vorhanden ist. In den meisten Fällen benötigen wir neben der Feststellung, ob ein Objekt existiert oder nicht, auch die Position des Objekts in der Sammlung – die Zugehörigkeitsoperatoren erfüllen diese Anforderung nicht.

## 8.3 Lineare Suche

Bei der *linearen Suche* wird die Datensammlung sequenziell durchsucht, also ein Element nach dem anderen, bis das gesuchte Element gefunden wird. Die lineare Suche ist zwar einfach aber relativ ineffizient. Sie hat aber den Vorteil, dass sie bei jeder Art von Datensammlung und unabhängig der Anordnung der Elemente (also auch auf unsortierten Daten) verwendet werden kann.

Die folgende Funktion `linear_search` implementiert die sequenzielle Suche nach einem Schlüssel `key` in einer Liste `lst`. Die Funktion verwendet eine `for`-Schleife zur Durchsuchung der Liste `lst`. Offensichtlich gibt das Verfahren den ersten Wert für `i` zurück, für den gilt

`lst[i]== key` (falls der gesuchte Schlüssel `key` nicht in der Datensammlung vorkommt, gibt die Funktion `-1` zurück). Wollen wir nicht nur das erste sondern *alle* Elemente in `lst` mit Schlüssel `key` finden, müsste der Algorithmus erweitert werden.

```
"""
linear_search.py
"""

def linear_search(lst, key):
    for i in range (len(lst)):
        if lst[i] == key:
            return i
    return -1

print(linear_search([1,2,3,4,5,2,1], 2)) # prints 1
```

Die Laufzeit der linearen Suche liegt in  $O(n)$ . Für Datensammlungen mit einer kleinen Anzahl Elementen ist die lineare Suche oftmals effizient genug, für grössere Datenmengen benötigen wir aber effizientere Verfahren.

## 8.4 Binäre Suche

Eine deutliche Verbesserung des Laufzeitverhaltens gegenüber der sequenziellen Suche kann durch eine *binäre Suche* erreicht werden.

Damit die binäre Suche angewendet werden kann, müssen zwei Voraussetzungen erfüllt sein:

- Die Datenelemente müssen sortiert sein.
- Es muss möglich sein, jeweils auf das “mittlere” Element der Datensammlung zuzugreifen (z.B. via Index) .

Intuitiv arbeitet die binäre Suche nach einem Schlüssel  $k$  wie folgt:

1. Teile die sortierte Folge von  $n$  Elementen in zwei Teilstufen von je  $n/2$  Elementen auf, so dass  $x$  das “mittlere” Element der Datensammlung ist (d.h. die erste Hälfte der Daten ist kleiner-gleich  $x$ , die zweite Hälfte grösser-gleich  $x$ ).
2. Das mittlere Element  $x$  kann kleiner, grösser oder gleich dem gesuchten Element  $k$  sein.
  - Ist  $x$  kleiner als das gesuchte Element  $k$ , befindet sich  $k$  in der zweiten Hälfte (falls es sich dort überhaupt befindet) und es wird dort weitergesucht.
  - Ist  $x$  hingegen grösser als  $k$ , muss nur in der ersten Hälfte weitergesucht werden (die jeweils andere Hälfte muss nicht mehr betrachtet werden).
  - Ist es gleich dem gesuchten Element ( $x = k$ ), ist die Suche beendet.

Die Länge des Suchbereiches wird mittels der binären Suche von Schritt zu Schritt halbiert. Spätestens wenn der Suchbereich auf ein Element geschrumpft ist, ist die Suche beendet: Dieses eine Element ist dann entweder das gesuchte Element, oder das gesuchte Element kommt in der Menge nicht vor.

**Beispiel 1** In Abbildung 8.3 ist die Arbeitsweise der binären Suche illustriert. Der Suchwert ist in diesem Beispiel einmal  $k = 7$  (Erfolg) und einmal  $k = 14$  (Misserfolg).

Nachfolgend ist die Idee der binären Suche in Python als Funktion `binary_search` implementiert. Eingabe des Suchverfahrens ist eine zu

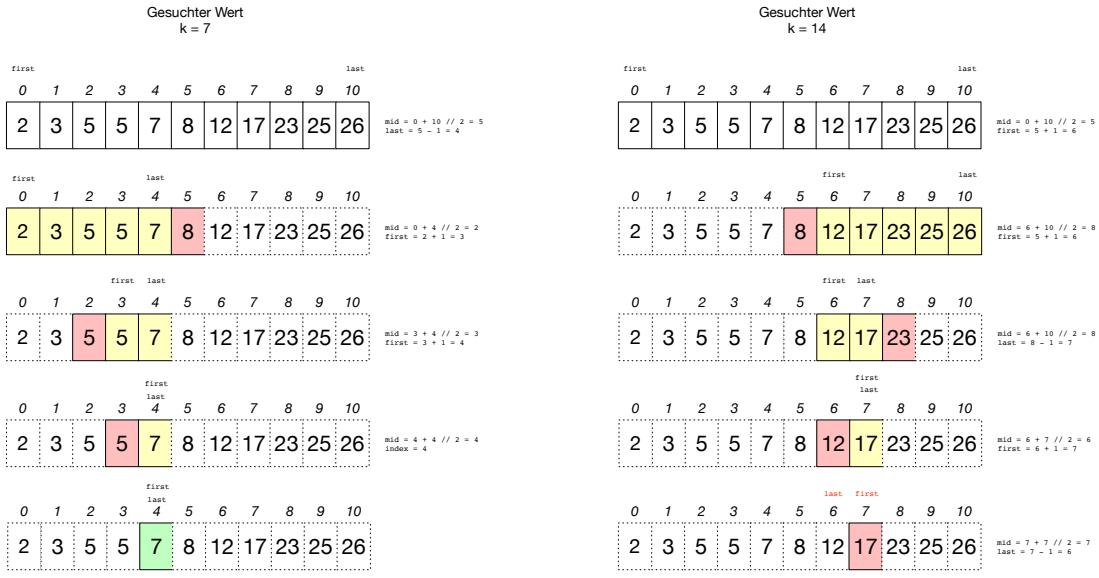


Abbildung 8.3: Die Arbeitsweise der binären Suche illustriert.

durchsuchende Liste `lst` und der gesuchte Wert `key`. Beim ersten Aufruf des Algorithmus ist `first = 0` und `last = len(lst) - 1` (entspricht also dem grössten gültigen Index in `lst`). Die Variable `index` speichert am Ende des Verfahrens die Position des gesuchten Elementes (bzw. `-1`, falls das gesuchte Element nicht vorhanden ist).

```
"""
binary_search.py
"""

def binary_search(lst, key):
    first = 0
    last = len(lst) - 1
    index = -1
    while first <= last and index == -1:
        mid = (first + last) // 2
        if lst[mid] == key:
            index = mid
        else:
            if key < lst[mid]:
                last = mid - 1
            else:
                first = mid + 1
    return index

print(binary_search([1,2,3,4,5,8,11], 2)) # prints 1
```

Zunächst überprüft die Funktion, ob der Suchbereich noch vorhanden ist bzw. ob das gesuchte Element immer noch nicht gefunden wurde (mit `first <= last` **and** `index == -1`). Solange beides wahr ist, wird jeweils die Mitte `mid` des Suchbereiches berechnet und überprüft, ob das Element `lst[mid]` dem Suchwert `key` entspricht – in diesem Fall setzen wir `index` auf den Wert `mid` und geben diesen zurück. Falls nicht, wird überprüft, ob unterhalb oder oberhalb der Position `mid` weitergesucht werden soll.

Ist die Datenmenge nicht sortiert, kann die binäre Suche nicht richtig funktionieren, da sie möglicherweise in die falsche Richtung läuft und das Element sich in der anderen Hälfte der unterteilten Sammlung befindet. Zudem garantiert die binäre Suche offensichtlich nicht, dass der kleinste Index zurückgegeben wird, an dessen Position der Suchwert gefunden werden kann.

Die Laufzeit der binären Suche ist in  $O(\log(n))$ . Es ist i.a. schneller, in einer unsortierten Sammlung sequenziell zu suchen, als zuerst die Sammlung zu sortieren (die schnellsten bekannten Sortieralgorithmen haben eine Laufzeit in  $O(n \log(n))$ ) und darin mit der binären Suche zu suchen. Wenn aber viele Suchvorgänge nacheinander in der gleichen Sammlung durchzuführen sind, lohnt sich das einmalige Sortieren der Sammlung und die anschliessende Anwendung der binären statt der linearen Suche.

# Kapitel 9

## Daten Visualisieren

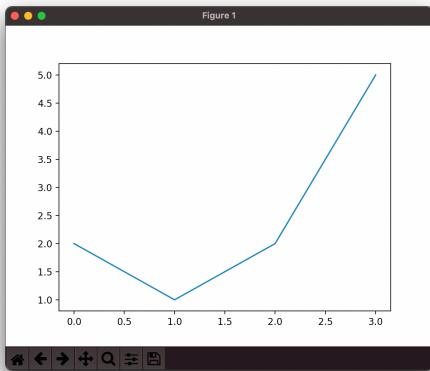
Das Paket `matplotlib` ist eine vielfältig einsetzbare Bibliothek zur Erstellung von Diagrammen und Grafiken. Das Paket `matplotlib` enthält ein Modul namens `pyplot`, das Sie installieren und importieren können. Verwenden Sie die folgende `import` Anweisung, um das Modul zu importieren und einen Alias namens `plt` zu erstellen:

```
import matplotlib.pyplot as plt
```

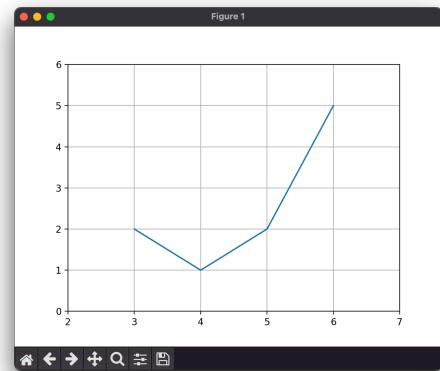
Sämtliche Beispiele in diesem Kapitel gehen davon aus, dass `plt` zur Verfügung steht.

### 9.1 Liniendiagramme

Der einfachste Weg, eine Abbildung zu erstellen, ist die Verwendung der Funktion `plt.plot(values)`, wobei `values` z.B. eine Liste mit numerischen Werten ist. Die Funktion `plot` zeigt die Reihe von Werten aus `values` auf der  $y$ -Achse als Liniendiagramm an, welches die Werte mit geraden Linien verbindet. Folgende Aufrufe erzeugen also bspw. die Grafik aus Abb. 9.1 (a) und zeigen diese an.



(a)



(b)

Abbildung 9.1: Zwei einfache Liniendiagramme.

```
plt.plot([2, 1, 2, 5])
plt.show()
```

Wenn man eine einzelne Liste angibt, geht die Funktion `plot` davon aus, dass es sich um eine Folge von  $y$ -Werten handelt und generiert die  $x$ -Werte automatisch (die  $x$ -Liste hat die gleiche Länge wie  $y$ , beginnt aber mit 0; daher sind die  $x$ -Daten im obigen Beispiel [0, 1, 2, 3]).

Man kann der Funktion `plot` auch eine zweite Liste mitgeben, wenn auch die  $x$ -Werte explizit definiert werden sollen. Zudem kann man mit den Funktionen `xlim` bzw. `ylim` die unteren und oberen Grenzen der  $x$ - und  $y$ -Achsen selber definieren und mit `grid(True)` ein Gitter anzeigen lassen. Folgende Aufrufe erzeugen also bspw. das Diagramm aus Abb. 9.1 (b).

```
plt.plot([3, 4, 5, 6], [2, 1, 2, 5])
plt.xlim(xmin=2, xmax=7) # x-Achse von 2 bis 7
plt.ylim(ymax=0, ymin=6) # y-Achse von 0 bis 6
plt.grid(True) # Gitter anzeigen
plt.show()
```

### 9.1.1 Linien und Farben

Der Funktion `plot` zeigt Daten standardmäßig mit einer durchgezogenen blauen Linie ohne Markierungen an. Mit Hilfe einer Formatzeichenfolge, welche `plot` neben den Listen `x` und `y` mitgegeben werden kann, kann die Linie formatiert werden. Eine Formatzeichenfolge besteht aus je einem Teil für die Markierung, die Linie und die Farbe:

" [Markierung] [Linie] [Farbe] "

Jede dieser drei Angaben ist optional. Wenn sie nicht angegeben werden, wird die Standardformatzeichenfolge "`-b`" verwendet (keine Markierung, durchgezogene Linie, Blau). Folgende Tabelle enthält einige mögliche Zeichen für alle drei Teile:

Markierung	Linie	Farbe
<code>"s"</code> : Quadrat	<code>"-"</code> : durchgehender Linie	<code>"b"</code> : blau
<code>"*"</code> : Stern	<code>"--"</code> : gestrichelte Linie	<code>"g"</code> : grün
<code>"D"</code> : Diamant	<code>"-.."</code> : gestrichelt-gepunktete Linie	<code>"r"</code> : rot
<code>"o"</code> : Kreis	<code>"::"</code> : gepunktete Linie	<code>"k"</code> : schwarz
<code>"^"</code> : Dreieck		

Die einzelnen Teile können beliebig zu einer Formatzeichenfolge kombiniert werden. Einige Beispiele von Formatzeichenfolgen (siehe auch Abb. 9.2):

- `"--"`: keine Markierung (Standard), gestrichelt, Blau (Standard)
- `"D-.r"`: Diamant, gestrichelt-gepunktet, Rot
- `"og"`: Kreis, keine Linie, Grün
- `"^:k"`: Dreieck, gepunktet, Schwarz

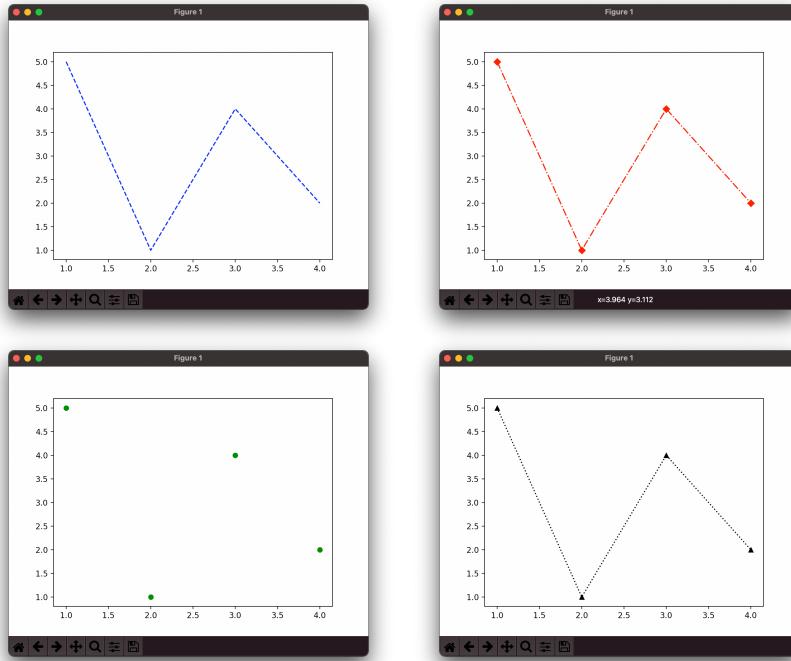


Abbildung 9.2: Von oben links nach unten rechts: `plt.plot(x, y, "--")`, `plt.plot(x, y, "D-.r")`, `plt.plot(x, y, "og")`, `plt.plot(x, y, "^:k")`.

