

# **Programmieren für Naturwissenschaften**

## **Teil 2: Python**

**Vorlesungsskript (3.0)**  
**FS 2023**

**PD Dr. Kaspar Riesen**  
**[kaspar.riesen@unibe.ch](mailto:kaspar.riesen@unibe.ch)**

# Inhaltsverzeichnis

<b>1</b>	<b>Einführung und Motivation</b>	<b>1</b>
1.1	Weshalb Python? . . . . .	1
1.2	Ein Erstes Python Programm . . . . .	2
1.2.1	Die Eingebaute Funktion <code>print</code> . . . . .	4
1.2.2	Kommentare . . . . .	7
1.3	Programmentwicklung . . . . .	9
1.3.1	Kompilieren und Interpretieren . . . . .	9
1.3.2	Python Interaktiv Verwenden . . . . .	12
1.3.3	Python im Skript Modus Verwenden . . . . .	17
1.3.4	Programmierfehler . . . . .	19
<b>2</b>	<b>Variablen und Listen</b>	<b>22</b>
2.1	Variablen . . . . .	22
2.1.1	Eingaben Entgegennehmen . . . . .	33
2.2	Listen – die Datenstruktur <code>list</code> . . . . .	34
<b>3</b>	<b>Schleifen und Bedingungen</b>	<b>45</b>
3.1	Die <code>if</code> -Anweisung . . . . .	46
3.1.1	Boolesche Ausdrücke . . . . .	48
3.1.2	Die <code>if-else</code> Anweisung . . . . .	52
3.1.3	Verschachtelte <code>if</code> -Anweisungen . . . . .	53
3.1.4	Die <code>if-elif</code> -Anweisung . . . . .	55
3.2	Die <code>while</code> -Anweisung . . . . .	57
3.2.1	Endlosschleifen . . . . .	61
3.2.2	Verschachtelte Schleifen . . . . .	61
3.3	Die <code>for</code> -Anweisung . . . . .	63

<b>4 Standardmodule Verwenden</b>	<b>66</b>
4.1 Das Modul <code>random</code> . . . . .	67
4.2 Das Modul <code>math</code> . . . . .	71
4.3 Das Modul <code>statistics</code> . . . . .	74
4.4 Dateien Lesen und Schreiben . . . . .	76
4.4.1 Vom Umgang mit Zeichenketten . . . . .	77
4.4.2 Dateien Lesen . . . . .	81
4.4.3 Dateien Schreiben . . . . .	83
<b>5 Eigene Funktionen Programmieren</b>	<b>87</b>
5.1 Funktionen Definieren . . . . .	88
5.2 Funktionen Aufrufen . . . . .	89
5.3 Parameter an Funktionen Übergeben . . . . .	92
5.4 Verändern von Parametern in Funktionen . . . . .	96
5.5 Die <code>return</code> Anweisung . . . . .	99
5.6 Teilen-und-Beherrschen . . . . .	103
<b>6 Weitere Datenstrukturen</b>	<b>108</b>
6.1 Zweidimensionale Listen . . . . .	108
6.1.1 Kopieren von Listen . . . . .	112
6.2 Wörterbücher – Die Datenstruktur <code>dict</code> . . . . .	114
6.3 Tupel – Die Datenstruktur <code>tuple</code> . . . . .	119
6.4 Mengen – Die Datenstruktur <code>set</code> . . . . .	123
<b>7 Externe Pakete und Projekte</b>	<b>126</b>
7.1 Das Paket <code>pandas</code> . . . . .	127
7.2 Das Paket <code>matplotlib</code> . . . . .	135
7.3 Das Paket <code>scipy</code> . . . . .	141
7.3.1 Integrieren . . . . .	141
7.3.2 Optimieren . . . . .	142
7.3.3 Gruppieren . . . . .	144

# 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 zur Bezeichnung eines Menschen, der Computerprogramme schreibt: *die Programmiererin*. Die männliche Form – also *der Programmierer* – ist aber immer mitgemeint. Umgekehrt nutzen wir meist nur die männliche Form zur Bezeichnung eines Menschen, der mit einem Computerprogramm

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 Vorzüge aufweist:

- Python ist intuitiv und leicht zu erlernen
- Python kann für unterschiedlichste Zwecke verwendet werden
- Python ist plattformunabhängig
- Python ist gemäss TIOBE die populärste Programmiersprache des Jahres 2021<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

---

interagiert: *der Benutzer* – und auch hier gilt: Die weibliche Form ist immer mitgemeint.

<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 für Menschen zwar möglich, aber 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 an-

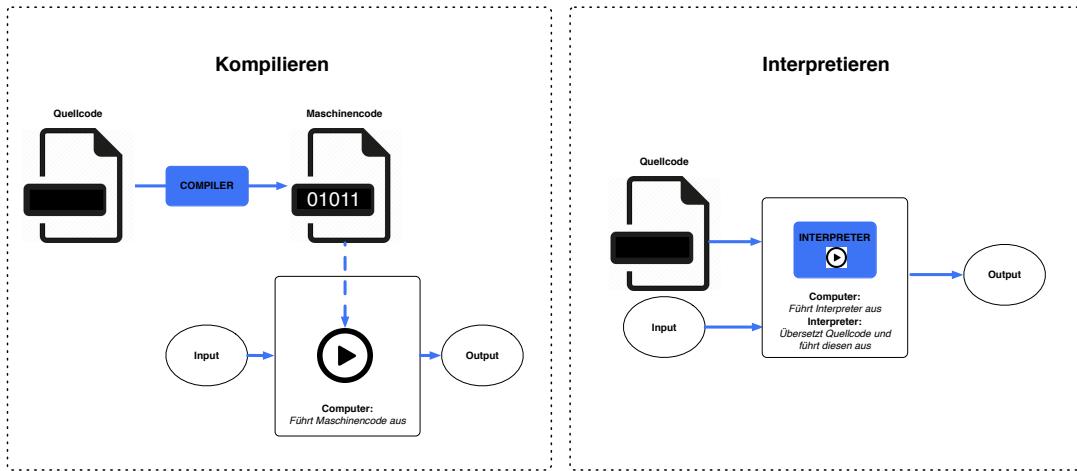


Abbildung 1.1: Kompilieren vs. Interpretieren.

dere Vorteile.

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.

- *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*

### 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 Programmentwicklung 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:

```
>>>
```

---

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

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*

...

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
```



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

```

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 ganzzahlige 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

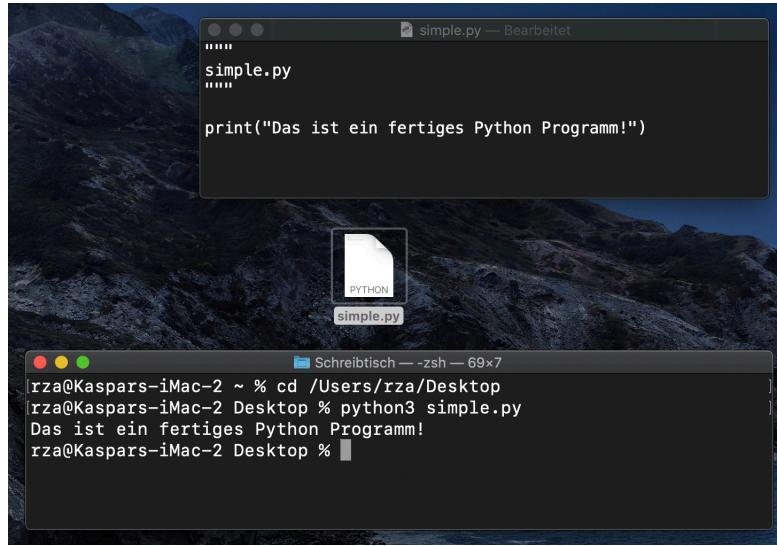


Abbildung 1.3: Python Programme in einer Konsole interpretieren und ausführen.

terminieren.

