



Programmieren in Python

Das vorliegende Kapitel beschreibt einleitend die Grundlagen der Programmiersprache Python. Dabei besteht nicht der Anspruch, eine umfassende und vollständige Einführung der Konzepte von Python zu vermitteln. Sie sollen vielmehr in die Lage versetzt werden, die im Buch gewonnenen Erkenntnisse, welche an der Programmiersprache Java vermittelt wurden, auch mit der Programmiersprache Python umsetzen zu können. Zudem wollen wir vermitteln, dass Python durchaus in bestimmten Situationen einige Vorteile gegenüber Java oder C++ bietet.

Nach einer allgemeinen Einführung in die Programmiersprache im Abschnitt 13.1 und einer Erklärung zur Entwicklung und Ausführung von Python am Beispiel eines ersten einfachen Programms (s. Abschnitt 13.2), schauen wir uns einige ausgewählte und bedeutsame Konzepte von Python an. Thematisiert werden Konzepte im Zusammenhang mit Funktionen (Abschnitt 13.3), Mechanismen der umfassenden und effizienten eingebauten Sammeldatentypen (Abschnitt 13.4) sowie Grundlagen zur objektorientierten Programmierung (Abschnitt 13.5).

13.1 Einführung in Python

Die erste Version der Programmiersprache Python wurde im Jahr 1991 von Guido van Rossum entwickelt. Dabei handelte es sich zunächst um eine reine Interpretersprache bzw. Skriptsprache. Was das bedeutet, schauen wir uns im Abschnitt 13.2.1 noch genauer an. Nach der Verfügbarkeit von Python in der Version 2 im Jahr 2000, welche aber nicht mehr unterstützt wird, liegt aktuell die Version 3 vor. Python gilt als für Anfängerinnen und Anfänger leicht erlernbar. Vergleichbar zu Java ist auch Python plattformübergreifend verfügbar und stellt ebenfalls umfangreiche Klassenbibliotheken für verschiedenste Fragestellungen zur Verfügung. Zudem hat sich Python beispielsweise zu einer Referenzsprache im Bereich des Maschinellen Lernens¹ entwickelt. All dies hat zu seiner Popularität und weiten Verbreitung beigetragen.

¹ Der Bereich des Maschinellen Lernens ist ein aktuelles Forschungsgebiet mit einer ebenfalls ausgeprägten praktischen Bedeutung. Es umfasst Konzepte und Verfahren zur (un)überwachten Klassifikation von allgemeinen und spezifischen Daten, insbesondere bildbasierten Daten, teilweise unter Beachtung der Zeit und zu verschiedenen Fragen aus den Datenwissenschaften (Data Science) [12, 13, 14].

Prinzipielle Übereinstimmungen	
arithmetische und logische Datentypen	prinzipiell identisch, zusätzlich „complex“ bei Python
bedingte Anweisungen, Schleifen, Verbundanweisungen (Blöcke)	prinzipiell identisch
objektorientierte Programmierung	benutzerdefinierte Klassen, Attribute, Methoden, Vererbung, jedoch teilweise andere Realisierung
Speicherverwaltung für Objekte	Objektverwaltung über implizite Referenzen, Speicherfreigabe automatisiert, bei Python zusätzlich Destruktoren verfügbar
Ausführung	Grundkonzept: Übersetzung des Programmcodes in plattformunabhängigen, interpretierend ausgeführten Bytecode
Deutliche Unterschiede	
interaktive Ausführung	Python ermöglicht, anders als Java, neben der Übersetzung auch die Ausführung von interaktiv eingegebenen Python-Anweisungen
Variablendeklaration	Java: statische Typisierung Python: dynamische Typisierung
Eingebaute Sammeltypen:	Java: Arrays, Zeichenketten Python: Sequenz-, Mengen-, Mapping-Typen
Funktionen	Python: Funktionen unabhängig von Klassen

Abbildung 13.1: Übersicht zum Vergleich ausgewählter Konzepte zwischen Python und Java

Im Folgenden wollen wir uns zunächst die grundlegenden Konzepte von Python im Rahmen einer vergleichenden Einführung mit den Java-Kenntnissen ansehen. Hierzu vermittelt die Abbildung 13.1 eine vergleichende Übersicht zu den prinzipiellen Übereinstimmungen respektive den deutlichen Unterschieden zwischen Python und Java. Die Ausführungen in den nachfolgenden Abschnitten werden diese Ähnlichkeiten und Unterschiede von Python und Java ausführlicher erklären.