Formatzeichenfolgen sind nur eine Abkürzung für die schnelle Einstellung grundlegender Linieneigenschaften. Man kann die Formatierung auch über die Schlüsselwortargumente `marker`, `linestyle`, `color` steuern:

```
plt.plot([1, 2, 3, 4], [5, 1, 4, 2], marker="s", linestyle="--", color="k")
```

Beachten Sie, dass man mit weiteren Argumenten viele weitere Formattierungen vornehmen kann. Z.B. kann man die Linienstärke verändern (`linewidth=2`) oder den Markierungen mit `markerfacecolor="b"` eine eigene Farbe geben.

Die Verwendung von Argumenten erlaubt es einem ferner feinere Farbdefinitionen vorzunehmen. Die Bibliothek `matplotlib` erkennt u.a. die folgenden Formate zur Angabe einer Farbe:



Abbildung 9.3: Die Palette der CSS Farbnamen.

- eine Zeichenkette mit einem einzigen Buchstaben, d.h. einer der Buchstaben '**b**', '**g**', '**r**', '**c**', '**m**', '**y**', '**k**', '**w**', die Kurzbezeichnungen für Blau, Grün, Rot, Cyan, Magenta, Gelb, Schwarz und Weiss
- ein RGB-Tupel von Werten aus dem Bereich [0, 1] (z.B. (0.1, 0.2, 0.5)). Dies definiert den Farbraum, der mittels der Farben Rot, Grün und Blau gebildet wird.
- eine der sogenannten Tableau-Farben: '**tab:blue**', '**tab:orange**', '**tab:green**', '**tab:red**', '**tab:purple**', '**tab:brown**', '**tab:pink**', '**tab:gray**', '**tab:olive**', '**tab:cyan**'
- ein CSS Farbname, (z.B. '**aquamarine**' oder '**mediumseagreen**' – siehe Abb. 9.3)

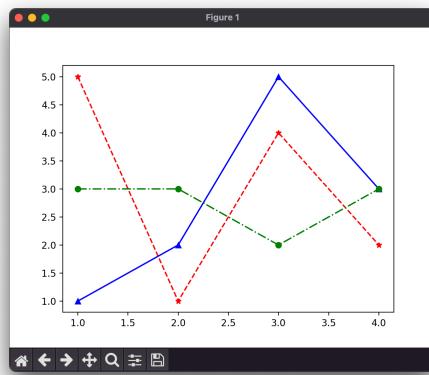


Abbildung 9.4: Drei Datenreihen in einer Abbildung gezeigt.

### 9.1.2 Mehrere Datenreihen

Oftmals möchte man mehrere Datenreihen gleichzeitig anzeigen. Hierzu gibt es unterschiedliche Möglichkeiten – zwei davon werden nachfolgend gezeigt.

Wir können die Funktion `plot` beliebig oft aufrufen (mit unterschiedlichen Daten und Argumenten) – wenn dann mit `show()` das Diagramm angezeigt wird, vereint die Abbildung alle vorher definierten Diagramme in einer Abbildung.

Folgendes Code-Fragment erzeugt bspw. Abb. 9.4.

```
plt.plot([1, 2, 3, 4], [5, 1, 4, 2], "*r--")
plt.plot([1, 2, 3, 4], [1, 2, 5, 3], "^b-")
plt.plot([1, 2, 3, 4], [3, 3, 2, 3], "og-.")
plt.show()
```

Mit Hilfe der Funktion `subplot` können wir separate Diagramme in einer Abbildung platzieren. Die einzelnen Diagramme werden dabei in Zeilen und Spalten angeordnet und der `subplot` Aufruf erwartet drei Argumente:

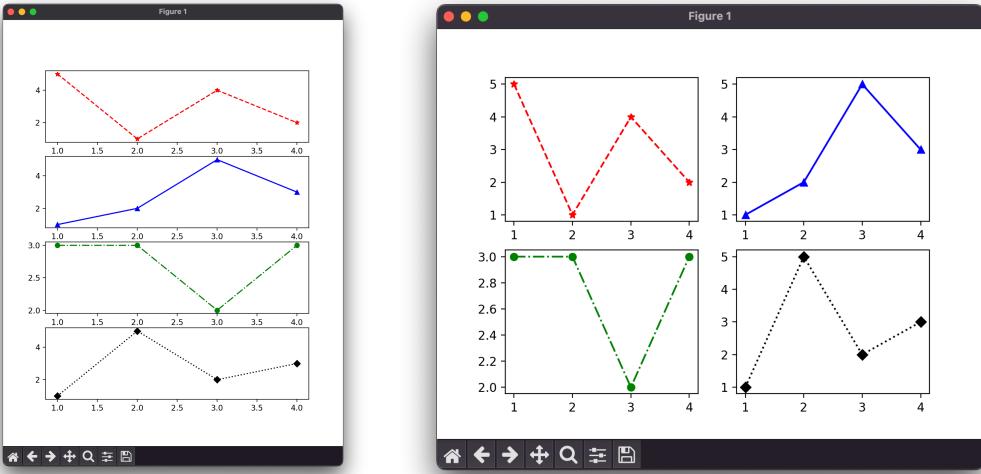


Abbildung 9.5: Vier Diagramme mit Hilfe der Funktion `subplot` in einer Abbildung platziert.

- `numrows` (Anzahl Zeilen)
- `numcols` (Anzahl Spalten)
- `plot_number` (Subplot Nummer)

wobei `plot_number` von 1 bis `numrows*numcols` reichen darf.

Folgendes Code-Fragment erzeugt bspw. die linke Seite von Abb. 9.5.  
Wir ordnen die drei Diagramme in 4 Zeilen, und 1 Spalte an .

```
plt.subplot(4,1,1)
plt.plot([1, 2, 3, 4], [5, 1, 4, 2], "*r--")
plt.subplot(4,1,2)
plt.plot([1, 2, 3, 4], [1, 2, 5, 3], "^b-")
plt.subplot(4,1,3)
plt.plot([1, 2, 3, 4], [3, 3, 2, 3], "og-.")
plt.subplot(4,1,4)
plt.plot([1, 2, 3, 4], [1, 5, 2, 3], "D:k")
plt.show()
```

Alternativ können wir z.B. mit `plt.subplot(2,2,1)` auch zwei Zeilen und zwei Spalten definieren (siehe rechte Seite der Abb. 9.5).

### 9.1.3 Text

Es kann hilfreich sein, an bestimmten Stellen eines Diagrammes Text einzufügen. Einige wichtige Text-Funktionen des Moduls `pyplot` sind:

- `plt.title("Titel")`  
Beschriftet das Diagramm mit einem Titel,
- `plt.xlabel("Beschriftung x-Achse")`  
`plt.ylabel("Beschriftung y-Achse")`  
Beschriftet die  $x$ - und  $y$ -Achsen.
- `plt.xticks(tick_list, name_list):`  
Passt die Beschriftung der einzelnen Markierungen an. Das erste Argument `tick_list` ist eine Liste von Positionen und das zweite Argument ist eine Liste von Bezeichnern, die an den angegebenen Positionen angezeigt werden sollen. Analog dazu existiert die Funktion `plt.yticks(tick_list, name_list)`.
- `plt.text(x, y, "text"):`  
Zeigt den Text an der Position  $(x, y)$  an.
- `plt.legend():` Zeigt das Label der gezeigten Diagramme an (hierzu muss in der Funktion `plot` das Argument `label="Label"` angegeben werden).

Man kann jeder Textfunktion (also z.B. `text` oder `title`) die Argumente `fontsize`, `color` und `fontname` mitgeben um die Grösse, Farbe und Schriftart des Textes anzupassen:

```
plt.title("Titel", fontsize=10, color='tab:red', fontname="Comic Sans MS")
```

**Beispiel 115** Das folgende Programm erzeugt das Diagramm in Abb. 9.6 (inkl. Titel, Achsen- und Markierungsbeschriftungen und Legende):

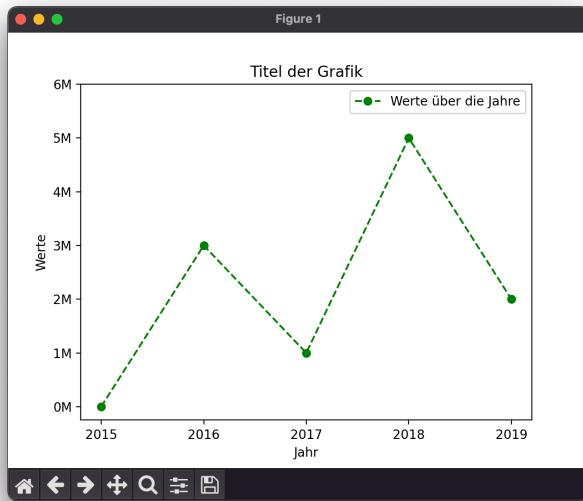


Abbildung 9.6: Ein Liniendiagramm mit eingebauten Texten (Titel, Achsen- und Markierungsbeschriftungen und Legende).

```
x_values = [0, 1, 2, 3, 4]
y_values = [0, 3, 1, 5, 2]

plt.plot(x_values, y_values, "o--g", label="Werte über die Jahre")

plt.title("Titel der Grafik")
plt.xlabel("Jahr")
plt.ylabel("Werte")
plt.xticks(list(range(5)),
           ["2015", "2016", "2017", "2018", "2019"])
plt.yticks(list(range(7)),
           ["0M", "1M", "2M", "3M", "4M", "5M", "6M"])
plt.legend() # Legende anzeigen

plt.show()
```

Eine häufige Anwendung für Text ist die Beschriftung eines Merkmals innerhalb des Diagrammes – die Funktion `annotate` ermöglicht dies mit einer Anmerkung. Bei einer Anmerkung sind zwei Punkte zu berücksichtigen: die Position, die durch das Argument `xy` dargestellt wird, und die Position des Textes `xytext` (beides (`x`, `y`) Tupel).

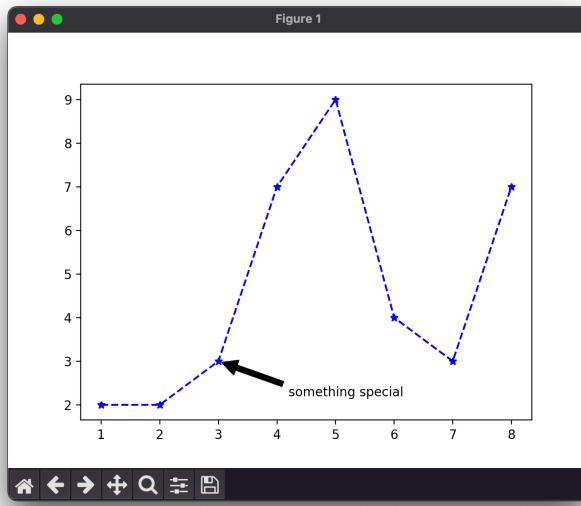


Abbildung 9.7: Die Funktion `annotate` demonstriert.

Folgendes Code-Fragment platziert den Text "`something special`" an Position `(4.2, 2.2)`. Ein schwarzer Pfeil zeigt dann vom Text auf die Position `(3,3)` (siehe Abb. 9.7).

```
plt.plot([1, 2, 3, 4, 5, 6, 7, 8], [2, 2, 3, 7, 9, 4, 3, 7], "*b--")
plt.annotate('something special', xy=(3, 3), xytext=(4.2, 2.2),
            arrowprops=dict(facecolor='k'))
```

## 9.2 Weitere Diagramme

Das Paket `matplotlib` kann selbstverständlich nicht nur Liniendiagramme erstellen. Im Folgenden wird eine kleine Auswahl an weiteren Möglichkeiten vorgestellt.

- Balkendiagramme stellen Daten mit Hilfe von vertikalen Säulen dar, so dass einzelne Werte leicht verglichen werden können. Die Funktion `bar` nimmt drei Listen entgegen:

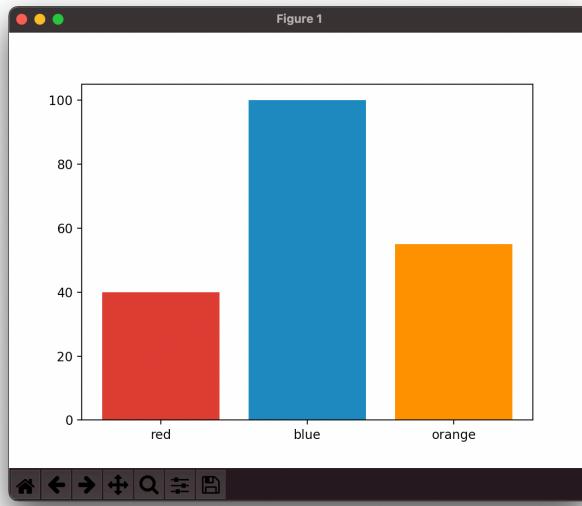


Abbildung 9.8: Ein Balkendiagramm.

- Kategorien
- Werte
- Farben

Danach wird für jede Kategorie eine Säule mit der Höhe entsprechend dem jeweiligen Wert und der jeweiligen Farbe gezeichnet (siehe bspw. Abb. 9.8):

```
categories = ['red', 'blue', 'orange']
counts = [40, 100, 55]
bar_colors = ["tab:red", "tab:blue", "tab:orange"]

plt.bar(categories, counts, color=bar_colors)
plt.show()
```

- Ein Streudiagramm zeigt eine Variable auf der vertikalen Achse und eine andere Variable auf der horizontalen Achse an. Mit der Funktion `scatter` wird jedes Datenelement als einzelner Punkt im Diagramm dargestellt (siehe bspw. Abb. 9.9 links):

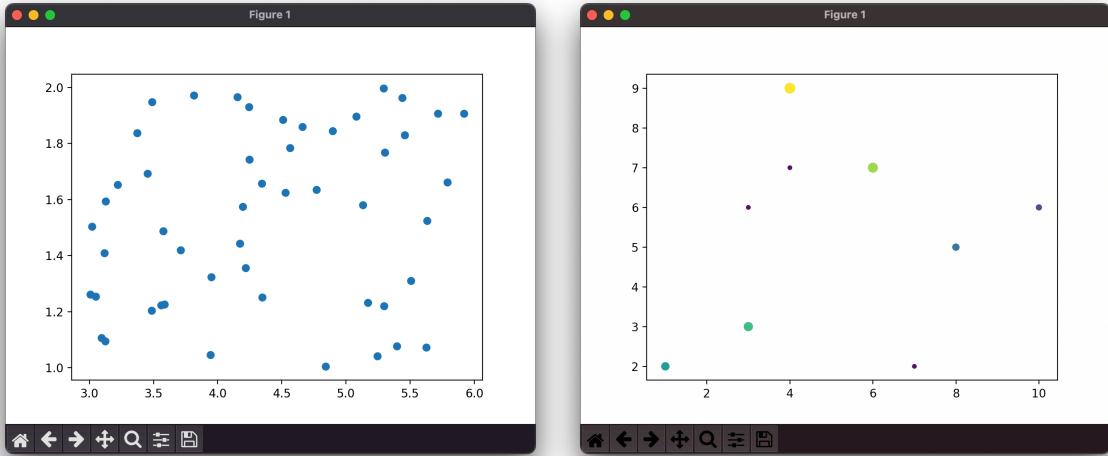


Abbildung 9.9: Zwei Streudiagramme.