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

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

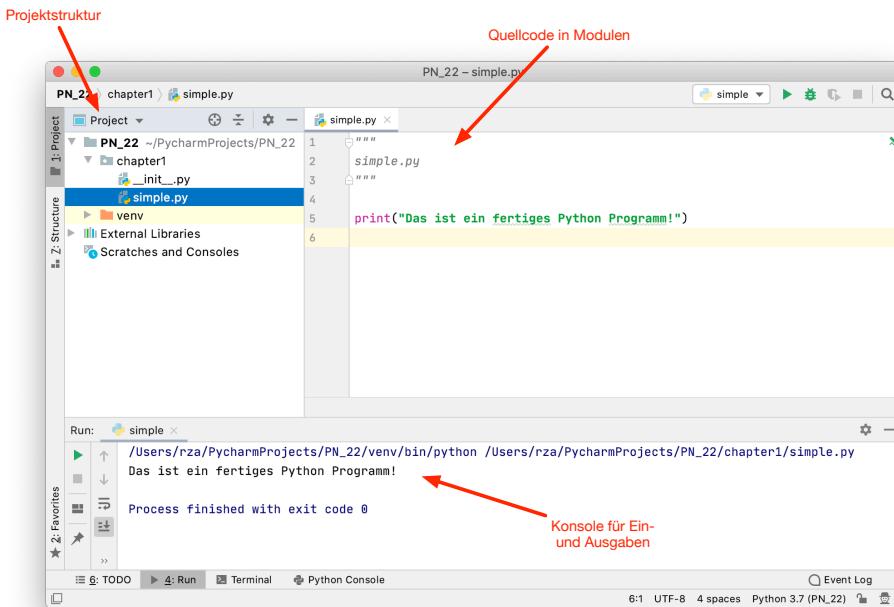


Abbildung 1.4: Python Programme in PyCharm erstellen, organisieren und ausführen.

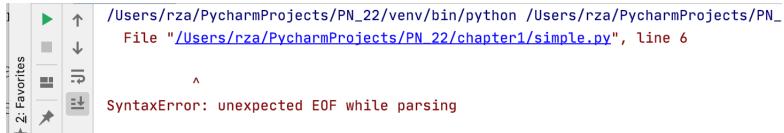
verwendet werden können, wie z.B. *PyCharm* oder *Eclipse*. Wir verwenden in dieser Vorlesung die IDE *PyCharm*<sup>7</sup> (siehe Abb. 1.4).

### 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

<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.



```

    /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.

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 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



The screenshot shows the PyCharm IDE interface. The project structure on the left includes a 'chapter1' folder containing 'simple.py', '\_init\_.py', and a 'venv' folder. The 'simple.py' file is open in the editor, showing the following code:

```
1     """  
2     simple.py  
3     """  
4  
5     print("Das ist ein fertiges Python Programm!")
```

A red squiggle underlines the word 'print' in line 5, indicating a syntax error. The status bar at the bottom right shows 'simple' and a red error icon.

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

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 Programmiererinnen und Programmierern, 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 Programmiererinnen und Programmierer nutzen können. Drei 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.)

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

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

**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>2</sup>:

```
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

---

<sup>2</sup>Wenn Sie andere Programmiersprachen kennen, kennen Sie vielleicht das Inkrement **++** und das Dekrement **--**. Diese Operatoren sind in Python nicht vorgesehen.

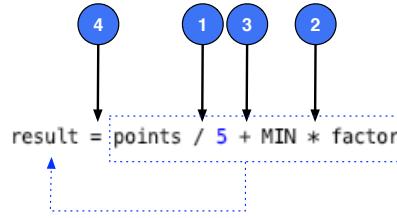


Abbildung 2.2: Die Reihenfolge einer Auswertung: Der Zuweisungsoperator `=` hat die niedrigste Priorität.

`count + 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 folgende 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). Dies gilt für alle Zuweisungsoperatoren. Das bedeutet, dass z.B.

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

äquivalent ist zu

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

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

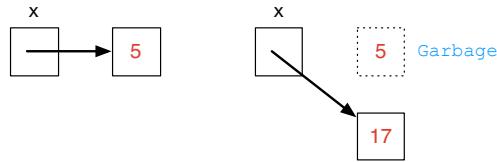


Abbildung 2.3: Werte, die nicht mehr referenziert werden, werden als *Garbage* markiert.

### 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 Variablen ändern!

*Betrachten Sie zum Beispiel die Anweisung*

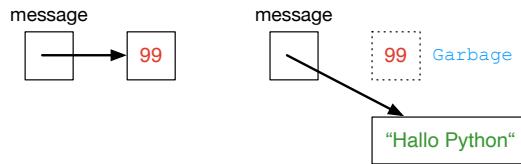


Abbildung 2.4: Eine Variable kann Daten unterschiedlicher Typen referenzieren.

```
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
'Hallo Python'
```

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

## Beispiel 19

```
>>> message = 99
>>> 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. Wol- len 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 Datenstruktu-

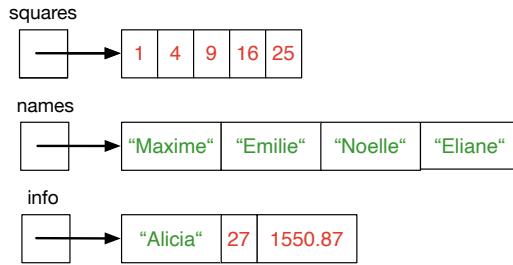


Abbildung 2.5: Drei Listen.

ren 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>3</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

---

<sup>3</sup>Obschon das eher nicht üblich ist – hierfür würde sich die Datenstruktur `tuple` besser eignen.

im Umgang mit Listen<sup>4</sup>. Die Funktion `range` erzeugt arithmetische Bereiche von `0` bis `end` (der angegebene Endpunkt ist nie Teil des 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]
```