Als anschauliches Beispiel soll die bereits bekannte Suche des Minimums in einer Zahlenfolge dienen. Der Quellcode 13.1 zeigt hierzu noch einmal das Java-Programm aus der Einführung in die Programmierung (Abschnitt 4.4). Der Quellcode 13.2 präsentiert ein Python-Programm mit der gleichen Funktionalität².

² Es sei hierzu noch angemerkt, dass sich die Minimumsuche durch spezielle Funktionalitäten der beiden Programmiersprachen sicherlich noch effizienter realisieren ließe. Im Sinne der didaktischen Vergleichbarkeit belassen wir es aber bei diesen Lösungen.

Quellcode 13.1: Java-Programm zur Minimumsuche

```

01: class ProgrammMinSuche{
02:     public static void main(String[] args){
03:         int[] a = {11,7,8,3,15,13,9,19,18,10,4};
04:         int merker = a[0]; // damit hat merker den Wert 11
05:         int i = 1;
06:         int n = a.length; // Laenge der Folge a
07:         while (i < n){
08:             if (a[i]<merker)
09:                 merker = a[i];
10:             i = i+1;
11:         }
12:         System.out.println(merker); // druckt das Ergebnis
13:     }
14: }

```

Quellcode 13.2: Python-Programm zur Minimumsuche

```

01: a = [11,7,8,3,15,13,9,19,18,10,4]
02: merker = a[0] # damit hat merker den Wert 11
03: i = 1
04: n = len(a)    # Laenge der Folge a
05: while i < n:
06:     if a[i] < merker:
07:         merker = a[i]
08:         i = i+1
09: print('Minimum:', merker) # druckt das Ergebnis

```

13.1.1 Anweisungen

Ein Python-Programm besteht wie ein Java-Programm aus einer Folge von Anweisungen. Der Vergleich der beiden Quellcodes 13.1 und 13.2 zeigt, dass die Schreibweise eines Python-Programms einfacher als die des Java-Programms wirkt. Sie ähnelt der Pseudocode-Notation von Algorithmen in Kapitel 3. Anstelle eines Semikolons sind aufeinanderfolgende Anweisungen durch einen Zeilenwechsel getrennt. Kommentare in Python beginnen mit dem Rautezeichen „#“ und erstrecken sich bis zum Ende der physischen Zeile.

Im Unterschied zu Java-Programmen ist kein Klassenrahmen und die Platzierung in der statischen `main`-Methode notwendig. Dies ermöglicht die Bereitstellung einer Python-Programmierungsumgebung, in welcher die einzelnen Anweisungen, analog zu einem Taschenrechner, interaktiv eingegeben werden können und welche das Resultat sofort anzeigt. Dieses Prinzip der interaktiven Ausführung von Python-Programmen wird im Abschnitt 13.2.2 eingehender besprochen. Zudem beschreibt der Abschnitt 13.2.3 die zweite Möglichkeit, Programme als Ganzes zur Ausführung zu bringen. Dies ist vergleichbar zu Java oder C++. Ein entsprechendes Python-Programm ist dann in einer oder mehreren Dateien gespeichert, welche Module genannt werden.

Analog zu Java können anstelle von einfachen Anweisungen auch Blöcke, d.h. Folgen von Anweisungen, verwendet werden. In Java sind Blöcke durch geschweifte Klammern begrenzt (Zeilen 8 bis 10 im Quellcode 13.1), wohingegen sie in Python durch linksausgerichtete Einrückung definiert werden (Zeilen 6 bis 8 im Quellcode 13.2).

Python verfügt wie Java über bedingte Anweisungen und Schleifen, insbesondere die `while`-Schleife zur Angabe des Programmablaufs. Die Konzepte unterscheiden sich in den beiden Sprachen leicht in der Notation der Bedingung. In Java steht die Bedingung in runden Klammern (Zeile 7 bzw. 8 im Quellcode 13.1), in Python wird sie ohne Klammern, aber gefolgt von einem Doppelpunkt, geschrieben (Zeilen 5 bzw. 6 im Quellcode 13.2).