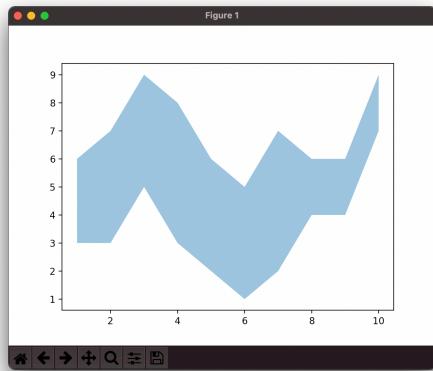
```
plt.scatter(x, y) # x und y sind Listen mit numerischen Werten
plt.show()
```

Wir können in einem Streudiagramm zusätzlich zwei weitere Datenreihen visualisieren: Mit der Grösse und der Farbe der Punkte (siehe bspw. Abb. 9.9 rechts):

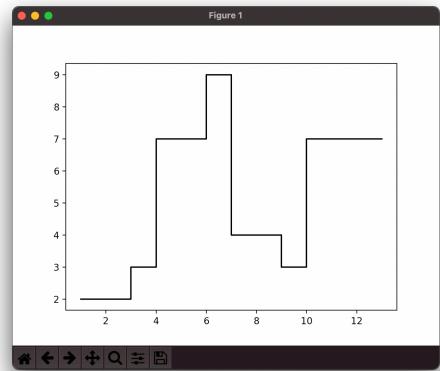
```
x = [3, 4, 7, 8, 10, 8, 1, 3, 6, 4]
y = [6, 7, 2, 5, 6, 5, 2, 3, 7, 9]
colors = [1, 1, 1, 2, 2, 3, 4, 5, 6, 7]
sizes = [10, 10, 10, 20, 20, 30, 40, 50, 60, 70]

plt.scatter(x, y, c=colors, s=sizes)
plt.show()
```

- Der Funktion `fill_between` können drei Wertebereiche `x`, `y1`, `y2` in Form von Listen mitgegeben werden. Es werden dann zwei Linendiagramme erstellt (einmal mit `y1` und einmal mit `y2`) und der Bereich zwischen den Linien wird gefüllt angezeigt (achten Sie den Parameter `alpha=.5` welche die Füllung zu 50% Durchsichtig macht – siehe Abb. 9.10 (a)):



(a)



(b)

Abbildung 9.10: (a) Mit der Funktion `fill_between` den Bereich zwischen zwei Liniendiagrammen füllen. (b) Erzeugtes Diagramm einer Schrittfunktion mit `step`

```
x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
y1 = [6, 7, 9, 8, 6, 5, 7, 6, 6, 9]
y2 = [3, 3, 5, 3, 2, 1, 2, 4, 4, 7]

plt.fill_between(x, y1, y2, alpha=.5)
plt.show()
```

- Mit der Funktion `step` werden zwei aufeinanderfolgende Datenpunkte  $(x_i, y_i)$  und  $(x_{i+1}, y_{i+1})$  nicht wie bei einem Liniendiagramm mit einer Strecke verbunden. Hier bleiben die  $y_i$  Werte konstant, bis der neue  $x_{i+1}$ -Wert erreicht wird, wo ein vertikaler Schritt zum Wert  $y_{i+1}$  stattfindet (siehe Abb. 9.10 (b)):

```
plt.step([1, 3, 4, 6, 7, 9, 10, 13], [2, 2, 3, 7, 9, 4, 3, 7], "k-")
plt.show()
```

- Mit der Funktion `imshow` können numerische Werte aus zweidimensionalen Tabellen farblich visualisiert werden. Jeder Wert der Tabelle wird mit einer farblich abgestuften Zelle dargestellt. Die resultierende Grafik nennt man auch *Heat-Map* (siehe Abb. 9.11):

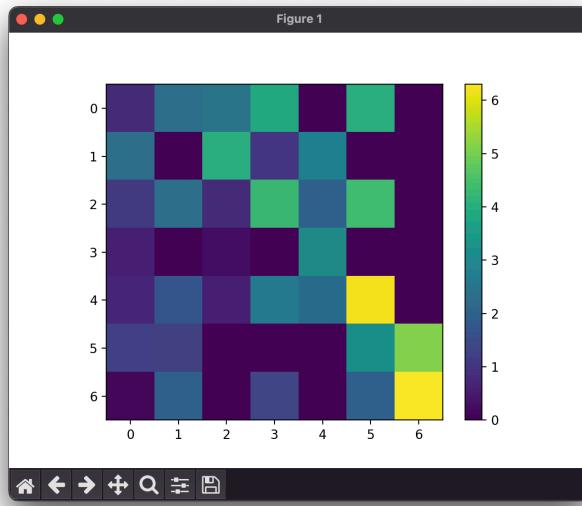


Abbildung 9.11: Erzeugte Heat-Map mit der Funktion `imshow`.

```

values = [[0.8, 2.4, 2.5, 3.9, 0.0, 4.0, 0.0],
          [2.4, 0.0, 4.0, 1.0, 2.7, 0.0, 0.0],
          [1.1, 2.4, 0.8, 4.3, 1.9, 4.4, 0.0],
          [0.6, 0.0, 0.3, 0.0, 3.1, 0.0, 0.0],
          [0.7, 1.7, 0.6, 2.6, 2.2, 6.2, 0.0],
          [1.3, 1.2, 0.0, 0.0, 0.0, 3.2, 5.1],
          [0.1, 2.0, 0.0, 1.4, 0.0, 1.9, 6.3]]

plt.imshow(values)
plt.colorbar() # zeigt die Legende der Farben an
plt.show()

```

- Ein Histogramm ist eine grafische Darstellung der Häufigkeitsverteilung und erfordert die Einteilung der Daten in Klassen (engl. *bins*). Es werden direkt nebeneinanderliegende Rechtecke von der Breite der jeweiligen Klasse gezeichnet, deren Höhe die (relative oder absolute) Häufigkeit darstellt. Die Funktion `hist` erwartet neben den Daten mindestens die Anzahl Klassen `bins` als Parameter:

```

plt.hist(x, bins=4)
plt.show()

```

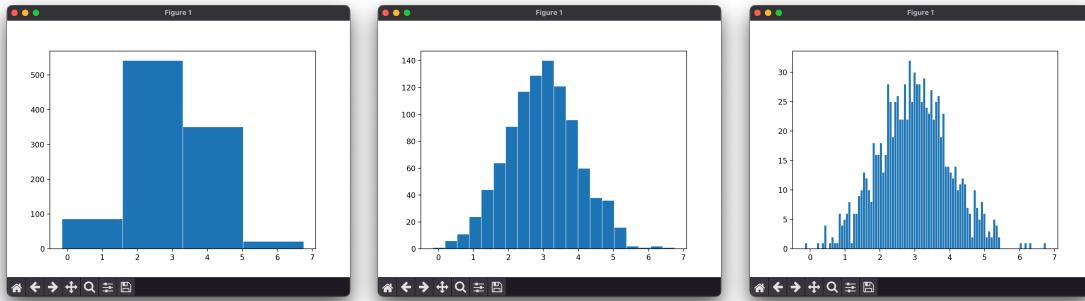


Abbildung 9.12: Histogramme mit 4, 20 bzw. 100 Klassen.

In Abb. 9.12 sind die gleichen Daten mit drei Histogrammen gezeigt, welche 4, 20 bzw. 100 Klassen verwendet.

- Ein Boxplot visualisiert verschiedene *k*-*Quantile*, wobei *k* im Bereich von  $[0, 1]$  liegt. Das 0.4-Quantil bedeutet z.B., dass 40% der Werte kleiner oder gleich dem berechneten Quantil sind, ein Quantil, das mit  $k = 0.9$  berechnet wird, bedeutet, dass 90% der Werte kleiner oder gleich dem berechneten Quantil sind.

Der *Median* ist das 0.5-Quantil und teilt die Datenreihe in zwei Hälften: Eine Hälfte ist kleiner als der Median, die andere grösser als der Median. Als *Quartile* werden die beiden Quantile mit  $k = 0.25$  und  $k = 0.75$  bezeichnet. Dabei heisst das 0.25-Quantil das untere Quartil und das 0.75-Quantil das obere Quartil. Zwischen oberem und unterem Quartil liegt die Hälfte der Daten, unterhalb des unteren Quartils und oberhalb des oberen Quartils jeweils ein Viertel der Datenwerte.

Auf Basis der Quartile wird der *Interquartilsabstand* (IQR) definiert. Der IQR ist die Differenz zwischen dem oberen und dem unteren Quartil und zeigt an, wie breit das Intervall ist, in dem

die mittleren 50% der Datenwerte liegen.

Ein Boxplot besteht in der Regel aus zwei Teilen, einer *Box* und einer Reihe von sogenannten *Whiskern*. Die Box erstreckt sich vom unteren zum oberen Quartil und beinhaltet eine horizontale Linie, um den Median zu kennzeichnen. Die Whisker enden an einem vorhandenen Datenpunkt und können auf verschiedene Weise definiert werden.

- Bei der einfachsten Methode ist die Grenze des unteren Whiskers der Minimalwert des Datensatzes und die Grenze des oberen Whiskers ist der Maximalwert des Datensatzes.
- Eine weitere Wahl für die Grenzen der Whisker basiert auf dem 1.5 IQR-Wert. Oberhalb des oberen Quartils wird ein Abstand vom 1.5-fachen des IQR gemessen, und ein Whisker wird bis zum grössten beobachteten Datenpunkt des Datensatzes gezogen, der innerhalb dieses Abstands liegt. In ähnlicher Weise wird ein Abstand vom 1.5-fachen des IQR unterhalb des unteren Quartils gemessen und ein Whisker nach unten zum niedrigsten beobachteten Datenpunkt aus dem Datensatz gezogen, der innerhalb dieses Abstands liegt. Da die Whisker an einem beobachteten Datenpunkt enden müssen, können die Whiskerlängen ungleich aussehen, obwohl 1.5 IQR für beide Seiten gleich ist. Alle anderen beobachteten Datenpunkte, die ausserhalb der Whisker-Grenze liegen, werden als Ausreisser gezeichnet.

Mit der Funktion `boxplot` können die Werte einzelner Listen oder mehrere Spalten einer zweidimensionalen Liste gleichzeitig als Boxplot dargestellt werden (siehe auch Abb. 9.13). Beachten

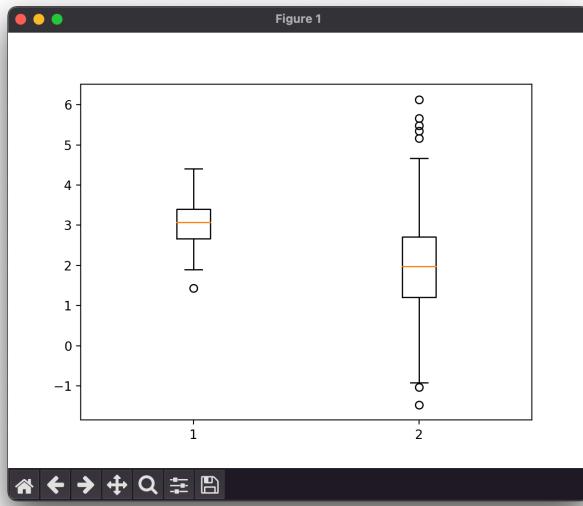


Abbildung 9.13: Zwei Boxplots zur Visualisierung der Quartile, des Medians, des kleinsten und grössten beobachteten Datenpunktes, die innerhalb des 1.5-IQRs liegen und von Ausreisern.

Sie, dass Sie mit dem optionalen Argument `showfliers=False` die Ausreisser verbergen können und mit `whis=1.5` können Sie die Grösse der Whisker beeinflussen.

```
# data ist eine zweidimensionale Liste mit zwei Spalten
plt.boxplot(data, showfliers=True, whis=1.5)
plt.show()
```

- Das Paket `matplotlib` bietet verschiedene Möglichkeiten zur Darstellung von einem Wert  $z$  in Abhängigkeit von zwei anderen Werten  $x$  und  $y$  (z.B. Messergebnisse in Abhängigkeit zweier Parameter). In Abb. 9.14 sind Beispieldiagramme für die Funktionen `contour`, `contourf`, `plot_wireframe` und `plot_surface` gezeigt<sup>1</sup>:

---

<sup>1</sup>Es gibt im Wesentlichen zwei Stile, `matplotlib` zu verwenden:

- *Funktionsstil*: Verwendung der `pyplot`-Funktionen (in diesem Kapitel angewendet).
- *Objektorientierter Stil*: Aufruf von Methoden auf `Figure` und `Achsen`-Objekten

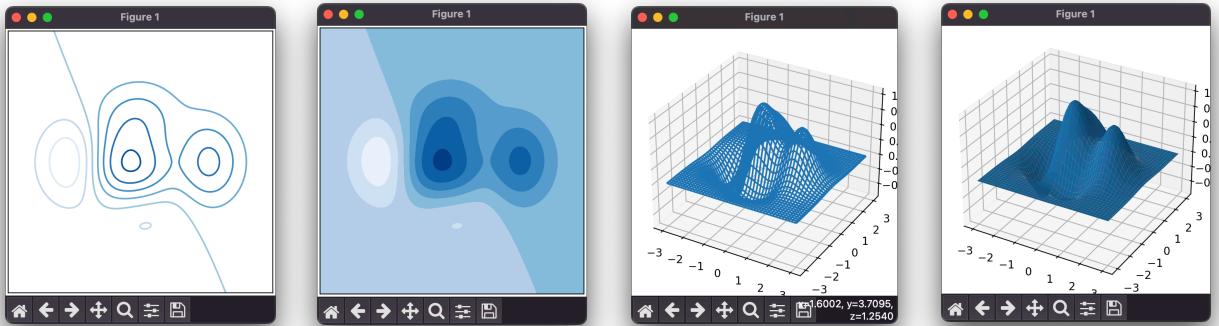


Abbildung 9.14: Verschiedene Möglichkeiten zur Darstellung von einem Wert  $z$  in Abhängigkeit von zwei anderen Werten  $x$  und  $y$ .

```
plt.contour(X, Y, Z)
plt.show()
```

```
plt.contourf(X, Y, Z)
plt.show()
```

```
fig = plt.figure()
ax = plt.axes(projection ='3d')
ax.plot_wireframe(X, Y, Z)
plt.show()
```

```
fig = plt.figure()
ax = plt.axes(projection ='3d')
ax.plot_surface(X, Y, Z)
plt.show()
```

---

Die Aufrufe `plot_wireframe` und `plot_surface` sind nur als Methodenaufrufe auf Achsen-Objekten im objektorientierten Stil möglich. Details werden hier nicht weiter erörtert.