---

<sup>4</sup>Die eckigen Klammern bedeuten, dass die Parameter `start` und `step` optional sind, nicht, dass Sie an dieser Stelle eckige Klammern eingeben sollten.

```
>>> 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:

- Standardwert für `start`: 0 (bzw. `len` – 1 falls die Schrittweite negativ ist)
- Standardwert für `end`: `len` (bzw. -1 falls die Schrittweite negativ ist)
- Standardwert für `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 ausgeben
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) *[]*

# 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.

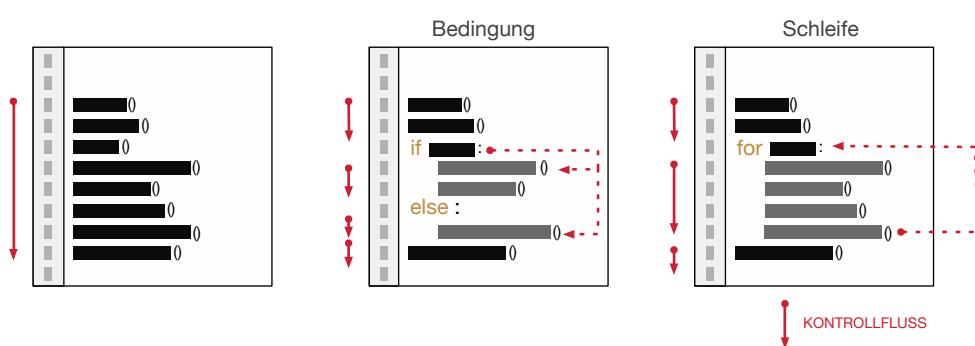


Abbildung 3.1: Schleifen und Bedingungen steuern den Kontrollfluss eines Programmes.

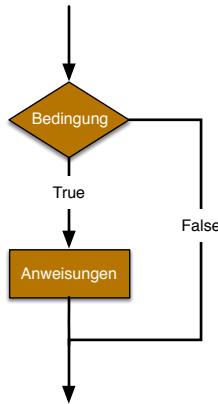


Abbildung 3.2: Die Logik einer `if`-Anweisung.

### 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 fortgefahren (siehe Abb. 3.2).

**Beispiel 37** 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 38** 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:

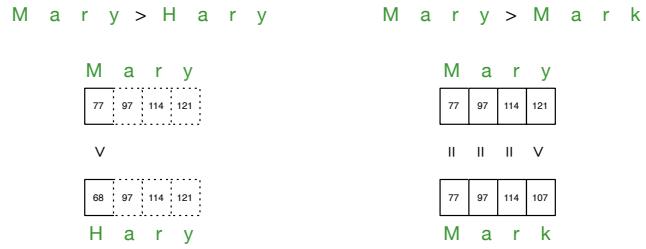


Abbildung 3.3: Zeichenketten werden auf Basis der Unicode Werte einzelner Zeichen von links nach rechts miteinander verglichen.

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 39** 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 40

```
>>> "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 41** 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 42** 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 43** 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 44** 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 45** 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 46** 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.

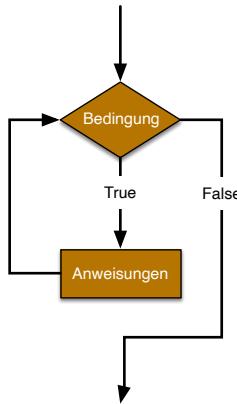


Abbildung 3.4: Die Logik von `while`-Schleifen.

## 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 47** 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 48** 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 potenzielle 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 49** 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 50** 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 51** 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

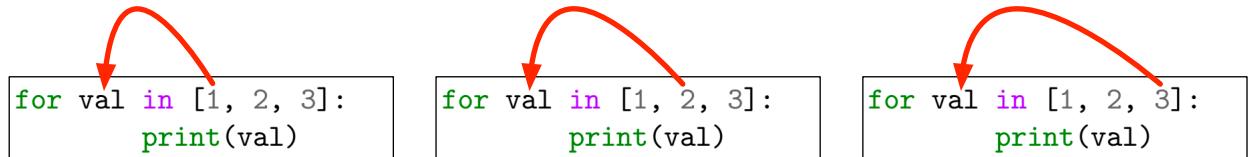


Abbildung 3.5: In jeder Iteration durch die `for`-Schleife zeigt die Laufvariable `val` auf ein anderes Element der Liste.

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).

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.

**Beispiel 52** Die Ausgabe der folgenden `for`-Schleife lautet:

```

cat
3
window
6
defenestr ate
12

```

```

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

```

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
```

Hinweis: 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.

# 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/3.8/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

```
math.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  $a \leq x \leq b$ .
- `random.randrange(a, b[, step]):`  
Liefert eine zufällige ganze Zahl  $x$ , so dass  $a \leq x < b$  und  $(x+a) \% step = 0$ . Der Standardwert für `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 53

```
>>> 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 54

```
>>> 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 55** 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 56** *Die folgende Programmieranweisung*

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

entspricht zum Beispiel der Formel

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

**Beispiel 57** 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 mitangeben 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 58** 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 59

```
>>> 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_object = 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. Die erlaubten 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

#### 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. 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 60

```
>>> name="kaspar"  
>>> name2 = name.upper()  
>>> name # 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 61

```
>>> 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

---

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

geöffneten Dateiobjekt `file_object` auf:

```
file_object.read()
```

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

Die Methode

```
file_object.readline()
```

liest nur eine einzelne Zeile aus der Datei und gibt diese Zeile als Zeichenkette zurück. Beachten Sie, dass jede eingelesene Zeile – ausser der *letzten* – mit einem Zeilenumbruchzeichen endet ("`\n`").

Um Zeilen einzeln aus einer Datei zu lesen (um diese einzeln zu bearbeiten), können wir bspw. eine `for`- Schleife programmieren.

**Beispiel 62** *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
```

*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 "`\n`" 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

file = open("webpages.txt", "r")

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

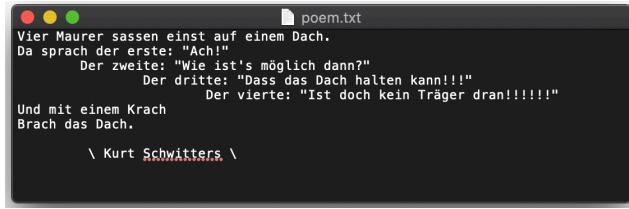


Abbildung 4.1: Die erzeugte Datei poem.txt.

```

"""
write_poem.py
"""

file = open("poem.txt", "w")

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. Der einfachste Weg hierzu bietet die eingebaute Funktion `str`.

**Beispiel 64** In folgendem Programm werden Zeichenketten mit (in Zeichenketten konvertierten) Zahlen konkateniert und ausgegeben:

```

"""
info.py
"""

file = open("info.txt", "w")

age = 43
height = 1.76
file.write("Ich bin " + str(age) +
          " Jahre alt und " + str(height) + "m gross.");

```

Die direkte Konkatenation von Zahlen und Zeichenketten ist in Python übrigens 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.

Eine andere Möglichkeit, Zahlen in Zeichenketten aufzunehmen, bietet die Methode `st.format(args)`, die Sie auf Zeichenketten aufrufen können: Die Zeichenkette `st`, auf der diese Methode aufgerufen wird, kann Text und sogenannte *Ersatzfelder* enthalten. Ein Ersatzfeld besteht aus einem Index `i` und geschweiften Klammern `"{i}"`. Die Methode `st.format` gibt eine Kopie der Zeichenkette `st` zurück, bei der jedes Ersatzfeld durch die Werte der entsprechenden Argumente `args` ersetzt wird.

**Beispiel 65** In folgendem Beispiel werden die Ersatzfelder `"{0}"` und `"{1}"` mit den Werten aus `age` und `height` ersetzt:

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

file = open("info.txt", "w")

age = 43
height = 1.76
file.write("Ich bin {0} Jahre alt und {1} m gross.".format(age, height))
```

Natürlich können Sie das Lesen und das Schreiben aus Dateien kombinieren. Das heisst, wir können aus einer Eingabedatei eine Ausgabedatei erzeugen.



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

**Beispiel 66** In folgendem Modul laden wir numerische Daten aus einer Datei `data.txt` (siehe Abb. 4.2). Aus jeder Zeile `line` aus `data.txt` erzeugen wir mit Hilfe der Methode `split` eine Liste aus einzelnen Elementen. Als nächstes erzeugen wir eine leere Liste `i_list`. Mit einer `for`-Schleife iterieren wir über die einzelnen Elemente der `s_list`, konvertieren diese in ganze Zahlen und fügen sie in die `i_list` ein. Schliesslich berechnen wir den Mittelwert der `i_list` und schreiben diesen inklusive der Zeilennummer in die Ausgabedatei.