Ein weiteres Hilfsmittel zur Formulierung von Anweisungen sind Funktionen (s. Kapitel 5). Funktionen werden in Java im Rahmen des Klassenkonzepts durch statische Methoden bereitgestellt (z.B. `main` in Zeile 2 im Quellcode 13.1). Python bietet im Unterschied dazu das Konzept der Funktion unabhängig von Klassen an (s. Abschnitt 13.3).

13.1.2 Variablen

Python hat wie Java Variablen, die ebenso durch Bezeichner repräsentiert werden (siehe z.B. `a`, `merker`, `i` und `n` in den beiden Quellcodes). Die Zuweisung von Werten wird in Python auch durch das Gleichheitszeichen „`=`“ notiert und der Vergleich auf Gleichheit durch „`==`“.

Ein wesentlicher Unterschied von Python zu Java ist, dass eine Variable durch ihre erste Verwendung deklariert wird. Eine explizite Angabe ihres Datentyps ist dabei nicht notwendig, der Datentyp des zugewiesenen Elements legt den Datentyp der Variablen fest. Dieses Vorgehen wird „dynamische Typisierung“ genannt. In Java ist hingegen vor der Verwendung oder bei der ersten Verwendung eine Variablendeklaration unter Angabe des Datentyps erforderlich (Zeile 3 bis 6 im Quellcode 13.1). Hierbei handelt es sich um eine „statische Typisierung“. Die dynamische Typisierung ist von Vorteil bei der erwähnten interaktiven Programmierung von Python (s. Abschnitt 13.2.2). Hingegen vermindert die statische Typisierung die Fehlerhäufigkeit in größeren Programmen oder Systemen.

13.1.3 Datentypen

Python stellt numerische, d.h. arithmetische und logische Datentypen, und die Möglichkeit zur Formulierung entsprechender Ausdrücke im Prinzip wie Java bereit. Es gibt die bekannten Datentypen `bool`, `int`, `long` und `float`, aber auch einen Datentyp `complex` für komplexe Zahlen der Mathematik. Ein Beispiel für eine komplexe Zahl ist `0 + 1.5j`. Dabei ist `0` der Realteil und `1.5` der Imaginärteil. Das Symbol `j` markiert den Imaginärteil. Die arithmetischen und booleschen Operatoren zur Formulierung von Ausdrücken sind im Wesentlichen wie bei Java, zusätzlich gibt es einen Potenzoperator „`**`“ für `int` und `float`. In konzeptioneller Hinsicht unterscheiden sich die numerischen Datentypen jedoch dadurch, dass es sich bei Python, im Unterschied zu Java, durchgängig um Klassentypen handelt, z.B. `class bool`, `class int`.

Der Java-Quellcode 13.1 speichert die Zahlen, deren Minimum zu bestimmen ist, in einem Array `a` (Zeile 3). In Python (vgl. Quellcode 13.2) wird hierfür eine Liste `a` (Zeile 1) verwendet.

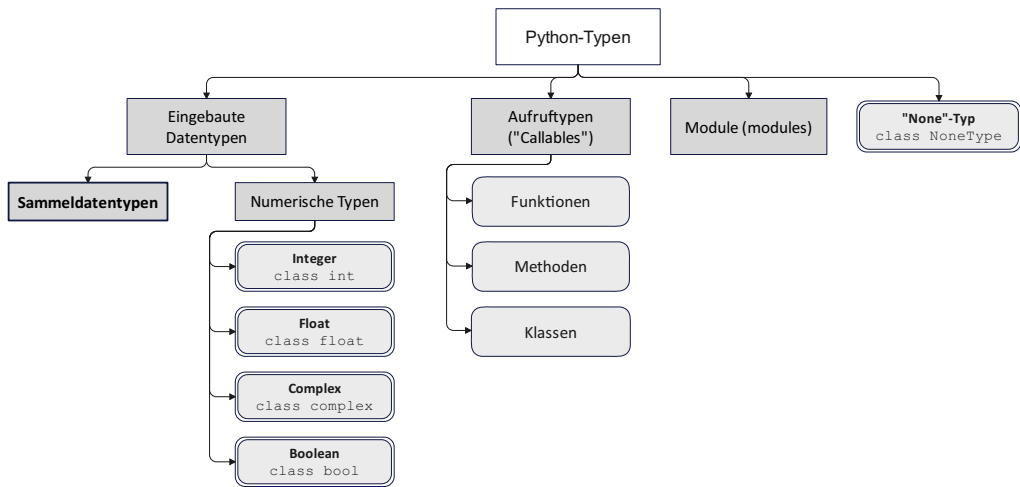


Abbildung 13.2: Schematische Übersicht zum Typkonzept der Programmiersprache Python.