# Kapitel 10

## Daten Vergleichen

Berechnungen von Ähnlichkeiten oder Distanzen zwischen Datenobjekten ist eine wichtige Aufgabe in vielen Anwendungen. In diesem Kapitel besprechen wir exemplarisch vier Ähnlichkeits- und Distanzmasse – zwei dieser Masse sind auf numerische Vektoren anwendbar, eines auf Mengen und eines auf Zeichenketten (siehe Abb. 10.1):

- Minkowski Distanzen
- Kosinus Ähnlichkeit
- Jaccard Ähnlichkeit
- Levenshtein Distanz

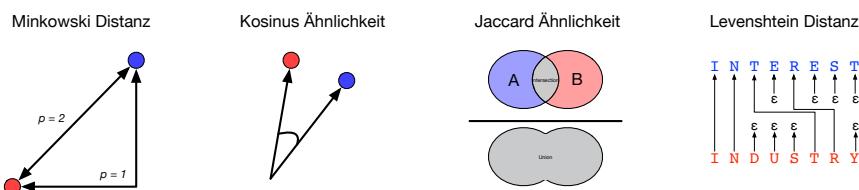


Abbildung 10.1: Illustration verschiedener Distanz- und Ähnlichkeitsmasse.

## 10.1 Minkowski Distanzen

In diesem Abschnitt gehen wir davon aus, dass zwei Datenobjekte  $\mathbf{x} = (x_1, x_2, \dots, x_n)$  und  $\mathbf{y} = (y_1, y_2, \dots, y_n)$  jeweils durch  $n$  numerische Merkmale  $x_i \in \mathbb{R}$  (bzw.  $y_i \in \mathbb{R}$ ) beschrieben sind. In Python könnten diese Datenobjekte bspw. mit Listen definiert werden:

```
x = [0, 2, 3, 4]
y = [2, 4, 3, 7]
```

Das wohl bekannteste Distanzmaß  $d(\mathbf{x}, \mathbf{y})$  für numerische Vektoren  $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$  ist die *Euklidische Distanz*. Die Distanz zwischen  $\mathbf{x}$  und  $\mathbf{y}$  nach Euklid ist gegeben durch

$$d_2(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2} = \sqrt{(x_1 - y_1)^2 + \dots + (x_n - y_n)^2}$$

Die *Minkowski Distanz*  $d_r$  generalisiert die Euklidische Distanz  $d_2$ :

$$d_r(\mathbf{x}, \mathbf{y}) = \left( \sum_{i=1}^n |x_i - y_i|^r \right)^{1/r}$$

wobei  $r$  ein freier Parameter ist. Die folgenden drei Beispiele sind die bekanntesten Instanzen von Minkowski Distanzen:

- $r = 1$  ( $L_1$  Norm): Die Distanz zwischen  $\mathbf{x}$  und  $\mathbf{y}$  nach der  $L_1$  Norm – auch *Manhattan Metrik* genannt – ist gegeben durch

$$d_1(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^n |x_i - y_i|$$

Die sog. *Hamming Distanz* zwischen binären Datenobjekten  $\mathbf{x}$  und  $\mathbf{y}$  ist ein typisches Beispiel einer Manhattan Metrik. Die

Hamming Distanz misst die Anzahl Bits die in  $\mathbf{x}$  und  $\mathbf{y}$  *nicht* übereinstimmen.

- $r = 2$  ( $L_2$  Norm): Euklidische Distanz  $d_2$  (siehe oben)
- $r = \infty$  ( $L_{\max}$  Norm): Die Distanz zwischen  $\mathbf{x}$  und  $\mathbf{y}$  nach der sogenannten *Maximum Metrik* ist gegeben durch

$$d_{\infty}(\mathbf{x}, \mathbf{y}) = \max_{i=1, \dots, n} \{|x_i - y_i|\}$$

In folgendem Python Programm definieren wir eine Funktion, welche die Minkowski Distanz  $d_r$  zweier Listen berechnet und zurückgibt:

```
import math

def minkowski_distance(x, y, r):
    d = 0
    for i in range(len(x)):
        d += math.fabs(x[i] - y[i]) ** r
    d = math.pow(d, 1/r)
    return d

vector1 = [0, 2, 3, 4]
vector2 = [2, 4, 3, 7]

print(minkowski_distance(vector1, vector2, 2))
```

Die Euklidische Distanz ist ein gängiges Distanzmaß. Zwei Dinge sollten allerdings berücksichtigt werden:

- Die Euklidische Distanz ist nicht skaleninvariant, was bedeutet, dass die berechneten Abstände je nach den Einheiten der einzelnen Merkmale verzerrt sein können. Normalerweise muss man die Daten normieren, bevor man dieses Abstandsmass verwenden kann.

Eine gängige Methode zur Normierung besteht darin, jedes Merkmal so zu transformieren, dass dessen Mittelwert und Standardabweichung 0 bzw. 1 wird (auch bekannt als *Z-Normierung*).

Gegeben sei ein Datensatz

$$X = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_N\} \text{ mit Datenobjekten } \boldsymbol{x}_i = (x_{i_1}, \dots, x_{i_n})$$

Mittelwert  $m_j$  und Standardabweichung  $\sigma_j$  des  $j$ -ten Merkmals sind

$$m_j = \frac{1}{N} \sum_{i=1}^N x_{i_j} \quad \sigma_j = \left[ \frac{1}{N-1} \sum_{i=1}^N (x_{i_j} - m_j)^2 \right]^{\frac{1}{2}}$$

Die *Z-Normierung* führt eine Translation und Skalierung wie folgt durch:

$$\hat{x}_{i_j} = \frac{x_{i_j} - m_j}{\sigma_j}$$

Dabei ist  $\hat{x}_{i_j}$  der Wert des  $j$ -ten Merkmals des  $i$ -ten Datenobjektes nach der Normierung.

- Die Euklidische Distanz ist mit zunehmender Dimensionalität der Daten immer weniger aussagekräftig. Dies hat mit dem sogenannten "Fluch der Dimensionalität" zu tun, der sich darauf bezieht, dass sich ein höherdimensionaler Raum nicht so verhält, wie wir es intuitiv von einem 2- oder 3-dimensionalen Raum kennen.

## 10.2 Kosinus Ähnlichkeit

Die Kosinus Ähnlichkeit  $s_{cos}$ , die als nächstes definiert wird, ist ein gebräuchliches Mass zur Berechnung der Ähnlichkeit numerischer Datenobjekte  $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ :

$$s_{cos} = \frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\|\mathbf{x}\| \|\mathbf{y}\|}$$

wobei

- $\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=1}^n x_i y_i$  das Skalarprodukt von  $\mathbf{x}$  und  $\mathbf{y}$  ist und
- $\|\mathbf{x}\| = \sqrt{\langle \mathbf{x}, \mathbf{x} \rangle} = \sqrt{\sum_{i=1}^n x_i^2}$  die Länge (oder Norm) des Vektors  $\mathbf{x}$  ist.

**Beispiel 2** Nehmen wir an:

$$\mathbf{x} = (3, 2, 0, 5, 0, 0, 0, 2, 0, 0)$$

$$\mathbf{y} = (1, 0, 0, 0, 0, 0, 0, 1, 0, 2)$$

$$\langle \mathbf{x}, \mathbf{y} \rangle = 3 \cdot 1 + 2 \cdot 1 = 5$$

$$\|\mathbf{x}\| = \sqrt{3^2 + 2^2 + 5^2 + 2^2} = 6.48$$

$$\|\mathbf{y}\| = \sqrt{1^2 + 1^2 + 2^2} = 2.45$$

$$s_{cos} = \frac{5}{6.48 \cdot 2.45} = 0.31$$

Wie Abb. 10.2 illustriert, ist die Kosinus Ähnlichkeit abhängig vom Winkel, der von den Vektoren  $\mathbf{x}$  und  $\mathbf{y}$  aufgespannt wird.

- Wenn der Winkel zwischen  $\mathbf{x}$  und  $\mathbf{y}$  z.B.  $0^\circ$  beträgt ( $\mathbf{x}$  und  $\mathbf{y}$  sind bis auf die Länge gleich), beträgt die Kosinus Ähnlichkeit 1.

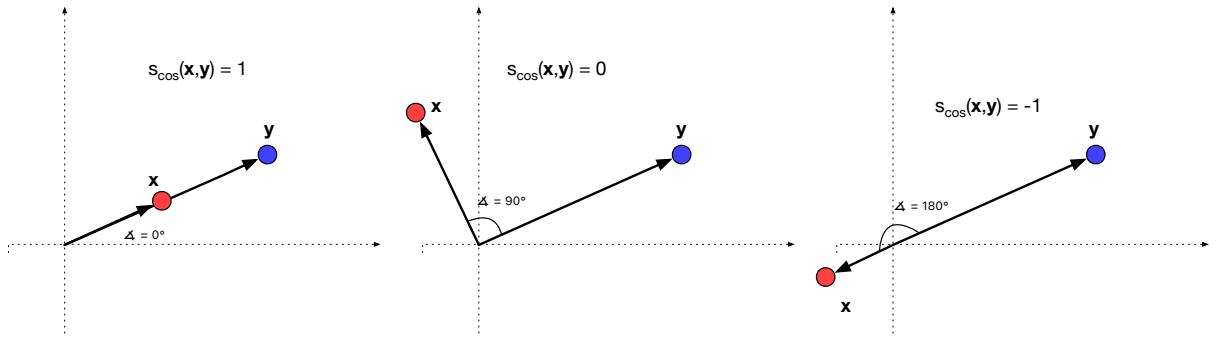


Abbildung 10.2: Illustration der Kosinus Ähnlichkeit.

- Wenn der Winkel zwischen  $x$  und  $y$  z.B.  $90^\circ$  bzw.  $180^\circ$  beträgt, dann ist die Kosinus Ähnlichkeit 0 bzw.  $-1$ .

In folgendem Programm ist die Kosinus Ähnlichkeit in einer Funktion definiert. Beachten Sie, dass in diesem Beispiel das externe Paket numpy verwendet wird zur Berechnung des Skalarproduktes und der Norm von  $x$  und  $y$  (mit der Funktion `np.dot(x, y)`).

```
import numpy as np

def cosine_similarity(x, y):
    return np.dot(x, y) / (np.sqrt(np.dot(x, x)) * np.sqrt(np.dot(y, y)))

vector1 = [0, 2, 3, 4]
vector2 = [2, 4, 3, 7]

print(cosine_similarity(vector1, vector2))
```

Die Kosinus Ähnlichkeit wird dann verwendet, wenn die Grösse der Vektoren keine Rolle spielen soll. Dies ist z.B. beim Vergleich von Texten der Fall, die durch die Anzahl der Wörter dargestellt werden. Wenn ein Wort (z.B. Wissenschaft) in Dokument  $d_1$  häufiger vorkommt als in Dokument  $d_2$ , könnte man davon ausgehen, dass  $d_1$  einen grösseren Bezug zum Thema Wissenschaft hat. Es könnte aber auch sein, dass das Thema Wissenschaft in  $d_1$  nur deshalb häufiger vorkommt, weil es viel länger ist als  $d_2$ . Die Kosinus Ähnlichkeit berücksichtigt das.

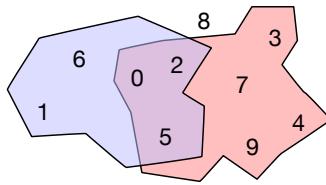


Abbildung 10.3: Illustration der Jaccard Ähnlichkeit.

### 10.3 Jaccard Ähnlichkeit

Die Jaccard Ähnlichkeit ist ein Mass zur Berechnung der Ähnlichkeit zweier Mengen  $A$  und  $B$ . Sie ist definiert als die Grösse der Schnittmenge von  $A$  und  $B$  geteilt durch die Grösse der Vereinigung der Mengen  $A$  und  $B$ .

$$s_j(A, B) = \frac{A \cap B}{A \cup B}$$

Die Jaccard Ähnlichkeit misst also die Gesamtzahl der identischen Elemente in zwei Mengen geteilt durch die Gesamtzahl vorhandener Elemente in beiden Mengen. Wenn zum Beispiel zwei Mengen drei Elemente gemeinsam haben und beide Mengen insgesamt neun verschiedene Elemente beinhalten, dann beträgt die Jaccard Ähnlichkeit  $\frac{3}{9} = 0.33$  (siehe Abb. 10.3).

Offensichtlich nimmt die Jaccard Ähnlichkeit Werte im Interval  $[0, 1]$  an. Mit

$$d_j(A, B) = 1 - \frac{A \cap B}{A \cup B}$$

kann deshalb einfach ein Distanz- statt ein Ähnlichkeitsmass definiert werden.

Nachfolgend ist die Jaccard Ähnlichkeit mit einer Python Funktion umgesetzt. Beachten Sie, dass wir hier die eingebaute Datenstruktur `set` und die zugehörigen Methoden `intersection` und `union` verwenden (siehe Abschnitt 6.4).

```
def jaccard_similarity(set1, set2):
    return len(set1.intersection(set2)) / len(set1.union(set2))

s1 = {0, 1, 2, 5, 6}
s2 = {0, 2, 3, 4, 5, 7, 9}

print(jaccard_similarity(s1, s2))
```

Die Jaccard Ähnlichkeit wird häufig in Anwendungen verwendet, bei denen binäre Daten verwendet werden. Das heisst, die zu vergleichenden Datenobjekte  $\mathbf{x} = (x_1, \dots, x_n)$  und  $\mathbf{y} = (y_1, \dots, y_n)$  bestehen aus  $n$  binären Merkmalen (0 oder 1). Folgende Häufigkeiten lassen sich berechnen:

- $f_{00}$  = Anzahl der Merkmale mit  $x_i = 0$  und  $y_i = 0$
- $f_{01}$  = Anzahl der Merkmale mit  $x_i = 0$  und  $y_i = 1$
- $f_{10}$  = Anzahl der Merkmale mit  $x_i = 1$  und  $y_i = 0$
- $f_{11}$  = Anzahl der Merkmale mit  $x_i = 1$  und  $y_i = 1$

Die Jaccard Ähnlichkeit ist nun folgendermassen definiert:

$$s_j(\mathbf{x}, \mathbf{y}) = \frac{f_{11}}{f_{11} + f_{01} + f_{10}}$$

Hinweis: Es existieren zahlreiche weitere Ähnlichkeitsmasse für binäre Datenobjekte (z.B. Tanimoto, Yule, Dice, u.a.)

## 10.4 Levenshtein Distanz

Ein *Alphabet*  $A$  ist eine endliche Menge von Symbolen oder Zeichen. Wir können zum Beispiel mit  $A = \{0, 1\}$  oder  $A = \{a, b, c, \dots, z\}$  arbeiten. Eine *Zeichenkette*  $x$  über einem Alphabet  $A$  ist eine Folge  $x = x_1 \dots x_n$  mit  $x_i \in A$  für  $1 \leq i \leq n$ . Die Länge einer Folge  $x = x_1 \dots x_n$  ist die Anzahl ihrer Zeichen, geschrieben als  $|x| = n$ . Die *leere Zeichenkette*  $\varepsilon$  ist die Zeichenkette, die die Länge Null hat.

Im Folgenden betrachten wir zwei Zeichenketten

$$x = x_1 \dots x_n \text{ und } y = y_1 \dots y_m$$

aus dem Alphabet  $A$  und wir versuchen, das Problem der Bestimmung der Distanz zwischen  $x$  und  $y$  zu lösen. Intuitiv setzen wir folgende Idee um: Zwei Zeichenketten  $x$  und  $y$  sind sich ähnlich, wenn die Anzahl der Änderungen, die erforderlich sind um  $x$  in  $y$  umzuwandeln, gering ist. Umgekehrt sind sich  $x$  und  $y$  unähnlich, wenn viele Änderungen erforderlich sind um  $x$  in  $y$  umzuwandeln.