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

import statistics

in_file = open("data.txt", "r")
out_file = open("result.txt", "w")

line_num = 1

for line in in_file:
    s_list = line.split(" ") # erzeugt eine Liste mit Zeichenketten
    i_list = []
    for element in s_list:
        i_list.append(int(element))
    mean = round(statistics.mean(i_list), 2)
    out_file.write("Mittelwert aus Zeile {} = {} \n".format(line_num, mean))
    line_num += 1
```

# 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 67** *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 zurückerhalten).

## 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 68** *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 69** 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()
```

Wenn Sie nur eine einzelne Funktion eines Moduls importieren wollen, können Sie dies folgendermassen tun:

```
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 70** *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

---

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

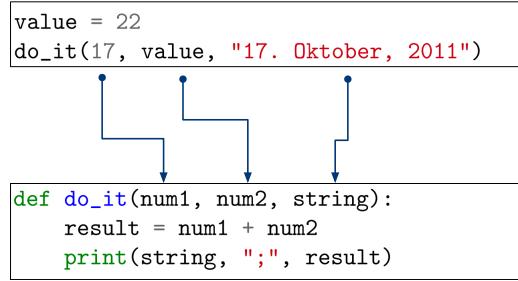


Abbildung 5.1: Parameterübergabe durch einen Funktionsaufruf: Alle Argumente werden in die formalen Parameter kopiert.

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

wobei `val` dem formalen Parameter entspricht.

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 71** 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 de-

finieren, der im Prinzip immer das Gleiche tut, dabei aber die Anweisungen bei jedem Aufruf mit unterschiedlichen Operanden ausführt.

**Beispiel 72** 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 73** 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 Parameter keine Argumente übergeben, werden nun einfach die Standardwerte verwendet.

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

def print_student(name, major="Informatik", semester=1):
    print(name, "studiert", 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 Ar-

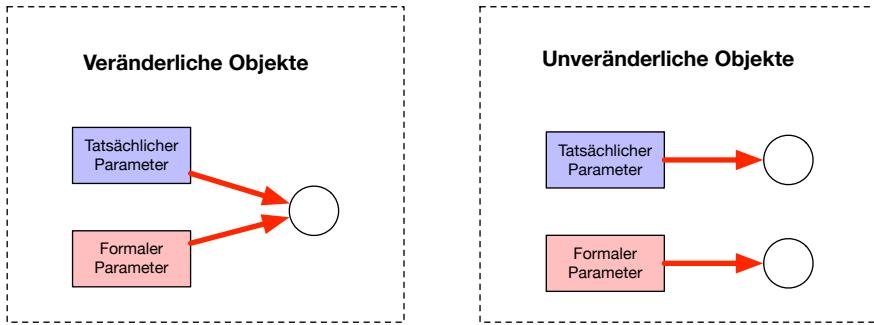


Abbildung 5.2: Tatsächliche und formale Parameter: Unterschied zwischen veränderlichen und unveränderlichen Objekten.

gument zu definieren. Die Anweisung:

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

führt somit zur Ausgabe:

*John studiert Informatik im 5. Semester*

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 74** 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]
```

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

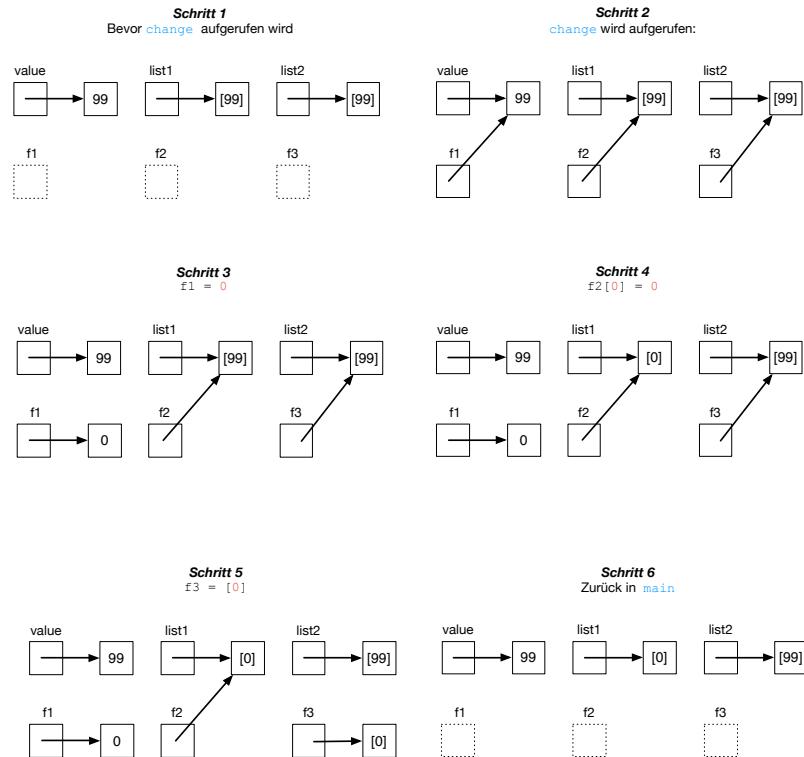


Abbildung 5.3: Veränderung der tatsächlichen und formalen Parameter.

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

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 Kon-

Fluss 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



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

abgearbeitet sind

Tatsächlich geben in Python auch Funktionen ohne Rückgabe einen 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 75** 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:

```
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.

---

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

In Python kann eine Funktion mehrere Werte gleichzeitig zurückgeben, indem nach der `return`-Anweisung mehrere durch Komma getrennte 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 76** 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):
    sum_total = op1 + op2
    difference = op1 - op2
    product = op1 * op2
    quotient = op1 / op2
    return sum_total, difference, product, quotient

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 unweigerlich 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 igtay 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):
    return word.startswith("bl") or word.startswith("sc") or \
           word.startswith("br") or word.startswith("sh") or \
```

```
word.startswith("ch") or word.startswith("sk") or \
word.startswith("cl") or word.startswith("sl") or \
word.startswith("cr") or word.startswith("sn") or \
word.startswith("dr") or word.startswith("sm") or \
word.startswith("dw") or word.startswith("sp") or \
word.startswith("fl") or word.startswith("sq") or \
word.startswith("fr") or word.startswith("st") or \
word.startswith("gl") or word.startswith("sw") or \
word.startswith("gr") or word.startswith("th") or \
word.startswith("kl") or word.startswith("tr") or \
word.startswith("ph") or word.startswith("tw") or \
word.startswith("pl") or word.startswith("wh") or \
word.startswith("pr") or word.startswith("wr")
```

# 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]
```

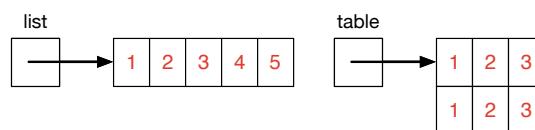


Abbildung 6.1: Ein- vs. zweidimensionale Listen.

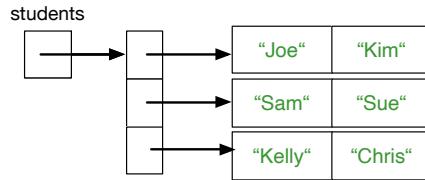


Abbildung 6.2: Eine zweidimensionale Liste ist eine Liste aus Listen.

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 77** 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 78** 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 79** 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 80** 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 81** 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 Punktezahl 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 82

```
>>> 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 83

```
>>> 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 84

```
>>> 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 85

```

>>> 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 86

```

>>> 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 87

```
>>> 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 88

```
>>> 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 Schlüsselwort `in` verwenden. Beachten Sie, dass dieser Test auf Mitgliedschaft nur für die Schlüssel aber nicht für die Werte gilt.