Listen von Python bieten eine erheblich erweiterte Funktionalität gegenüber den Arrays von Java. In Python sind darüber hinaus noch weitere sogenannte Sammeltypen in die Sprache integriert, welche die Programmierung erleichtern und in anderen Programmiersprachen, potentiell weniger effizient realisiert, in Programmbibliotheken verfügbar sind. Zu den eingebauten Sammeltypen gehören auch Zeichenketten (Strings), welche in Java ebenfalls ein Bestandteil der Sprache sind. Auf Sammeldatentypen geht der Abschnitt 13.4 weitergehend ein. Im Zusammenhang mit Sammeldatentypen wird das `for`-Schleifen-Konstrukt bedeutungsvoll, das in der Programmier-einführung dieses Buches vermieden wurde. Python erweitert `for`-Schleifen, die sonst meist als „Zählschleifen“ eingesetzt werden, zum komfortablen iterativen Durchlauf durch die Elemente von iterierbaren Sammeldatentypen.

Python ist wie Java für die objektorientierte Programmierung ausgelegt. Es bietet die Möglichkeit zur Deklaration und Instanziierung von Klassen, wodurch die Deklaration neuer Datentypen ermöglicht wird. Klassen verfügen über Methoden, die durch Funktionen in Klassen realisiert sind und es gibt die Möglichkeit der Vererbung. Anders als in Java wird ein Objekt durch Aufruf seiner Klasse, analog zum Aufruf einer Funktion, instantiiert. Dabei ist eine Initialisierung durch eine spezielle Methode möglich. Vergleichbar zu Java werden Objekte über Referenzen gehandhabt. Alle eingebauten Datentypen sind Klassen, was deren Erweiterung durch Vererbung erlaubt. Der Typ eines Objekts kann zur Laufzeit des Programms ermittelt werden und ist selbst ein Objekt. Ferner können die Attribute eines Objekts herausgefunden werden. Beides kann im Rahmen der dynamischen Typisierung hilfreich sein. Abschnitt 13.5 führt in die objektorientierte Programmierung mit Python ein.

Die Abbildung 13.2 fasst die Typstruktur von Python zusammen. Neben den erwähnten eingebauten numerischen Datentypen gibt es mehrere eingebauten Sammeltypen, die im Abschnitt 13.4

näher besprochen werden. Hinzu kommen aufrufbare Typen (engl. *callable*s), zu denen Funktionen, Methoden und Klassen gehören. Auch die aufrufbaren Typen bilden jeweils eine Klasse. So ist beispielsweise eine konkrete Funktion ein Objekt der Klasse der Funktionen. Der „None“-Typ `NoneType` ist ein Platzhalter für ein Python-Objekt in Situationen, in welchen ein Python-Objekt zu verwenden ist, aber kein konkretes Python-Objekt genutzt werden soll. Dies ist ähnlich zu „void“ oder „null“ in Java. Schließlich werden noch Module zu den Python-Typen gezählt.

Python stellt vergleichbar zu Java Möglichkeiten zur impliziten und expliziten Typkonvertierung zur Verfügung. Beispielsweise findet eine implizite Typkonvertierung bei der Multiplikation einer komplexen Zahl mit einer ganzen Zahl statt. Dabei wird die ganze Zahl zur Durchführung der Multiplikation für komplexe Zahlen in eine komplexe Zahl konvertiert. Die Syntax von Python für die explizite Typkonvertierung, `Typ(var)`, ähnelt jener von Java, `(Typ) var`. Dabei steht `var` für die Variable, welche der Typkonvertierung unterworfen wird und `Typ` für den neuen Typ.