Wir bezeichnen die Änderungen, die eine Zeichenkette  $x$  in die andere Zeichenkette  $y$  umwandeln, als *Editieroperationen*, und wir erlauben die folgenden drei Editieroperationen:

- Ersetzung eines Symbols  $a$  durch  $b$ :  $a \rightarrow b; a, b \in A; a \neq b$
- Einfügung eines Symbols  $a$ :  $\varepsilon \rightarrow a; a \in A$
- Löschung eines Symbols  $a$ :  $a \rightarrow \varepsilon; a \in A$

**Beispiel 3** Gegeben sind zwei Zeichenketten

$$x = INDUSTRY \quad \text{und} \quad y = INTEREST$$

Durch mehrmaliges Anwenden der drei Editieroperationen gelingt es, die Zeichenkette  $x$  vollständig in  $y$  umzuwandeln:

- $INDUSTRY \xrightarrow{D \rightarrow \varepsilon} INUSTRY$
- $INUSTRY \xrightarrow{U \rightarrow \varepsilon} INSTRY$
- $INSTRY \xrightarrow{Y \rightarrow S} INSTRS$
- $INSTRS \xrightarrow{\varepsilon \rightarrow E} INSTERS$
- $INSTERS \xrightarrow{\varepsilon \rightarrow E} INSTERES$
- $INSTERES \xrightarrow{S \rightarrow \varepsilon} INTERES$
- $INTERES \xrightarrow{\varepsilon \rightarrow T} INTEREST$

Im Allgemeinen gibt es mehrere Möglichkeiten, eine Zeichenkette  $x$  in eine andere Zeichenkette  $y$  umzuwandeln.

**Beispiel 4** Gegeben sind  $x = ABBA$  and  $y = BABA$ .

Sequenz 1:  $ABBA \xrightarrow{A \rightarrow B} BBBA; BBBA \xrightarrow{B \rightarrow A} BABA$

Sequenz 2:  $ABBA \xrightarrow{\varepsilon \rightarrow A} BBA; BBA \xrightarrow{B \rightarrow \varepsilon} BA; BA \xrightarrow{\varepsilon \rightarrow B} BAB; BAB \xrightarrow{\varepsilon \rightarrow A} BABA$

Für jede der drei Editieroperationen können Kosten  $c$  definiert werden:

- Kosten für Ersetzung:  $c(a \rightarrow b)$
- Kosten für Einfügung:  $c(\varepsilon \rightarrow a)$
- Kosten für Löschung:  $c(a \rightarrow \varepsilon)$

Die intuitive Grundlage für diese Kosten ist die folgende: Je kleiner/grösser die Kosten einer Editieroperation sind, desto kleiner/grösser ist die durch die Editieroperation verursachte Änderung. Die Kosten messen also die Schwere der jeweiligen Änderung an der Zeichenkette.

**Beispiel 5** *Die Einheitskosten für Editieroperationen sind wie folgt definiert ( $a, b \in A$  und  $a \neq b$ ):*

- $c(a \rightarrow b) = c(\varepsilon \rightarrow a) = c(a \rightarrow \varepsilon) = 1$
- $c(a \rightarrow a) = 0$

Die Kosten einer Folge von Editieroperationen  $S = e_1, \dots, e_t$ , die eine Zeichenkette  $x$  in eine andere Zeichenkette  $y$  umwandelt, sind gegeben durch

$$c(S) = \sum_{i=1}^t c(e_i)$$

Wir interessieren uns für die kostenminimale Folge von Editieroperationen, um eine Zeichenkette in eine andere zu übertragen. Dies bringt uns zur Definition der *Levenshtein Distanz*.

**Definition 4 (Levenshtein Distanz)** *Die Levenshtein Distanz  $d(x, y)$  zweier Zeichenketten  $x, y$  ist gegeben durch*

$$d(x, y) = \min_{S \in \mathcal{S}} \{c(S)\}$$

wobei  $\mathcal{S}$  die Menge aller möglichen Folgen von Editieroperationen zur Umwandlung von  $x$  zu  $y$  darstellt.

Je kleiner die Levenshtein Distanz  $d(x, y)$  ist, desto weniger (bzw. kostengünstigere) Editieroperationen müssen für die Umwandlung angewendet werden und desto ähnlicher sind sich  $x$  und  $y$ .

Alg. 1 zur Berechnung der Levenshtein Distanz  $d(x, y)$  zwischen Zeichenketten  $x$  und  $y$  hat eine Laufzeit in  $O(nm)$  mit  $|x| = n$  und  $|y| = m$ . Wir gehen davon aus, dass ein Kostenmodell für die drei Editieroperationen vorliegt (z.B. die Einheitskosten).

---

**Algorithm 1** Levenshtein( $x, y$ ).

---

```

1:  $n = |x|, m = |y|$ 
2: Sei  $D$  eine zweidimensionale Liste der Dimension  $(n + 1) \times (m + 1)$ 
3:  $D[0][0] = 0$ 
4: for  $i = 1, \dots, n$  do
5:    $D[i][0] = D[i - 1][0] + c(x_i \rightarrow \varepsilon)$ 
6: end for
7: for  $j = 1, \dots, m$  do
8:    $D[0][j] = D[0][j - 1] + c(\varepsilon \rightarrow y_j)$ 
9: end for
10: for  $i = 1, \dots, n$  do
11:   for  $j = 1, \dots, m$  do
12:      $m_1 = D[i - 1][j - 1] + c(x_i \rightarrow y_j)$ 
13:      $m_2 = D[i - 1][j] + c(x_i \rightarrow \varepsilon)$ 
14:      $m_3 = D[i][j - 1] + c(\varepsilon \rightarrow y_j)$ 
15:      $D[i][j] = \min(m_1, m_2, m_3)$ 
16:   end for
17: end for
18: return  $D[n][m]$ 

```

---

Als Eingabe erwartet Alg. 1 zwei Zeichenketten

$$x = x_1 x_2 \dots x_n \text{ and } y = y_1 y_2 \dots y_m$$

In Zeile 2 wird eine zweidimensionale Liste  $D$  (die *Kostenmatrix*) der Dimension  $(n + 1) \times (m + 1)$  entsprechend den Längen der Zeichenketten initialisiert. Die Kostenmatrix  $D$  wird schrittweise gefüllt:  $D[i, j]$  enthält die Levenshtein Distanz  $d(x_1 \dots x_i, y_1 \dots y_j)$ . Schließlich enthält  $D[n, m]$  am Ende der Prozedur den Wert  $d(x, y)$ .

In Zeile 3 wird das Element  $D[0, 0]$  auf Null gesetzt, und in den Zeilen 4 bis 6 wird die Spalte 0 der Tabelle  $D$  gefüllt, wobei  $D[i, 0]$  die Summe aus  $D[i - 1, 0]$  und den beim Löschen von  $x_i$  entstehenden Kosten ist. Vertikale Bewegungen in der Kostenmatrix stellen also Löschungen des entsprechenden Symbols aus der Zeichenkette  $x$  dar. Entsprechend liegen die Kosten für die Editierfolge, welche die Zeichenkette  $x = x_1x_2 \dots x_n$  vollständig löscht, in  $D[n, 0]$ .

In den Zeilen 7 bis 9 wird die Zeile 0 der Tabelle  $D$  gefüllt, wobei  $D[0, j]$  die Summe aus  $D[0, j - 1]$  und den beim Einfügen von  $y_j$  entstehenden Kosten ist. Horizontale Verschiebungen in der Kostenmatrix stellen also Einfügungen des entsprechenden Symbols in die Zeichenkette  $y$  dar. Dementsprechend haben wir in  $D[0, m]$  schliesslich die Kosten für die Editiersequenz, welche die Zeichenkette  $y$  vollständig einfügt.

Mit Hilfe von zwei verschachtelten **for**-Schleifen wird der Rest der Kostenmatrix gefüllt. Index  $i$  läuft über die Zeichen von  $x = x_1x_2 \dots x_n$  und Index  $j$  über die Zeichen von  $y = y_1y_2 \dots y_m$ . Für jeden Eintrag  $D[i, j]$  wird geprüft, was mit den aktuellen Zeichen  $x_i$  und  $y_j$  geschehen soll:

- $m_1$  (*Ersetzung*): Symbol  $x_i$  wird durch  $y_j$  ersetzt ( $x_i \rightarrow y_j$ ). Der Vorgänger dieser Operation befindet sich in  $D[i - 1, j - 1]$  (*Diagonalschritt*).
- $m_2$  (*Lösung*): Symbol  $x_i$  wird gelöscht ( $x_i \rightarrow \varepsilon$ ). Der Vorgänger dieser Operation befindet sich in  $D[i - 1, j]$  (*Vertikalschritt*).
- $m_3$  (*Einfügung*): Symbol  $y_j$  wird eingefügt ( $\varepsilon \rightarrow y_j$ ). Der Vorgänger dieser Operation steht in  $D[i, j - 1]$  (*Horizontalschritt*).

Die Operation, welche die partielle Levenshtein Distanz  $d(x_1 \dots x_i, y_1 \dots y_j)$  minimiert, wird ausgewählt (Zeilen 12 bis 15).

**Beispiel 6** Gegeben sind zwei Zeichenketten

$$x = ABABBB \text{ and } y = BABAAA$$

Die Kosten sind folgendermassen definiert:

$$c(A \rightarrow \varepsilon) = c(B \rightarrow \varepsilon) = c(\varepsilon \rightarrow A) = c(\varepsilon \rightarrow B) = 1$$

$$c(A \rightarrow B) = c(B \rightarrow A) = 2 \quad A \neq B$$

Dies ergibt nach Alg. 1 die Distanzmatrix  $D$

	$\varepsilon$	$B$	$A$	$B$	$A$	$A$	$A$
$\varepsilon$	0	1	2	3	4	5	6
$A$	1	2	1	2	3	4	5
$B$	2	1	2	1	2	3	4
$A$	3	2	1	2	1	2	3
$B$	4	3	2	1	2	3	4
$B$	5	4	3	2	3	4	5
$B$	6	5	4	3	4	5	6

Nach dieser Kostenmatrix beträgt die Levenshtein-Distanz  $d(x, y)$  also

$$d(ABABBB, BABAAA) = 6$$

Im folgenden Programm ist die Levenshtein Distanz als Funktion definiert. Beachten Sie, dass für die Kosten der drei Editieroperationen (Lösung, Einfügung und Ersetzung) je eine Hilfsfunktion programmiert ist. Ansonsten folgt der Quellcode den Vorgaben aus Alg. 1.

```

def cost_del():
    return 1

def cost_ins():
    return 1

def cost_sub(symbol1, symbol2):
    if symbol1 == symbol2:
        return 0
    else:
        return 2

def levenshtein(x, y):
    n = len(x)
    m = len(y)
    D = [[0 for x in range(m + 1)] for x in range(n + 1)]
    for i in range(1, n + 1):
        D[i][0] = D[i - 1][0] + cost_del()
    for j in range(1, m + 1):
        D[0][j] = D[0][j - 1] + cost_ins()
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            m1 = D[i - 1][j - 1] + cost_sub(x[i - 1], y[j - 1])
            m2 = D[i - 1][j] + cost_del()
            m3 = D[i][j - 1] + cost_ins()
            D[i][j] = min(m1, m2, m3)
    return D[n][m]

print(levenshtein("ABABBB", "BABAAA"))

```

# Kapitel 11

## Daten Gruppieren

### 11.1 Clustering: Definition und Ziele

*Clustering-Algorithmen* gruppieren Datenobjekte auf Basis ihrer Merkmale in verschiedene Gruppen (die man *Cluster* nennt). Das Ziel eines Clusterings ist es, dass Objekte innerhalb eines Clusters einander ähnlich sind und sich gleichzeitig von den Objekten in anderen Clustern unterscheiden. Je grösser die Ähnlichkeit der Objekte innerhalb eines Clusters und je grösser der Unterschied zwischen Objekten aus unterschiedlichen Clustern ist, desto besser ist das Clustering.

Anders als bei der *Klassifikation* (siehe nächstes Kapitel) geht es beim Clustering nicht darum, *neue* Datenobjekte einer Gruppe zuzuordnen, sondern nur darum, die *aktuelle* Datenmenge in *sinnvolle* oder *nützliche* Cluster zu unterteilen:

- Wenn *sinnvolle* Cluster zum *Verständnis* der Daten das Ziel sind, dann sollten die Cluster die natürliche Struktur der Daten erfassen. Sinnvolle Cluster von Objekten, die gemeinsame Merkmale aufweisen, spielen eine wichtige Rolle dabei, wie Menschen die Welt analysieren, verstehen und beschreiben.

- In anderen Fällen ist das Resultat eines Clusterings nur ein *nützlicher Ausgangspunkt* für andere Zwecke, z.B. für die Zusammenfassung von Daten. Einige Clustertechniken erlauben zum Beispiel die Zusammenfassung jedes Clusters in Form eines Clusterprototyps, d.h. eines Datenobjekts, das repräsentativ für die anderen Objekte im Cluster steht.

Formal ist ein Clustering wie folgt definiert:

**Definition 5 (Clustering)** Sei  $X = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$  die Menge der Datenobjekte mit

$$\mathbf{x}_i = (x_{i_1}, \dots, x_{i_n}) \in \mathbb{R}^n$$

Ein Clustering ist eine Zerlegung von  $X$  in  $K$  disjunkte Teilmengen – die Cluster. Ein Clustering  $R$  ist definiert als

$$R = \{C_1, \dots, C_K\}$$

mit

$$C_i \cap C_j = \emptyset$$

für  $i \neq j$ ;

$$\bigcup_{i=1}^K C_i = X$$

Das heisst, ein Clustering gemäss dieser Definition ist *vollständig* und *exklusiv*. Mit anderen Worten: *jedes* Datenobjekt wird *genau einem* Cluster zugewiesen (es existieren weitere Möglichkeiten Clusterings zu definieren).

Beachten Sie, dass die Anzahl der Cluster  $K$  im Allgemeinen nicht bekannt ist und in praktischen Anwendungen oft geschätzt werden muss.

Im Prinzip könnte jede Clustering-Aufgabe so gelöst werden, dass für eine gegebene Menge von Datenobjekten  $X = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$  alle möglichen Zerlegungen berechnet und mit einem geeigneten Gütekriterium bewertet werden. Dieser Ansatz scheitert jedoch in der Regel an der grossen Anzahl kombinatorischer Möglichkeiten. Für  $N=30$  und  $K=3$  gibt es z.B. etwa  $3.4 \cdot 10^{13}$  verschiedene Clusterings. Daher ist dieser *Brute-Force*-Ansatz keine brauchbare Lösung.

Es wurden zahlreiche Clustering-Algorithmen entwickelt, welche Clusterings effizient berechnen können (im nächsten Abschnitt wird die erste Kategorie von Clustering-Algorithmen genauer vorgestellt):

- Hierarchische Clusterings
- $k$ -Means Clustering
- DBScan Clustering
- Spektrale Clusterings
- u.v.a

## 11.2 Hierarchische Clusterings

Hierarchische Clusterings sind eine wichtige Kategorie von Clustering-Algorithmen. Es gibt zwei grundlegende Ansätze, um ein hierarchisches Clustering zu erzeugen:

- *Agglomerativ*: Wir starten mit den Datenobjekten als einzelne Cluster und wir verschmelzen in jedem Schritt das nächstgelegene Paar von Clustern miteinander.