### Beispiel 89

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

Mit den Methoden `keys` bzw. `values` können alle Schlüssel bzw. alle Werte eines Wörterbuches ausgelesen werden. Die Rückgaben sind dabei Objekte vom Typ `dict_keys` bzw. `dict_values` – mit Hilfe der Funktion `list` können diese in Listen umgewandelt werden:

### Beispiel 90

```
>>> squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}  
>>> keys = list(squares.keys())  
>>> keys  
[1, 2, 3, 4, 5]  
>>> values = list(squares.values())  
>>> values  
[1, 4, 9, 16, 25]
```

**Beispiel 91** Gegeben sei die Datei `capitals.csv` mit folgendem Inhalt:

*England;London*

*Deutschland;Berlin*

*Norwegen;Oslo*

*USA;Washington DC*

*Schweiz;Bern*

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`. Die Schlüssel sind dabei jeweils die Namen der Länder und die Werte die zugehörigen Hauptstädte. Die Funktion `impute_country` erstellt im als Parameter übergebenen Wörterbuch `country_capital_dict` einen neuen Eintrag für das Land `country`.

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

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

# Fügt in das Land-Hauptstadt Wörterbuch einen neuen Eintrag ein
def impute_country(country_capital_dict, country):
    capital = input("Hauptstadt von {} eingeben: ".format(country))
    country_capital_dict[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()
another = "y"

while another == "y":
    country = input("Land eingeben: ")
    capital = country_capital_dict.get(country)
    if capital != None:
        print("Hauptstadt: ", capital)
    else:
        decision = input("Land noch nicht erfasst. Land erfassen? (y/n) ")
        if decision == "y":
            data_manager.impute_country(country_capital_dict, country)
    another = input("Weitere Abfrage? (y/n): ")
```

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 92

```
>>> 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:

---

<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.

- Tupel sind *unveränderlich* und enthalten in der Regel eine *heterogene* Folge von Elementen, auf die durch *Entpacken* (siehe weiter unten) zugegriffen wird.
- Listen sind *veränderliche* Objekte, ihre Elemente sind in der Regel *homogen*, 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 93

```
>>> 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!" 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 94

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

Wir können Listen und Tupel kombinieren.

**Beispiel 95** 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.

```
"""  
read_measurements.py  
"""  
  
measurement_list = []  
  
file = open("measurements.txt")  
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)
```

Die Ausgabe des Programmes lautet:

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

## 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. Zu den grundlegenden Verwendungszwecken gehören die Prüfung der Mitgliedschaft (mit dem Schlüsselwort `in`) und die automatische Eliminierung doppelter Einträge.

### Beispiel 96

```
>>> 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
>>> '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 97

```
>>> my_set = {1,3}  
>>> 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 mit der Methode `intersection` und die Differenz mit dem Operator `-` oder mit der Methode `difference` durchgeführt.

### Beispiel 98

```
>>> A = {1, 2, 3, 4, 5}  
>>> B = {4, 5, 6, 7, 8}  
>>> C = A | B
```

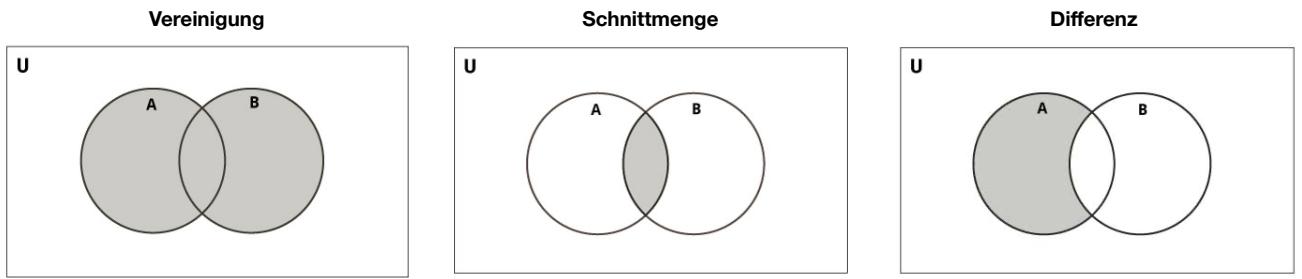


Abbildung 6.3: Vereinigung, Schnittmenge und die Differenz von Mengen  $A$  und  $B$  illustriert.

```

>>> 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}

```

# Kapitel 7

## Externe Pakete und Projekte

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.

In diesem Kapitel betrachten wir exemplarisch die folgenden drei Projekte<sup>2</sup>:

- Pandas: Funktionen zur Datenanalyse
- Matplotlib: Funktionen zur Erzeugung von Grafiken
- SciPy: Funktionen für wissenschaftliche Berechnungen

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

---

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

<sup>2</sup>Beachten Sie, dass Sie auf PyPi sehr viele weitere spannende und nützliche Projekte finden können.

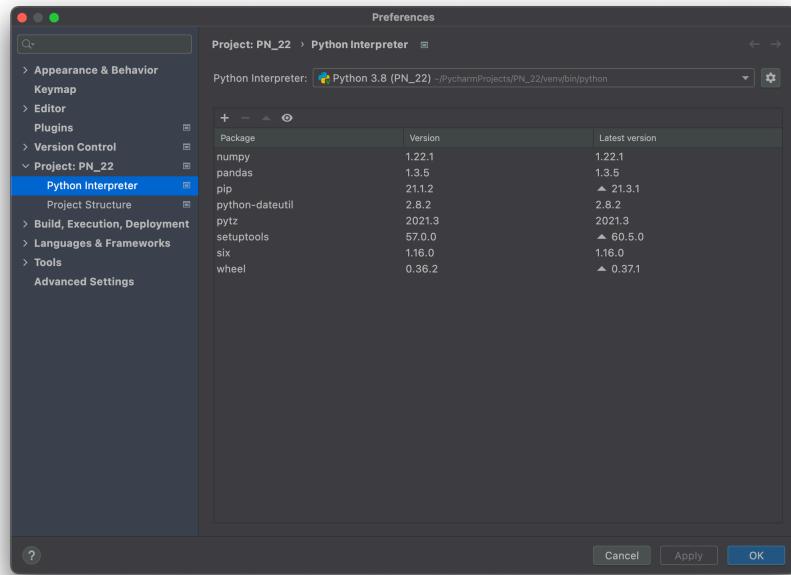


Abbildung 7.1: In der IDE PyCharm nach einem Paket suchen und dieses installieren.

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**.

## 7.1 Das Paket `pandas`

Das Paket `pandas` bietet leistungsfähige Funktionen und Datenstrukturen zur Datenanalyse und Datenmanipulation. Um das Paket `pandas` zu laden und mit ihm zu arbeiten, importieren Sie das Paket (der unter Programmiererinnen vereinbarte Alias für `pandas` ist `pd`).