Aufgabe 13.1:


Gegeben sei die nachfolgende Anweisungsfolge:

```
>>> a = 1
>>> b = a
>>> print(b)
>>> a=2
>>> print(b)
```

Welche Ausgaben resultieren durch die Anweisungsfolge? Begründen Sie das Zustandekommen.

13.2 Ausführung von Python-Programmen

Die Python-Programme innerhalb des Buches wurden mit der Version 3.10 erstellt. Sofern eine andere Version auf dem Rechner installiert ist, sollte dies im Normalfall aber auch kein Problem sein, da bei der Entwicklung von Python auf eine gute Kompatibilität zwischen den Versionen geachtet wurde.

Sofern noch kein Python installiert ist oder die Version aktualisiert werden soll, steht diese über die Python-Homepage  <http://www.python.org> jeweils in der aktuellen Version für Microsoft Windows, Linux und macOS zur Verfügung. Abbildung 13.3 zeigt den Installationsdialog am Beispiel von Microsoft Windows 10³. Im Normalfall müssen die Einstellungen des Installationsprogramms nicht geändert werden. Es ist nur zu empfehlen, die Option „Add Python 3.10 to PATH“ auszuwählen, damit Python einfacher zu starten ist. Sofern der Installationsort geändert werden soll, ist dies über den Auswahldialog „Customize installation“ möglich. Unter dem Betriebssystem Linux kann die Installation über den Befehl

```
sudo apt-get install python3.10
```

erfolgen. Mit dem Befehl „python -V“ kann in der Kommandozeile zudem überprüft werden,

³ Die im vorliegenden Kapitel darstellten Screenshots wurden unter dem Betriebssystem Microsoft Windows 10 erstellt.

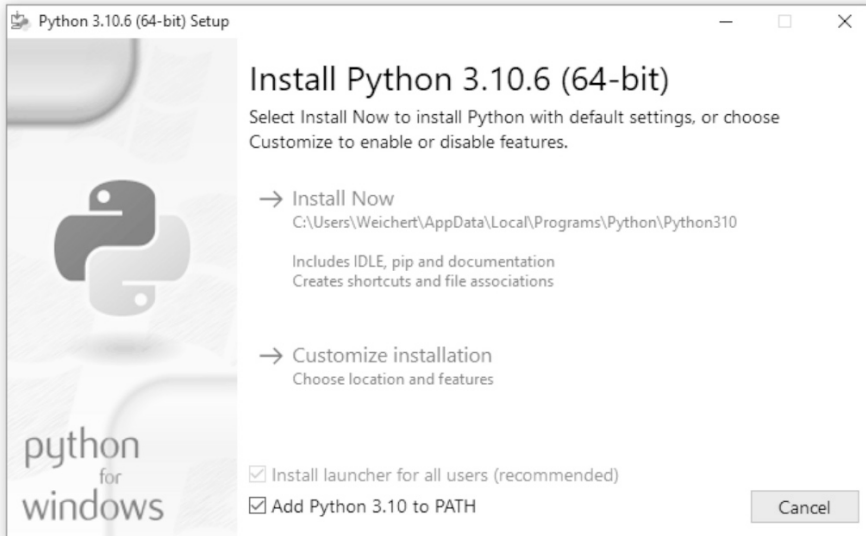


Abbildung 13.3: Screenshot zur Installation der 64 Bit-Version von Python 3.10.6 mittels der Datei `python-3.10.6-amd64.exe` unter Microsoft Windows 10.

ob Python schon installiert ist und in welcher Version. Dieser Befehl ist auch unter Windows über die Eingabeaufforderung (Befehl „CMD“) verfügbar. Sofern die Installation erfolgreich war, sollte – abhängig von der installierten Version – z. B. die Ausgabe „Python 3.10.6“ zu sehen sein. Unter macOS ist Python standardmäßig bereits installiert. Daher kann über den Befehl auch in einem Terminal überprüft werden, welche Version vorliegt. Ansonsten steht über die Python-Homepage <http://www.python.org> eine für das jeweilige Apple-System passende Installationsdatei zur Verfügung.