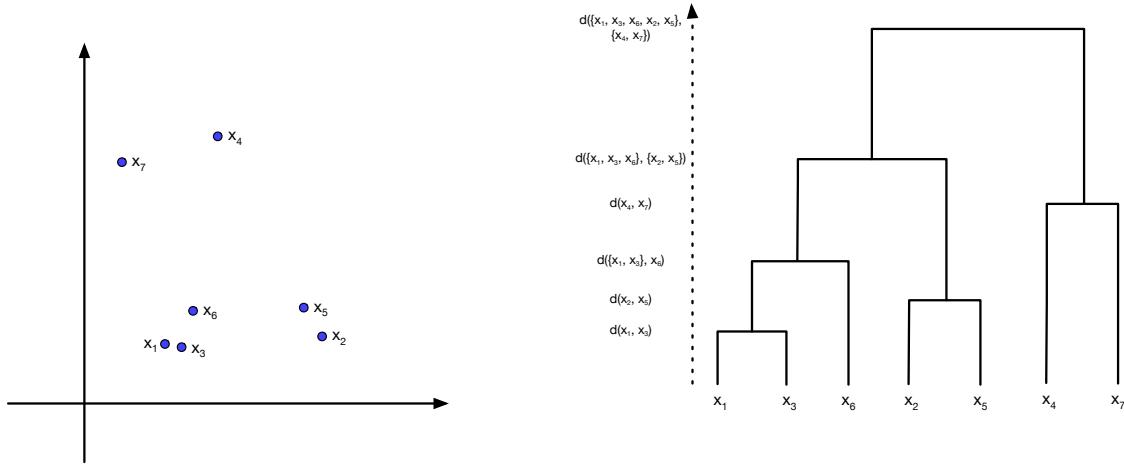


Abbildung 11.1: Zweidimensionale Datenobjekte und ein zugehöriges Dendrogramm.

- *Divisiv*: Wir starten mit einem einzigen, alles umfassenden Cluster und teilen in jedem Schritt ein Cluster in zwei Teile auf.

Ein hierarchisches Clustering wird oft grafisch durch ein baumartiges Diagramm, ein sogenanntes *Dendrogramm*, dargestellt. Ein Dendrogramm zeigt sowohl die Cluster-Teilcluster-Beziehungen als auch die Reihenfolge, in der die Cluster zusammengeführt (agglomerativ) oder aufgeteilt (divisiv) werden.

**Beispiel 7** Abb. 11.7 zeigt ein Beispiel eines Dendrogrammes.

- Zunächst werden die Datenobjekte  $\mathbf{x}_1$  und  $\mathbf{x}_3$  auf der Höhe  $d(\mathbf{x}_1, \mathbf{x}_3)$  zu einem Cluster  $\{\mathbf{x}_1, \mathbf{x}_3\}$  zusammengefasst.
- Im nächsten Schritt werden die Datenobjekte  $\mathbf{x}_2$  und  $\mathbf{x}_5$  auf der Höhe  $d(\mathbf{x}_2, \mathbf{x}_5)$  zu einem Cluster  $\{\mathbf{x}_2, \mathbf{x}_5\}$  zusammengefasst.
- Im nächsten Schritt wird das Cluster  $\{\mathbf{x}_1, \mathbf{x}_3\}$  und das Datenobjekt  $\mathbf{x}_6$  auf der Höhe  $d(\{\mathbf{x}_1, \mathbf{x}_3\}, \mathbf{x}_6)$  zu einem Cluster  $\{\mathbf{x}_1, \mathbf{x}_3, \mathbf{x}_6\}$  zusammengefasst.
- etc.

Ein Dendrogramm ohne überlappende Kanten ist immer möglich (gegebenenfalls muss hierzu aber eine geeignete Reihenfolge der Datenobjekte  $\mathbf{x}_i$  gewählt werden).

Ein Dendrogramm kann einerseits direkt als Ergebnis des Clusterings verwendet werden, da dieses einen visuellen Eindruck der Struktur der Daten vermittelt. Andererseits kann das Dendrogramm auch verwendet werden, um die Menge  $X$  in eine bestimmte Anzahl von Clustern zu zerlegen. Dazu muss im Dendrogramm lediglich ein horizontaler Schnitt zwischen zwei Ebenen durchgeführt werden.

**Beispiel 8** In Abb. 11.2 wird das Dendrogramm auf zwei verschiedenen Höhen horizontal geschnitten um zwei bzw. drei Cluster zu erhalten.

Die Grundform eines agglomerativen hierarchischen Clusterings ist in Algorithmus 2 dargestellt.

---

**Algorithm 2** Hierarchical-Clustering( $X$ )

---

```

1:  $R_0 = \{C_1 = \{\mathbf{x}_1\}, \dots, C_N = \{\mathbf{x}_N\}\}$ 
2:  $t = 0$ 
3: repeat
4:    $t = t + 1$ 
5:   take all cluster from  $R_{t-1}$  to  $R_t$ 
6:   determine the one with the smallest cluster distance among all possible pairs
      of clusters  $C_r, C_s$  in  $R_t$ :  $d(C_i, C_j) = \min\{d(C_r, C_s) | C_r \neq C_s; C_r, C_s \in R_t\}$ ;
7:   define a new cluster  $C_q = C_i \cup C_j$ 
8:   remove  $C_i$  and  $C_j$  from  $R_t$ 
9:   insert  $C_q$  into  $R_t$ 
10: until all  $\mathbf{x}_i \in X$  belong to the same cluster  $C$ 
```

---

Vor dem ersten Eintritt in die **repeat**-Schleife existieren  $N$  Cluster (jedes Datenobjekt bildet einen eigenen separaten Cluster):  $R_0 = \{C_1 =$

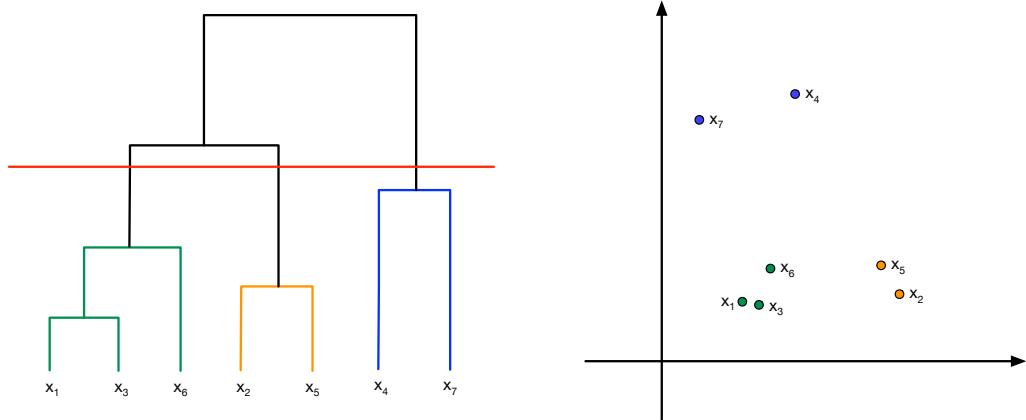
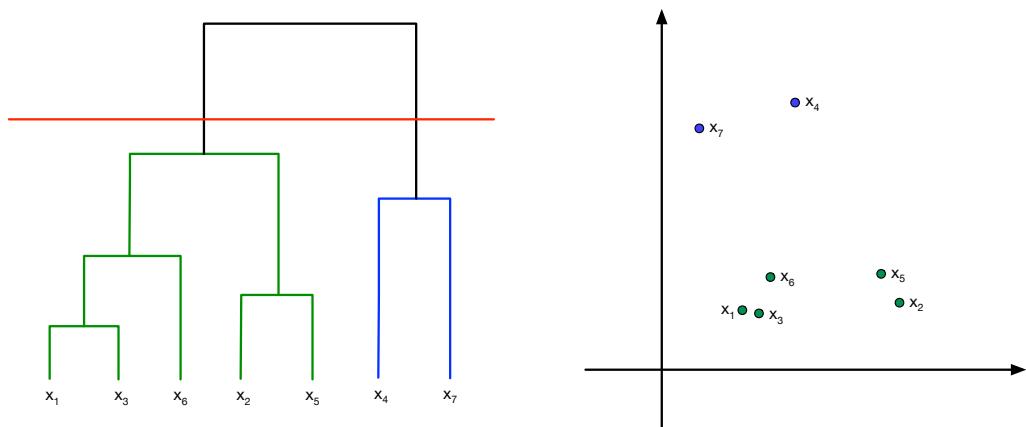


Abbildung 11.2: Aus einem Dendrogramm ein Clustering definieren.

$\{\mathbf{x}_1\}, \dots, C_N = \{\mathbf{x}_N\}$ . Bei jedem Schleifendurchlauf wird die Anzahl der Cluster um 1 reduziert, weshalb der Algorithmus nach  $N - 1$  Durchläufen mit  $R_{N-1} = \{C = \{\mathbf{x}, \dots, \mathbf{x}_N\}\}$  endet (alle Datenobjekte gehören zum selben Cluster).

Agglomerative hierarchische Clustering-Techniken verwenden verschiedene Kriterien, um in jedem Schritt zu entscheiden, welche zwei Cluster zusammengeführt werden sollen. Dieser Ansatz vermeidet die Schwierigkeit, das kombinatorische Optimierungsproblem eines Clusterings zu lösen. Allerdings kann eine getroffene Entscheidung, zwei Cluster zu verschmelzen, später nicht mehr rückgängig gemacht werden.

Die Schlüsseloperation von Algorithmus 2 ist die Berechnung der *Distanz*  $d(C_i, C_j)$  zwischen zwei Clustern  $C_i$  und  $C_j$ . Zwei sehr gebräuchliche Ansätze sind:

- **Single-Linkage Distanz**

$$d(C_i, C_j) = \min\{d(\mathbf{x}, \mathbf{y}) | \mathbf{x} \in C_i, \mathbf{y} \in C_j\}$$

Der Abstand zwischen zwei Clustern ist gleich dem Abstand zwischen den beiden ähnlichsten Datenobjekten aus  $C_i$  bzw.  $C_j$ .

- **Complete-Linkage Distanz**

$$d(C_i, C_j) = \max\{d(\mathbf{x}, \mathbf{y}) | \mathbf{x} \in C_i, \mathbf{y} \in C_j\}$$

Hier wird der Abstand zwischen den beiden entferntesten Datenobjekten aus verschiedenen Clustern verwendet.

**Beispiel 9** Betrachten Sie die euklidischen Distanzen zwischen sechs Datenobjekten in Tabelle 11.1. Gegeben seien die Cluster  $C_1 = \{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3\}$  und  $C_2 = \{\mathbf{x}_4, \mathbf{x}_5, \mathbf{x}_6\}$ .

$d(\cdot, \cdot)$	$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$	$\mathbf{x}_4$	$\mathbf{x}_5$	$\mathbf{x}_6$
$\mathbf{x}_1$	0.00	0.24	0.22	0.37	0.34	0.23
$\mathbf{x}_2$	0.24	0.00	0.15	0.20	0.14	0.25
$\mathbf{x}_3$	0.22	0.15	0.00	0.18	0.28	0.11
$\mathbf{x}_4$	0.37	0.20	0.18	0.00	0.29	0.22
$\mathbf{x}_5$	0.34	0.14	0.28	0.29	0.00	0.39
$\mathbf{x}_6$	0.23	0.25	0.11	0.22	0.39	0.00

Tabelle 11.1: Paarweise Distanzen  $d(\cdot, \cdot)$  zwischen sechs Datenobjekten.

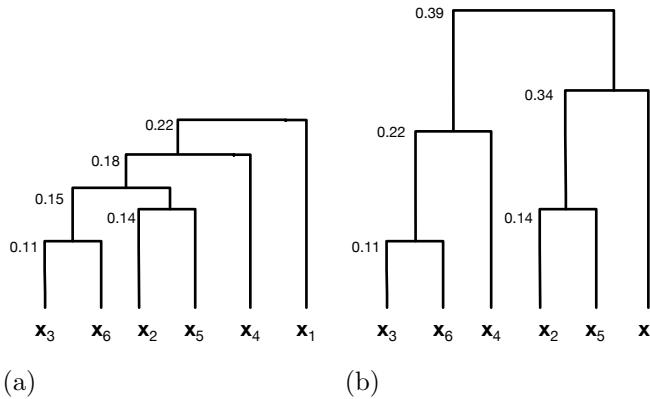


Abbildung 11.3: Dendrogramm basierend auf (a) Single-Linkage und (b) Complete-Linkage.

- *Single*:  $d(C_1, C_2) = \min\{d(\mathbf{x}, \mathbf{y}) | \mathbf{x} \in C_1, \mathbf{y} \in C_2\} = 0.11$
- *Complete*:  $d(C_1, C_2) = \max\{d(\mathbf{x}, \mathbf{y}) | \mathbf{x} \in C_1, \mathbf{y} \in C_2\} = 0.37$

**Beispiel 10** Abb. 11.3 zeigt die Dendrogramme der Clusterings mit Single- und Complete Linkage auf unseren Beispieldaten aus Tabelle 11.1. Die ersten beiden Verbindungen sind in beiden Fällen identisch. Die dritte und die folgenden Verschmelzungen unterscheiden sich hingegen und zeigen die unterschiedlichen Vorgehensweisen.

Welches Distanzmaß besser geeignet ist, lässt sich nicht verallgemeinern – die Entscheidung hängt massgeblich von den zu Grunde liegenden Daten ab. In Abb. 11.4 werden beide Distanzmasse angewendet um in künstlich erzeugten Daten zwei (obere Reihe) bzw. drei (untere

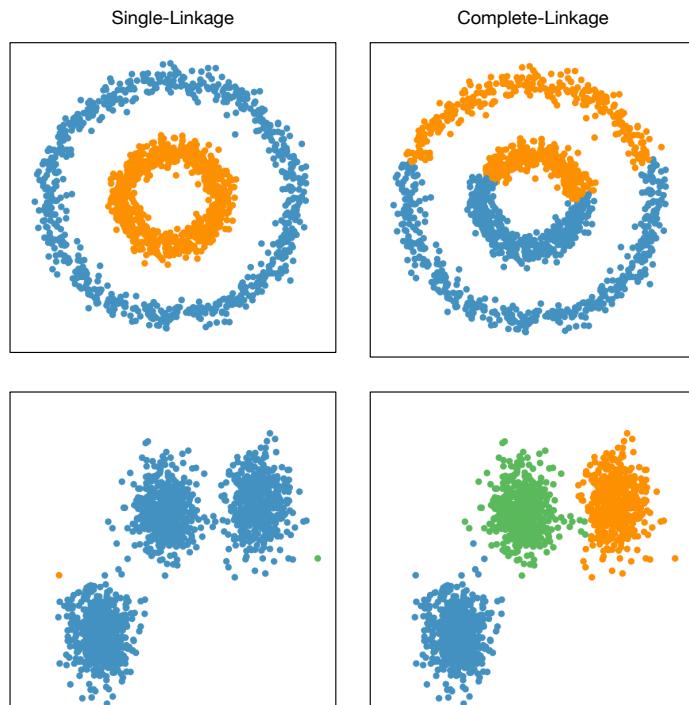


Abbildung 11.4: Single- vs. Complete-Linkage Clusterings auf künstlichen Daten.

Reihe) Cluster zu finden. Es ist deutlich sichtbar, dass auf den oberen Daten Complete-Linkage versagt, während Single-Linkage auf den unteren Daten kein gutes Ergebnis erzielt.