```
import pandas as pd
```

`pandas` stellt zum Beispiel die Funktion `read_csv()` zur Verfügung, um Daten, die als csv-Datei gespeichert sind, in einen sogenannten

Abbildung 7.2: Datei mit Passagierdaten der Titanic.

`DataFrame` einzulesen. Ein `DataFrame` ist eine zweidimensionale Datenstruktur, die Daten verschiedener Typen (einschliesslich Zeichenketten, Ganzzahlen und Gleitkommazahlen) in Spalten speichern kann. Ein `DataFrame` ist vergleichbar mit einem Tabellenblatt einer Tabellenkalkulation.

Wenn ein `DataFrame` mit `print` ausgegeben wird, werden standardmässig die ersten und letzten 5 Zeilen der Daten angezeigt. Um die ersten  $N$  Zeilen eines `DataFrame` anzuzeigen, verwenden wir die Methode `head()` mit der gewünschten Anzahl Zeilen als Argument.

**Beispiel 99** Für das folgende Beispiel laden wir eine Datei `titanic.csv` (siehe Abb. 7.2) mit `pandas`. Die Datei `titanic.csv` enthält Angaben von 891 Passagieren der Titanic – unter anderem ist das Überleben des Unglücks, die Reiseklasse oder der Ticketpreis erfasst. Wir geben zuerst das ganze und danach die ersten 3 Zeilen des `DataFrame` Objektes aus.

```

"""
pandas_demo.py
"""

import pandas as pd

titanic = pd.read_csv("titanic.csv")
print(titanic)
print(titanic.head(3))

```

Die Ausgabe lautet:

	<i>PassengerId</i>	<i>Survived</i>	<i>Pclass</i>	...	<i>Fare</i>	<i>Cabin</i>	<i>Embarked</i>
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]

	<i>PassengerId</i>	<i>Survived</i>	<i>Pclass</i>	...	<i>Fare</i>	<i>Cabin</i>	<i>Embarked</i>
0	1	0	3	...	7.2500	NaN	S
1	2	1	1	...	71.2833	C85	C
2	3	1	3	...	7.9250	NaN	S

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 Klammern definieren dabei eine Liste mit Spal-

tennamen, während die äusseren Klammern verwendet werden, um die Daten aus einem `DataFrame` auszuwählen).

**Beispiel 100** *In folgendem Beispiel wählen wir einmal nur die Spalte "Age" und einmal die beiden Spalten ["Age", "Sex"] und geben danach die ersten drei Einträge aus<sup>3</sup>.*

```
""" Fortsetzung des obigen Beispiels """

# Auswahl einzelner Spalten
ages = titanic["Age"] # Series Objekt
ages_sex = titanic[["Age", "Sex"]] # DataFrame Objekt

print(ages.head(3))
print(ages_sex.head(3))
```

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.

Wenn wir nun 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 101** *In folgendem Beispiel rufen wir die Methoden `max()` und `describe()` auf dem Objekt `ages = titanic["Age"]` auf.*

---

<sup>3</sup>Dieses und die weiteren Beispiele führen Beispiel 99 fort. Das Laden der Datei in ein `Dataframe` wird also nicht nochmals gezeigt.

```
""" Fortsetzung des obigen Beispiels """

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
```

Auf einem `DataFrame` Objekt kann man nicht nur Spalten sondern auch Zeilen selektieren. Um Zeilen auszuwählen, definieren wir eine Boolesche Bedingung innerhalb der Auswahlklammern `[]`.

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 102** *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.*

```
""" Fortsetzung des obigen Beispiels """

above_35 = titanic[titanic["Age"] > 35]
class_23 = titanic[(titanic["Pclass"] == 2) | (titanic["Pclass"] == 3)]

print(above_35.head(3))
print(class_23.head(3))
```

Die Ausgabe lautet:

	<i>PassengerId</i>	<i>Survived</i>	<i>Pclass</i>	...	<i>Fare</i>	<i>Cabin</i>	<i>Embarked</i>
1	2	1	1	...	71.2833	C85	C
6	7	0	1	...	51.8625	E46	S
11	12	1	1	...	26.5500	C103	S

[3 rows x 12 columns]

	<i>PassengerId</i>	<i>Survived</i>	<i>Pclass</i>	...	<i>Fare</i>	<i>Cabin</i>	<i>Embarked</i>
0	1	0	3	...	7.250	NaN	S
2	3	1	3	...	7.925	NaN	S
4	5	0	3	...	8.050	NaN	S

[3 rows x 12 columns]

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 103** Mit folgender Auswahl definieren wir ein *Dataframe* Objekt, das den Namen, das Alter und den Status *Survived* aller überlebenden Passagiere unter 10 Jahren anzeigt:

```
""" Fortsetzung des obigen Beispiels """

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

Wir können Berechnungen und Analysen auch auf bestimmte Gruppen einschränken.

**Beispiel 104** 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. Anschliessend wird die Methode *groupby()* auf das Geschlecht angewendet, um eine Gruppe pro Kategorie zu bilden. Schliesslich wird der Durchschnitt der Spalten pro Geschlecht berechnet und zurückgegeben.

```
""" Fortsetzung des obigen Beispiels """

survived_m_f = titanic[["Sex", "Survived"]].groupby("Sex").mean()

print(survived_m_f)
```

Ausgabe:

```
Survived
Sex
female    0.742038
male      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

```
""" Fortsetzung des obigen Beispiels """
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

```
""" Fortsetzung des obigen Beispiels """
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
```

Wir können berechnete `DataFrame` Objekte mit der Methode `to_excel()` jederzeit als Excel Datei speichern (ggf. müssen Sie hierzu noch das Paket `openpyxl` installieren).

**Beispiel 105** *In folgendem Beispiel wird das `DataFrame` `survived_kids` in eine Excel Datei `titanic.xlsx` geschrieben, Der Blattname wird als `"survived_kids"` definiert und durch die Einstellung `index=False` werden die Zeilenindexbezeichnungen nicht im Blatt aufgenommen.*

```
""" Fortsetzung des obigen Beispiels """
survived_kids = titanic.loc[(titanic["Age"] < 10) &
                           (titanic["Survived"] == 1),
                           ["Name", "Age", "Survived"]]

survived_kids.to_excel("titanic.xlsx", sheet_name="survived_kids", index=False)
```

## 7.2 Das Paket `matplotlib`

Das Paket `matplotlib` ist eine Bibliothek zur Erstellung von zweidimensionalen Diagrammen und Grafiken. Das Paket `matplotlib` enthält ein Modul namens `pyplot`, das Sie 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
```

Einige wichtige Funktionen des Moduls `pyplot` sind:

- `plt.plot(x_coords, y_coords[, marker="o"]):`

Erstellt ein Liniendiagramm, das eine Reihe von Punkten mit geraden Linien verbindet. Die Parameter `x_coords`, `y_coords` sind dabei Listen mit *x*- und *y*-Koordinaten.

Optional können Sie den Parameter `marker` definieren. Diese Option erstellt für jeden Datenpunkt eine Markierung (z.B. einen runden Punkt). Einige andere Markierungen:

- `"s"`: Quadrat
  - `"*"`: Stern
  - `"D"`: Diamant
  - `"^"`: Dreieck
- `plt.title("Titel")`, `plt.xlabel("Beschriftung x-Achse")`, `plt.ylabel("Beschriftung y-Achse")` und `plt.grid(True)`: Beschriftet das Diagramm mit einem Titel, beschriftet die  $x$ - und  $y$ -Achsen und zeigt ein Gitter an.
  - `plt.xlim(xmin=0, xmax=10)` und `plt.ylim(ymin=0, ymax=10)`: Ändert die untere und obere Grenze der  $x$ - 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.show()`:  
Zeigt das erstellte Diagramm in einem Fenster an.

Hinweis: Es existieren zahlreiche weitere Funktionen, die wir hier aber nicht näher betrachten.

**Beispiel 106** Das folgende Programm erzeugt das Diagramm in Abb. 7.3:

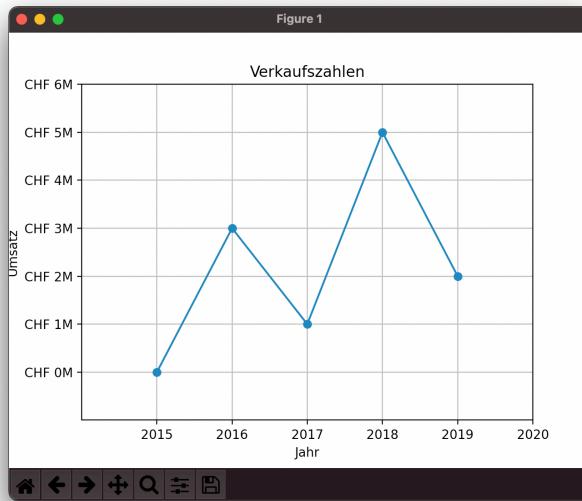


Abbildung 7.3: Erzeugtes Liniendiagramm.

```
"""
matplotlib_example.py
"""

import matplotlib.pyplot as plt

x_values = list(range(5))
y_values = [0, 3, 1, 5, 2]

plt.plot(x_values, y_values, marker="o")

plt.title("Verkaufszahlen")
plt.xlabel("Jahr")
plt.ylabel("Umsatz")

plt.grid(True)
plt.xlim(xmin=-1, xmax=5)
plt.ylim(ymin=-1, ymax=6)

plt.xticks(list(range(6)),
           ["2015", "2016", "2017", "2018", "2019", "2020"])
plt.yticks(list(range(7)),
           ["CHF 0M", "CHF 1M", "CHF 2M", "CHF 3M", "CHF 4M", "CHF 5M", "CHF 6M"])

plt.show()
```

Die Pakete `pandas` und `matplotlib` können miteinander kombiniert

```

air_quality_no2.csv
datetime,station_antwerp,station_paris,station_london
2019-05-07 02:00:00,,,23.0
2019-05-07 03:00:00,,50.5,25.0,19.0
2019-05-07 04:00:00,45.0,27.7,19.0
2019-05-07 05:00:00,,50.4,16.0
2019-05-07 06:00:00,,61.9
2019-05-07 07:00:00,,72.4,26.0
2019-05-07 08:00:00,,77.7,32.0
2019-05-07 09:00:00,,67.9,32.0
2019-05-07 10:00:00,,56.0,28.0
2019-05-07 11:00:00,,34.5,21.0
2019-05-07 12:00:00,,20.1,21.0
2019-05-07 13:00:00,,13.0,18.0
2019-05-07 14:00:00,,10.6,20.0
2019-05-07 15:00:00,,13.2,18.0
2019-05-07 16:00:00,,11.0,20.0
2019-05-07 17:00:00,,11.7,20.0
2019-05-07 18:00:00,,18.2,21.0

```

Abbildung 7.4: Datei mit Luftqualitätsdaten für drei Messstationen in London, Paris und Antwerpen.

werden. Hierzu kann man die Methode `plot()` sowohl auf `Series` als auch auf `DataFrame` Objekten aufrufen.

Für die folgenden Beispiele verwenden wir Luftqualitätsdaten aus der Datei `air_quality_no2.csv` (diese Datei enthält die  $NO_2$  Werte für drei Messstationen in London, Paris und Antwerpen – siehe Abb. 7.4).

**Beispiel 107** Das folgende Programm verwendet `pandas` und `matplotlib`:

```

"""
matplotlib and pandas combined
"""

import pandas as pd
import matplotlib.pyplot as plt

air_quality = pd.read_csv("air_quality_no2.csv", index_col=0, parse_dates=True)
air_quality.plot()
plt.show()

```

Wir öffnen mit der `pandas` Funktion `read_csv` zunächst die Datei `air_quality_no2.csv`<sup>4</sup>. Danach wird mit `air_quality.plot()` das

<sup>4</sup>Hinweis: Die Verwendung der Parameter `index_col` und `parse_dates` führt dazu, dass die erste (0.) Spalte als Index des resultierenden `DataFrame` definiert wird bzw. dass die Daten der Spalte `datetime` in Zeitstempelobjekte konvertiert werden.

Diagramm erstellt und mit der Funktion `plt.show()` angezeigt.

Wird die Methode `plot()` auf einem `DataFrame` aufgerufen, wird standardmäßig ein Liniendiagramm für jede Spalte mit numerischen Daten erzeugt (siehe Abb. 7.5 (a)). Um nur eine bestimmte Spalte als Liniendiagramm darzustellen, verwenden wir die Auswahlklammern von `pandas` in Kombination mit der Methode `plot()` (siehe Abb. 7.5 (b)):

```
air_quality["station_paris"].plot()
```

Abgesehen von der standardmässigen Liniendarstellung bei Verwendung der Methode `plot()` gibt es eine Reihe von Alternativen zur Darstellung von Daten.

Mit folgendem Aufruf erzeugen wir z.B. das Diagramm in Abb. 7.5 (c):

```
air_quality.plot.scatter(x="station_london", y="station_paris", alpha=0.5)
```

Während folgender Aufruf das Diagramm in Abb. 7.5 (d) erzeugt:

```
air_quality.plot.box()
```

Die eingebauten Optionen, die in den `plot`-Methoden verfügbar sind, sind vielfältig. Zum Beispiel kann man mit `subplots=True` getrennte Teilplots für jede Datenspalte erstellen oder mit dem Argument `figsize` kann man die Grösse der Abbildung anpassen (siehe Abb. 7.5 (e)):

```
air_quality.plot.area(figsize=(12, 4), subplots=True)
```



Abbildung 7.5: Mit `pandas` und `matplotlib` lassen sich vielfältige Diagramme erstellen.

## 7.3 Das Paket `scipy`

Das Paket `scipy` bietet Funktionen zur Optimierung, Integration, Interpolation, algebraische Gleichungen, Differentialgleichungen, Statistik und viele andere Klassen von Problemen. Wir zeigen hier nur drei von zahlreichen Möglichkeiten.

### 7.3.1 Integrieren

Mit der Funktion `quad` kann man das Integral einer Funktion zwischen zwei Punkten berechnen. Das erste Argument von `quad` ist ein "aufrufbares" Python-Objekt (d.h. z.B. eine Funktion). Die nächsten beiden Argumente sind die Grenzen der Integration. Der Rückgabewert ist ein Tupel, wobei das erste Element dem geschätzten Wert des Integrals und das zweite Element der Obergrenze des Fehlers entspricht.

**Beispiel 108** Nehmen wir zum Beispiel an, Sie wollen das Integral der Funktion  $f(x) = x^3 + 3x^2$  im Intervall  $[1, 4]$  berechnen:

$$\int_1^4 x^3 + 3x^2 dx$$

In folgendem Modul definieren wir zunächst die Funktion  $f$  als eigene Funktion  $f(x)$ .