Hinweis: Es existieren weitere – hier nicht im Detail betrachtete – Möglichkeiten die Distanz zwischen zwei Clustern zu definieren. Man kann zum Beispiel den durchschnittlichen Abstand aller Paare von Datenobjekten aus verschiedenen Clustern berechnen oder man repräsentiert jedes Cluster durch sein Zentrum und definiert dann die Cluster Distanz als den Abstand zwischen diesen Zentren.

```

"sepal.length","sepal.width","petal.length","petal.width","variety"
5.1,3.5,1.4,.2,"Setosa"
4.9,3.1,4.,.2,"Setosa"
4.7,3.2,1.3,.2,"Setosa"
4.6,3.1,1.5,.2,"Setosa"
5.3,6.1,4.,.2,"Setosa"
5.4,3.9,1.7,.4,"Setosa"
4.6,3.4,1.4,.3,"Setosa"
5.3,4.1,1.5,.2,"Setosa"
4.4,2.9,1.4,.2,"Setosa"
4.9,3.1,1.5,.1,"Setosa"
5.4,3.7,1.5,.2,"Setosa"
4.8,3.4,1.6,.2,"Setosa"
4.8,3.1,4.,.1,"Setosa"

```

Abbildung 11.5: Die Datei `iris.csv` enthält Länge und Breite von Blütenblättern drei verschiedener Iris-Arten.



Abbildung 11.6: Die drei verschiedenen Iris-Arten.

### 11.3 Mit Python ein Clustering Berechnen

In folgendem Modul laden wir die Datei `iris.csv` (mit dem Paket `pandas`), welche die gemessene Länge und Breite von Blütenblättern drei verschiedener Iris-Arten enthält (siehe Abb. 11.5 und Abb. 11.6). Beachten Sie, wie wir die vier Spalten filtern.

Das Clustering führen wir mit Hilfe des externen Paketes `scipy` durch. Das Paket `scipy` bietet u.a. Funktionen zur Optimierung, Integration, Interpolation, und vielem anderen an. Wir importieren das Modul `scipy.cluster.hierarchy` und verwenden hieraus zwei Funktionen:

- Mit der Funktion `complete` erzeugen wir ein Clustering unter Verwendung der *Complete-Linkage Distanz*. Zu Verwendung der

Single-Linkage Distanz existiert die Funktion `single`:

```
linkage_array = hierarchy.single(data)
```

Die Rückgabe beider Funktion definiert die Distanzen, auf denen zwei Cluster miteinander verschmolzen werden.

- Distanzen, auf denen zwei Cluster miteinander verschmolzen werden (in unserem Beispiel in der Variablen `linkage_array` gespeichert), wird als Parameter an die Funktion `dendrogram` übergeben (um ein Dendrogramm zu erzeugen). Mit dem zweiten Parameter `color_threshold` kann angegeben werden, ab welcher Höhe im Dendrogramm die Teilcluster anders eingefärbt werden sollen (in unserem Beispiel setzen wir `color_threshold=3.5`).

Mit Hilfe von `matplotlib` wird schliesslich das Dendrogramm als Grafik erzeugt und angezeigt (siehe Abb. 11.7). Das Dendrogramm offenbart die Struktur der Daten sehr schön und man kann die drei Iris-Arten deutlich erkennen.

```
import pandas as pd
import scipy.cluster.hierarchy as hierarchy
import matplotlib.pyplot as plt

# load the data
data = pd.read_csv('iris.csv')
data = data[['sepal.length", "sepal.width", "petal.length", "petal.width"]]

# cluster the data
linkage_array = hierarchy.complete(data)
hierarchy.dendrogram(linkage_array, color_threshold=3.5)

# plot dendrogram
plt.xlabel("Data Object")
plt.ylabel("Cluster Distance")
plt.show()
```

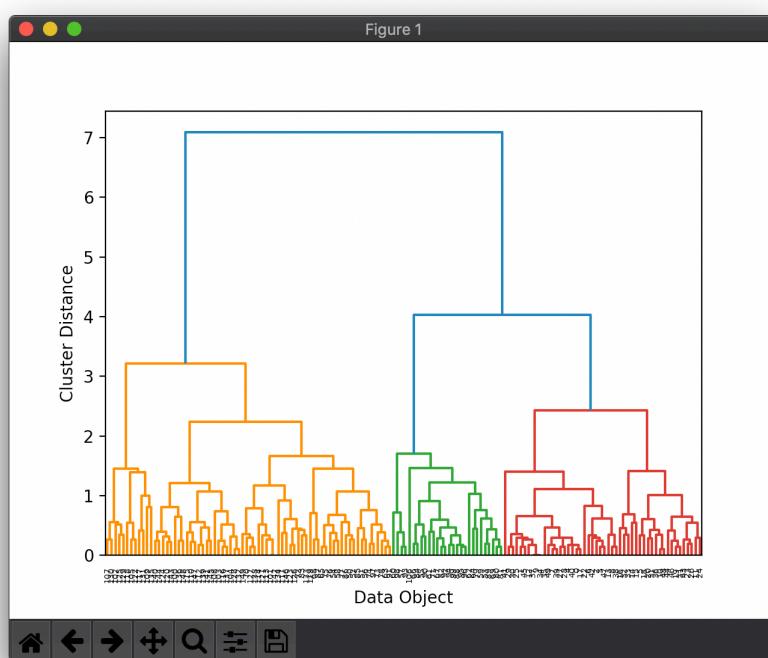


Abbildung 11.7: Ein Dendrogramm, das die Struktur der Iris Daten zeigt.

# Kapitel 12

## Daten Klassifizieren

Die Klassifikation von Daten, d.h. die Zuordnung von Datenobjekten zu einer von mehreren vordefinierten Kategorien, ist ein weit verbreitetes Problem, das viele verschiedene Anwendungen umfasst. Dieses Kapitel beschreibt zunächst die allgemeine Vorgehensweise der Klassifikation und einige der wichtigsten Probleme (wie z.B. das Übertrainieren von Modellen), die man dabei lösen muss. Danach wird ein konkretes Klassifikationsmodell vorgestellt und in Python umgesetzt.

### 12.1 Lernen eines Klassifikationsmodells

In den Anfängen der "intelligenten" Anwendungen verwendeten viele Systeme von Menschen codierte "**if-else**"-Regeln, um Daten automatisiert zu klassifizieren (denken Sie z.B. an einen Spam-Filter: eine Mail wird als Spam markiert, wenn diese genügend Wörter aus einer *Black-List* enthält). Die manuelle Erstellung von Entscheidungsregeln ist für einige Anwendungen machbar (insbesondere bei Klassifikationsproblemen, für die Menschen ein gutes Verständnis besitzen). Die spannendsten Klassifikationsprobleme sind aber meist so komplex, dass das manuelle Erstellen von Regeln schwierig oder unmöglich ist.

Moderne Klassifikationssysteme verwenden *maschinelles Lernen* um Klassifikationsprobleme zu lösen. Im Gegensatz zum handcodierten Ansatz, bei dem genaue Spezifikationen des Algorithmus erforderlich sind, soll die Maschine bei diesem Ansatz selbst lernen, Objekte zu identifizieren oder zwischen ihnen zu unterscheiden.

Beim maschinellen Lernen wird eine Maschine mit Trainingsdaten aus einer bestimmten Problemdomäne "gefüttert", worauf diese versucht, ein Modell aus den Daten zu generieren, welches das gegebene Klassifikationsproblem lösen kann. Dieser Ansatz ist inspiriert von der menschlichen Fähigkeit zu lernen: Zeigt man einem Kleinkind z.B. ein paar Beispiele von Hunden (zusammen mit der Information, dass dies jeweils ein Hund ist), wird dieses Kind in die Lage versetzt, ein vorher noch nie gesehenes Tier als Hund zu erkennen.

Es existieren viele Algorithmen, welche ein Klassifikationsmodell auf Basis von Trainingsdaten lernen können (*Entscheidungsbäume*, *Neuronale Netze*, *Support Vektor Maschinen*, u.v.a.). Bei all diesen Algorithmen bestehen die Trainingsdaten aus einer Sammlung von Datenobjekten. Jedes Datenobjekt der Trainingsdaten wird durch ein Tupel  $(\mathbf{x}, y)$  charakterisiert, wobei  $\mathbf{x} \in \mathbb{R}^n$  der Merkmalsvektor und  $y$  ein kategoriales Attribut ist, das als *Klasse* bezeichnet wird. Nun soll ein Algorithmus auf den Trainingsdaten ein *Klassifikationsmodell* lernen, das einem gegebenen Datenobjekt  $\mathbf{x}$  die richtige Klasse zuweisen kann. Das Hauptziel ist es dabei, ein Modell zu lernen, das gut *generalisieren* kann: Das Klassifikationsmodell soll also insbesondere die Klasse von unbekannten Datenobjekten richtig vorhersagen können.

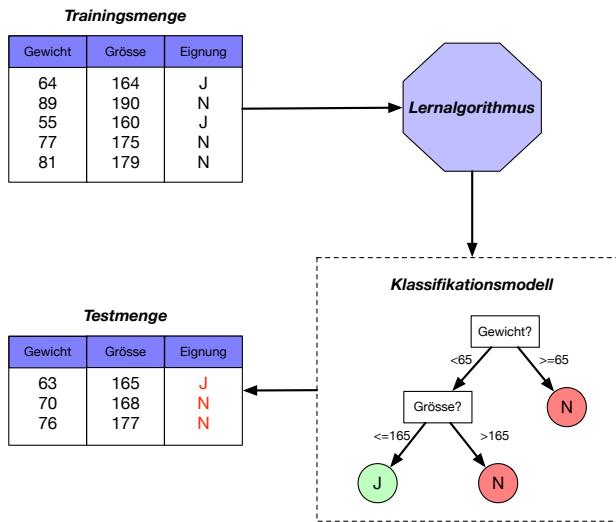


Abbildung 12.1: Ein Klassifikationsmodell wird auf der Trainingsmenge gelernt und auf einer Testmenge evaluiert.

Abb. 12.1 zeigt einen allgemeinen Ansatz zum Lösen von Klassifikationsproblemen. Zunächst muss eine *Trainingsmenge*, die aus Datenobjekten besteht, deren Klassen bekannt sind, bereitgestellt werden. Die Trainingsmenge wird verwendet, um ein Klassifikationsmodell zu lernen. Um die Leistung des Modells zu beurteilen, klassifizieren wir mit dem gelernten Modell neue Daten – die sogenannte *Testmenge* (für die wir die korrekte Klassenzugehörigkeit ebenfalls kennen).

Verwenden wir für das Lernen des Modells und dessen Evaluation die gleichen Daten, wird sich das gelernte Modell zu stark an diese Daten anpassen (bekannt als *Übertrainieren* (engl. *Overfitting*)). Ein übertrainiertes Modell kann eine perfekte Klassifikation für alle Trainingsdaten erzeugen, schneidet dann bei Testdaten aber schlecht ab. Umgekehrt tritt *Untertrainieren* (engl. *Underfitting*) auf, wenn das Modell zu einfach ist und bereits auf der Trainingsmenge versagt. Im besten Fall wählt ein Algorithmus zwischen *Under-* und *Overfitting* ab und findet so den *Sweet Spot* dazwischen (siehe Abb. 12.2).

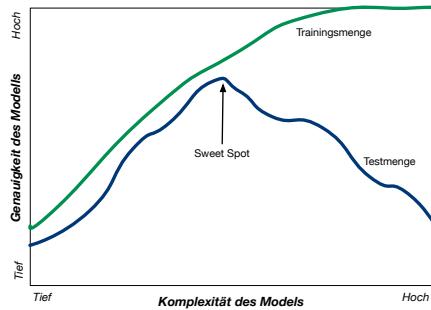


Abbildung 12.2: Wir suchen ein Klassifikationsmodell, das gut generalisiert.

		Vorausgesagte Klasse	
		Klasse 1	Klasse 0
Tatsächliche Klasse	Klasse 1	$f_{11}$	$f_{10}$
	Klasse 0	$f_{01}$	$f_{00}$

Tabelle 12.1: Eine Konfusionsmatrix für zwei Klassen.

Die Bewertung eines Klassifikationsmodells basiert oftmals auf der Anzahl der Datenobjekte, die vom Modell richtig oder falsch klassifiziert wurden. Diese Anzahlen können in einer Tabelle, der sogenannten *Konfusionsmatrix*, tabellarisch dargestellt werden. In Tabelle 12.1 ist die Konfusionsmatrix für ein Klassifikationsproblem mit zwei Klassen dargestellt. Die vier Ziffern entsprechen:

1.  $f_{11}$  (*True Positive (TP)*): Anzahl der Datenobjekte der *Klasse 1*, die korrekt als *Klasse 1* klassifiziert werden.
2.  $f_{00}$  (*True Negative (TN)*): Anzahl der Datenobjekte aus der *Klasse 0* die korrekt als *Klasse 0* klassifiziert sind.
3.  $f_{01}$  (*False Positive (FP)*): Anzahl der Datenobjekte der *Klasse 0*, die fälschlicherweise als *Klasse 1* klassifiziert werden.
4.  $f_{10}$  (*False Negative (FN)*): Anzahl der Datenobjekte der *Klasse 1*, die fälschlicherweise als *Klasse 0* klassifiziert werden.

Die Gesamtzahl der richtigen Vorhersagen des Modells ist  $f_{11} + f_{00}$  und die Gesamtzahl der falschen Vorhersagen ist  $f_{01} + f_{10}$ . Die *Fehlerrate*  $E$  bzw. *Erkennungsrate*  $RR$  eines Modells ist gegeben durch den relativen Anteil der falsch bzw. richtig klassifizierten Datenobjekte gemessen an allen Vorhersagen:

- $E = \frac{f_{01} + f_{10}}{f_{11} + f_{00} + f_{01} + f_{10}}$
- $RR = \frac{f_{11} + f_{00}}{f_{11} + f_{00} + f_{01} + f_{10}}$

Offensichtlich ist die Klassenverteilung wichtig für die Beurteilung der Qualität eines Modells. Wenn z.B. 98% aller Datenobjekte aus der Klasse 1 und nur 2% aus der Klasse 0 stammen, entspricht eine Erkennungsrate von 98% keinem guten Ergebnis. Deshalb werden je nach Anwendung und Datenlage auch andere Qualitätskriterien zur Beurteilung von Klassifikationsmodellen berechnet, z.B. die *Precision*  $p$  und/oder der *Recall*  $r$ :

- $p = \frac{f_{11}}{f_{11} + f_{01}}$
- $r = \frac{f_{11}}{f_{11} + f_{10}}$

Die *Precision* gibt den Anteil der richtig klassifizierten positiven Objekte an allen als positiv klassifizierten Datenobjekten an. Je weniger *False-Positives* erzeugt werden, desto grösser ist die *Precision* eines Klassifikators (er sagt die positive Klasse mit einer hohen *Präzision* voraus). Der *Recall* definiert den Anteil der positiven Datenobjekte, die tatsächlich als positiv klassifiziert werden. Klassifikatoren mit einem hohen *Recall*-Wert erzeugen sehr wenige falsche Zuordnungen von positiven Objekten (sie sind in der Lage, sich an die positiven Datenobjekte zu *erinnern*).