```
"""
integrate.py
"""

from scipy.integrate import quad

def f(x):
    return x**3 + 3*x**2

I = quad(f, 1, 4)
print(I)
```

Der Funktion `quad` aus dem Paket `scipy.integrate` kann nun diese Funktion als Parameter übergeben werden (zusammen mit den numerischen Grenzen der Integration). In diesem Beispiel lautet die Ausgabe von `I`:

`(126.75, 1.407207683712386e-12)`

### 7.3.2 Optimieren

Wir betrachten das Problem der Auswahl von Schülern für eine Lagen-Staffel im Schwimmen. Wir haben eine Tabelle mit Zeiten für jeden Schwimmstil von fünf Schülern (A - E):

	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>E</b>
<b>backstroke</b>	43.5	45.5	43.4	46.5	46.3
<b>breaststroke</b>	47.1	42.1	39.1	44.1	47.8
<b>butterfly</b>	48.4	49.6	42.1	44.5	50.4
<b>freestyle</b>	38.2	36.8	43.2	41.2	37.2

Wir müssen für jeden der vier Schwimmstile einen Schüler so auswählen, dass die Gesamtzeit der Staffel minimiert wird<sup>5</sup>. Dies ist ein typisches lineares Summenzuordnungsproblem.

Das lineare Summenzuordnungsproblem ist eines der bekanntesten kombinatorischen Optimierungsprobleme. Gegeben eine sogenannte *Kostenmatrix*, besteht das Problem darin,

- genau ein Element aus jeder Zeile zu wählen
- ohne mehr als ein Element aus einer Spalte zu wählen
- so dass die Summe der gewählten Elemente minimiert wird

---

<sup>5</sup>Beachten Sie, dass z.B. Schüler C der beste Schwimmer sowohl im Brustschwimmen als auch im Schmetterlingsstil ist. Wir können C aber nicht beiden Schwimmstilen zuordnen.

Der erste Schritt besteht darin, die Kostenmatrix als zweidimensionale Liste zu definieren. Die Eingabe an die Funktion `linear_sum_assignment` aus dem Paket `scipy.optimize` ist dann diese Kostenmatrix und die Rückgabe besteht aus zwei Listen mit den optimal zugeordneten Zeilen- und Spaltenindizes.

```
"""
lsap.py
"""

from scipy.optimize import linear_sum_assignment

styles = ["backstroke", "breaststroke", "butterfly", "freestyle"]
students = ["A", "B", "C", "D", "E"]

cost = [[43.5, 45.5, 43.4, 46.5, 46.3],
        [47.1, 42.1, 39.1, 44.1, 47.8],
        [48.4, 49.6, 42.1, 44.5, 50.4],
        [38.2, 36.8, 43.2, 41.2, 37.2]]

row_indices, col_indices = linear_sum_assignment(cost)

optimal_time = 0
for i in range(len(row_indices)):
    print(students[col_indices[i]], "<->", styles[row_indices[i]])
    optimal_time += cost[row_indices[i]][col_indices[i]]

print("Optimale Zeit:", round(optimal_time, 1))
```

Die Ausgabe des Programmes lautet:

```
A <-> backstroke
C <-> breaststroke
D <-> butterfly
B <-> freestyle
Optimale Zeit: 163.9
```

### 7.3.3 Gruppieren

Ein *Clustering* ist eine Gruppierung von Datenobjekten in  $K$  disjunkte Teilmengen – die *Cluster*. Das Ziel ist, dass die Objekte innerhalb eines Clusters einander ähnlich sind und sich gleichzeitig von den Objekten in anderen Clustern unterscheiden.

Im Prinzip könnte jede Clustering-Aufgabe so gelöst werden, dass für eine gegebene Menge von  $N$  Datenobjekten *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.

In diesem Abschnitt betrachten wir einen Algorithmus, der das Clustering von Daten effizient durchführen kann – das *agglomerative hierarchische Clustering*. Die Idee dieses Verfahrens ist, mit den Datenobjekten als einzelne Cluster zu starten und in jedem Schritt das nächstgelegene Paar von Clustern miteinander zu verschmelzen (dies erfordert die Definition einer Clusterdistanz, damit in jedem Schritt entschieden werden kann, welche zwei Cluster als nächstes verschmolzen werden sollen).

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 wurden.

Ein Dendrogramm kann einerseits direkt als Ergebnis des Clustering-

iris.csv	
"	sepal.length", "sepal.width", "petal.length", "petal.width", "variety"
5.1	3.5,1.4,"Setosa"
4.9	3.0,1.4,2,"Setosa"
4.7	3.2,1.3,2,"Setosa"
4.6	3.1,1.5,2,"Setosa"
5.3	3.6,1.4,2,"Setosa"
5.4	3.9,1.7,2,"Setosa"
4.6	3.4,1.4,2,"Setosa"
5.0	3.4,1.4,2,"Setosa"
4.4	2.9,1.4,2,"Setosa"
4.9	3.1,1.5,2,"Setosa"
5.4	3.7,1.5,2,"Setosa"
4.8	3.4,1.6,2,"Setosa"
4.8	3.1,1.4,1,"Setosa"

Abbildung 7.6: Die Datei `iris.csv` enthält Länge und Breite von Blütenblättern drei verschiedener Iris-Arten.

Prozesses verwendet werden, da dieses einen visuellen Eindruck der Struktur der Daten vermittelt. Andererseits kann der Baum auch verwendet werden, um die Daten in eine bestimmte Anzahl von Clustern zu zerlegen. Dazu muss im Dendrogramm lediglich ein horizontaler Schnitt zwischen zwei Ebenen durchgeführt werden.

In folgendem Modul laden wir die Datei `iris.csv`, welche die gemessene Länge und Breite von Blütenblättern drei verschiedener Iris-Arten enthält (siehe Abb. 7.6).

```
"""
cluster.py
"""

import matplotlib.pyplot as plt
import pandas as pd
from scipy.cluster.hierarchy import dendrogram
from scipy.cluster.hierarchy import complete

data = pd.read_csv('iris.csv')
data = data[['sepal_length", "sepal_width", "petal_length", "petal_width"]]

linkage_array = complete(data)
dendrogram(linkage_array, color_threshold=3.5)

plt.xlabel("Data Object")
plt.ylabel("Cluster Distance")
plt.show()
```

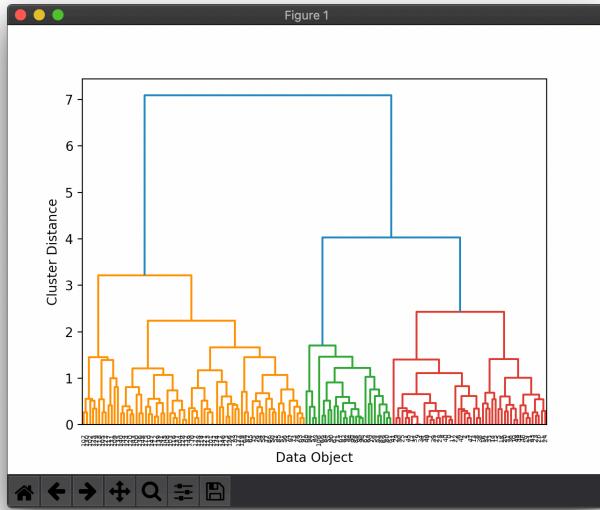


Abbildung 7.7: Ein Dendrogramm, das die struktur der Daten zeigt.

Mit der Funktion `complete` erzeugen wir ein Clustering unter Verwendung der sogenannten *Complete-Linkage Distanz* (es existieren weitere Möglichkeiten zur Definition einer Clusterdistanz).

Die Rückgabe der Funktion definiert die Distanzen, auf denen zwei Cluster miteinander verschmolzen werden und wird als Parameter an die Funktion `dendrogram` übergeben (um ein Dendrogramm zu erzeugen). Mit dem 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`).

Die Ausgabe ist in Abb. 7.7 zu sehen. Das Dendrogramm offenbart die Struktur der Daten und man kann die drei Iris-Arten recht gut erkennen.