## 12.2 Der $k$ -NN Klassifikator

Der  *$k$ -Nächster-Nachbar Klassifikator* ( $k$ -NN) ist der wohl einfachste Algorithmus zum Erlernen eines Klassifikationsmodells aus Daten. Das Lernen des Modells besteht nur aus dem Speichern der Trainingsmenge. Um eine Vorhersage für ein neues Datenobjekt zu treffen, findet der  $k$ -NN die nächstgelegenen Datenobjekte in der Trainingsmenge und weist dem unbekannten Datenobjekt die Klasse dieser nächsten Nachbarn zu. In seiner einfachsten Version berücksichtigt der  $k$ -NN nur genau einen nächsten Nachbarn ( $k = 1$ ).

**Definition 6 (Nächster-Nachbar-Klassifikator)** Gegeben sei eine klassifizierte Trainingsmenge  $X = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)\}$ , wobei  $\mathbf{x}_i$  der Merkmalsvektor und  $y_i$  die Klasse bezeichnet. Mit dem Nächster-Nachbar-Klassifikator (1-NN) ordnen wir ein unbekanntes Datenobjekt der Klasse zu, aus der der nächste Nachbar aus der Trainingsmenge stammt. Das abgeleitete Klassifikationsmodell entspricht folgender Regel:

$$\mathbf{x} \in y_j \Leftrightarrow d(\mathbf{x}, \mathbf{x}_j) = \min\{d(\mathbf{x}, \mathbf{x}_k) \mid k = 1, \dots, N\} \quad (12.1)$$

**Beispiel 11** Abb. 12.3 (a) veranschaulicht das Prinzip des 1-NN: Für die drei neuen Datenobjekte (graue Quadrate) wird das nächstgelegene Datenobjekt in der Trainingsmenge markiert. Die Vorhersage entspricht der Klasse des nächsten Nachbarn (dargestellt durch die Farbe).

Der 1-NN ist recht anfällig für Ausreisser in der Trainingsmenge. Eine Erweiterung dieses Klassifikationsmodells ergibt sich dadurch, dass nicht nur ein sondern  $k > 1$  nächste Nachbarn verwendet werden, wobei  $k$  im Falle von zwei Klassen meist ungerade definiert wird. Dies ergibt den sogenannten  $k$ -NN-Klassifikator:

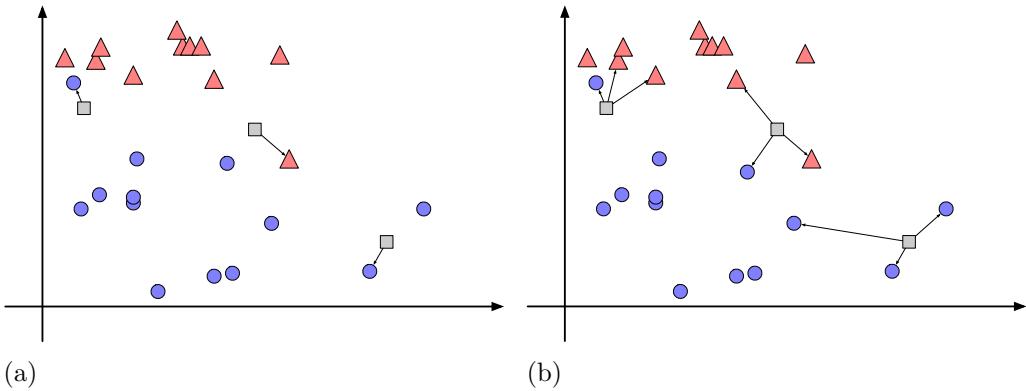


Abbildung 12.3: Illustration des (a) 1-NN und (b) 3-NN.

$\mathbf{x} \in y_j \Leftrightarrow$  die  $k$  nächsten Nachbarn von  $\mathbf{x}$  sind  $\{\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_k}\}$  und  
die Klasse  $y_j$  ist die häufigste Klasse der Objekte  $\{\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_k}\}$ .

**Beispiel 12** Abb. 12.3 (b) veranschaulicht das Prinzip des 3-NN.

Unsere Abbildungen gingen bis jetzt von einem binären Klassifikationsproblem aus. Die Methode des  $k$ -NN kann aber auch auf Probleme mit einer beliebigen Anzahl von Klassen angewendet werden. Das Prinzip bleibt das Gleiche: Der Algorithmus zählt, wie viele Nachbarn zu jeder Klasse gehören, und sagt wiederum die häufigste Klasse voraus. Bei mehr als zwei Klassen muss allerdings damit gerechnet werden, dass es zu einem Konflikt, d.h. zu einem “Unentschieden”, zwischen verschiedenen Klassen kommen kann. In solchen Fällen weisen wir  $\mathbf{x}$  zurück, treffen eine zufällige Entscheidung oder wenden eine Regel an, um den Konflikt aufzulösen. Eine häufig verwendete Konfliktlösungsregel besteht darin,  $\mathbf{x}$  der Klasse seines nächsten Nachbarn zuzuordnen (d.h. wir verwenden den 1-NN-Klassifikator<sup>1</sup>).

---

<sup>1</sup>Wenn der nächste Nachbar nicht eindeutig ist und die nächsten Nachbarn (mit gleichem Abstand) aus verschiedenen Klassen kommen, funktioniert diese Heuristik allerdings nicht.

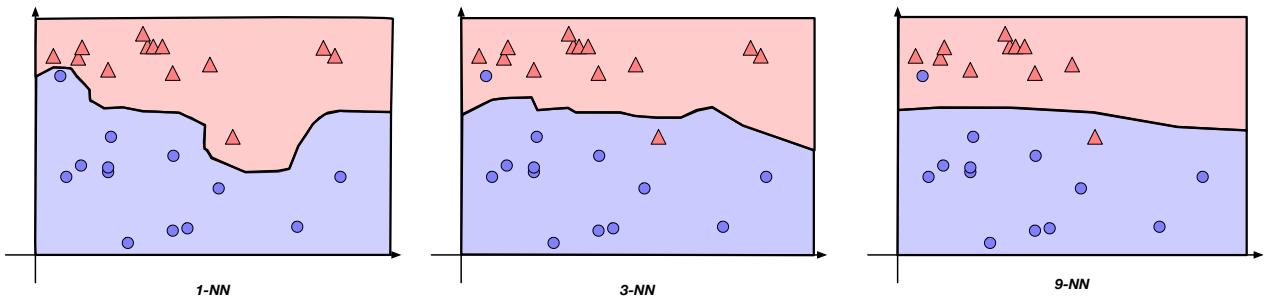


Abbildung 12.4: Der Effekt unterschiedlicher Anzahlen Nachbarn.

Für zweidimensionale Datenobjekte können wir die Vorhersage in einer Ebene darstellen. Wir färben die Ebene entsprechend der Klasse ein, die einem Datenobjekt in diesem Bereich zugewiesen wird. Auf diese Weise können wir die Entscheidungsgrenze des Modells visualisieren.

**Beispiel 13** Wie man links in Abb. 12.4 sehen kann, führt die Verwendung eines einzelnen Nachbarn zu einer Entscheidungsgrenze, die den Trainingsdaten ziemlich genau folgt. Die Berücksichtigung von mehr und mehr Nachbarn führt zu einer "glatteren" Entscheidungsgrenze. Eine glatttere Grenze entspricht einem einfacheren Modell. Mit anderen Worten, die Verwendung von wenigen Nachbarn entspricht einer hohen Modellkomplexität, und die Verwendung von vielen Nachbarn entspricht einer eher niedrigen Modellkomplexität<sup>2</sup>.

Dieses Beispiel unterstreicht, wie wichtig es ist, den richtigen Wert für  $k$  zu wählen. Zur Wahl eines geeigneten Wertes für  $k$  wird die Testmenge verwendet. Diese unabhängige Menge wird mit Hilfe der Trainingsmenge unter Verwendungen unterschiedlicher Anzahlen Nachbarn klassifiziert.

---

<sup>2</sup>Extremfall: Die Anzahl der Nachbarn entspricht der Anzahl Datenobjekte in der Trainingsmenge. In diesem Fall wird jedes Testobjekt der gleichen Klasse zugewiesen (nämlich der am häufigsten vorkommenden Klasse in der Trainingsmenge).

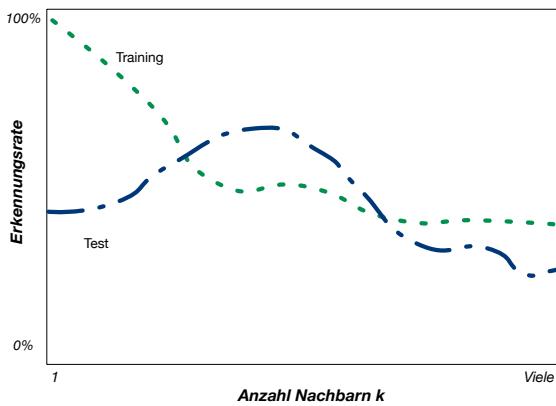


Abbildung 12.5: Mit Hilfe der Testmenge finden wir den geeigneten Wert für  $k$ .

Das Diagramm in Abb. 12.5 zeigt beispielhaft die Erkennungsraten auf der Trainings- und Testmenge auf der  $y$ -Achse in Abhängigkeit der Anzahl Nachbarn  $k$  auf der  $x$ -Achse. Bei Berücksichtigung eines einzigen nächsten Nachbarn ist die Vorhersage auf der Trainingsmenge perfekt (es passiert kein Fehler). Wenn jedoch mehr Nachbarn berücksichtigt werden, wird das Modell einfacher und die Trainingsgenauigkeit sinkt.

Die Erkennungsrate auf der Testmenge bei Verwendung eines Nachbarn ist niedriger als bei Verwendung mehrerer Nachbarn, was darauf hinweist, dass die Verwendung eines 1-NN zu einem zu komplexen Modell geführt hat (also zu einem übertrainierten Modell). Andererseits ist das Modell zu einfach, wenn zu viele Nachbarn (oder im Extremfall alle Datenobjekte) zur Klassifikation berücksichtigt werden.

Man wählt den Wert für Parameter  $k$ , welcher das beste Ergebnis auf den Testdaten erzielt. Das gleiche Prinzip wird bei anderen Klassifikationsalgorithmen angewendet. Oftmals besitzen diese Algorithmen allerdings mehr als nur einen Parameter, was die Suche nach einer optimalen Parameterkonfiguration i.A. zeitaufwändiger macht.

## 12.3 Mit Python Daten Klassifizieren

Zur Klassifikation von Daten verwenden wir das externe Paket `scikit-learn`, welches vielfältige Werkzeuge für maschinelles Lernen bereitstellt.

Wir gehen davon aus, dass wir eine klassifizierte Datenmenge zur Verfügung haben. Wir können z.B. die Iris-Daten aus der Datei `iris.csv` laden (mit dem Paket `pandas` – siehe Abschnitt 11.3). Wir verwenden die gemessene Länge und Breite von Blütenblättern als Merkmalsvektoren `X` und die jeweilige Iris-Art als Klasse `y`:

```
import pandas as pd

# load the data
data = pd.read_csv('iris.csv')
X = data[["sepal.length", "sepal.width", "petal.length", "petal.width"]]
y = data[["variety"]]
```

Als nächstes teilen wir die Daten in eine Trainings- und Testmenge auf. Hierzu verwenden wir die Funktion `train_test_split`. Standardmäßig werden 25% der Daten in die Testmenge und 75% der Daten in die Trainingsmenge verschoben (mit dem Parameter `test_size` kann das angepasst werden). Damit die Klassenverteilung bei der Aufteilung auf beiden Mengen erhalten bleibt, kann man den Parameter `stratify=y` setzen (`y` referenziert die Klasse). Da es sich um eine zufällige Aufteilung handelt, die man evtl. später genau gleich wiederholen möchte, kann man mit `random_state=43` die zufällige Wahl reproduzierbar machen (43 ist dabei ein beliebiger Wert).

```
from sklearn.model_selection import train_test_split

# split the data to training and test set
X_tr, X_te, y_tr, y_te = train_test_split(X, y, stratify=y, test_size=0.4,
                                            random_state=43)
```

Der  $k$ -NN kann nun folgendermassen importiert, trainiert und verwendet werden:

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=1, metric="euclidean")
knn.fit(X_tr, y_tr)
y_pred = knn.predict(X_te) # predictions for each test object
print("Recognition Rate on Test Set:", knn.score(X_te, y_te))
print("Predictions:", y_pred)
```

- Die Funktion `KNeighborsClassifier` erzeugt ein  $k$ -NN Objekt und erwartet einen Parameter `n_neighbors` – also die Anzahl Nachbarn. Mit `metric="euclidean"` kann zudem das Distanzmass angegeben werden (z.B. `"cosine"`, `"manhattan"`, etc.).
- Mit der Methode `fit` wird das Klassifikationsmodell `knn` auf der Trainingsmenge gelernt.
- Mit der Methode `predict` können die Voraussagen des Klassifikationsmodells `knn` für jedes Testobjekt berechnet und in einer Liste gespeichert werden.
- Mit der Methode `score` wird die Erkennungsrate des Modells `knn` berechnet.

Die Ausgabe des obigen Programmes lautet bspw.:

```
Recognition Rate on Test Set: 0.9666666666666667
Predictions: ['Virginica' 'Versicolor' 'Virginica' 'Setosa' ...]
```

Folgendes Programm findet die optimale Anzahl Nachbarn im Bereich 1, 3, ..., 31. Hierzu wird in einer `for`-Schleife jeweils die Erkennungsrate des  $k$ -NN auf der Trainings- und Testmenge berechnet und in zwei separaten Listen gespeichert. Beide Listen werden in einem Liniendiagramm ausgegeben (siehe Abb. 12.6).

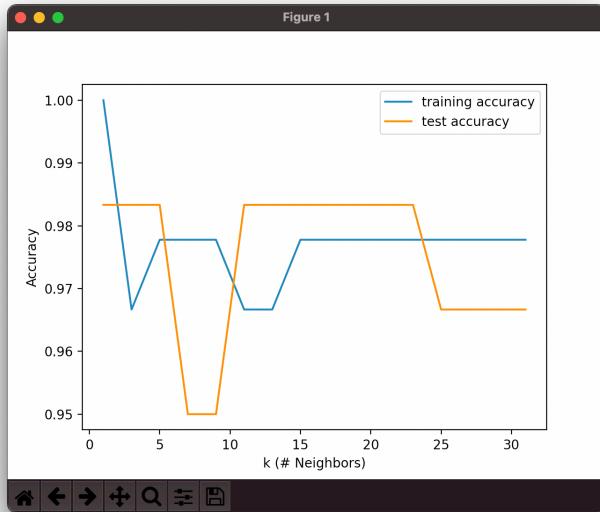


Abbildung 12.6: Die Erkennungsraten auf der Trainings- und Testmenge mit unterschiedlichen Anzahlen Nachbarn.

```

import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
import matplotlib.pyplot as plt

# load the data and split the data to training and test set as shown above

# evaluate number of neighbors via for loop
training_accuracy = []
test_accuracy = []
num_of_neighbors = range(1, 32, 2)
for n in num_of_neighbors:
    knn = KNeighborsClassifier(n_neighbors=n, metric="cosine")
    knn.fit(X_tr, y_tr)
    # store train and test accuracy
    training_accuracy.append(knn.score(X_tr, y_tr))
    test_accuracy.append(knn.score(X_te, y_te))

# plot the results
plt.plot(num_of_neighbors, training_accuracy, label="training accuracy")
plt.plot(num_of_neighbors, test_accuracy, label="test accuracy")
plt.ylabel("Accuracy")
plt.xlabel("k (# Neighbors)")
plt.legend()
plt.show()

```