13.2.1 Aufbau und Übersetzung von Python-Programmen

Bevor wir im Folgenden ein erstes einfaches Python-Programm übersetzen und ausführen, sei noch einmal auf die Ausführungen im Abschnitt 8 hingewiesen, dass Python das Prinzip der imperativen und objektorientierten Programmierung umsetzt. Das bedeutet, dass die Funktionalität eines Programms aus der definierten Reihenfolge von Anweisungen resultiert. Python ordnet sich somit in das gleiche Programmierparadigma ein wie Java und C++. In beiden Fällen war die Ausführung von Programmen festgelegt. Der Quellcode wurde zunächst in einem Editor oder eine integrierte Entwicklungsumgebung (IDE, Integrated Development Environment) geschrieben, dann übersetzt und schließlich ausgeführt (C++, s. Abschnitt 12.5) bzw. interpretiert (Java, s. Abschnitt 4.3). An dieser Stelle gibt es einen Unterschied gegenüber den Gegebenheiten bei Java und C++. In Python steht als zweite Option ein interaktiver Modus zur Verfügung. Das

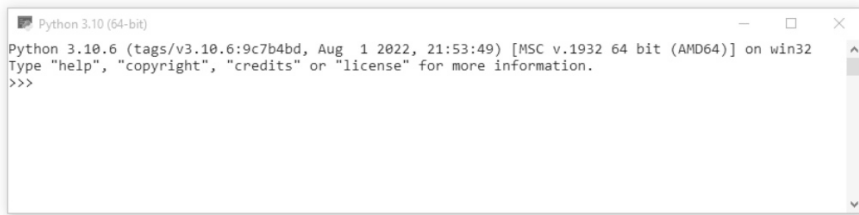


Abbildung 13.4: Darstellung der Python-Shell zur interaktiven Programmierung.

bedeutet, es besteht die Möglichkeit Python-Anweisungen direkt auszuführen. Korrekterweise müsste vergleichbar zu Java von interpretiert gesprochen werden, da Python ebenfalls im Kern eine Interpretersprache ist – die Details folgen in diesem Abschnitt.

Aufgabe 13.2:

Verändern Sie die Anweisung `print('Vorkurs!')` in der Form, dass die Zeichenkette „Vorkurs!“ zuerst einer Variablen zugewiesen und der Inhalt der Variablen dann über die `print`-Anweisung ausgegeben wird.

13.2.2 Interaktive Ausführung von Python-Programmen

Unter Windows kann der interaktive Modus, auch Python-Shell genannt, über den Menüpunkt **Startmenü** **Python 3.10** **Python 3.10 (64-bit)** im Windows-Startmenü oder über den Befehl `python` in der Windows-Kommandozeile aufgerufen werden. Es sollte dann eine vergleichbar zur Abbildung 13.4 dargestellte Ausgabe erscheinen. Neben der Versionsnummer werden noch weitere Zusatzinformationen angezeigt. In der nachfolgenden Zeile folgt die Eingabeaufforderung. Diese ist erkennbar an den drei Größerzeichen `>>>`, welche auch als Prompt bezeichnet werden. Der Vollständigkeit halber sei noch angeführt, dass die Eingabeaufforderung auch aus drei Punkten `...` besteht, sofern die Anweisungen, wie etwa eine bedingte Anweisung, über mehrere Zeilen gehen. Dieser Umstand wurde im Abschnitt 13.1 einleitend besprochen. Beendet werden kann die interaktive Shell mit dem Befehl `quit()`. Sowohl in Java und C++ wurde mit der Ausgabe „Vorkurs!“ auf dem Bildschirm begonnen. Eine Gegenüberstellung der beiden Quellcodes 12.1 und Quellcode 12.2 haben wir uns im Kapitel 12.1.2 (Seite 220) angesehen. Die Eingabe in Python hierzu (nach dem Prompt) ist

```
>>> print('Vorkurs!')
```

gefolgt durch ein Drücken der **Return**-Taste. Sollte es evtl. zu der Fehlermeldung „invalid character“ kommen, bitte zuerst darauf achten, dass die korrekten Anführungszeichen verwendet werden. Sofern alles korrekt ist, wird die bekannte Zeichenkette „Vorkurs!“ ohne die Anführungszeichen ausgegeben. Eine nützliche Funktionalität (auch zur Korrektur von Tippfehlern) in der Python-Shell ist die History-Funktion – die eingegebenen Befehlszeichen werden

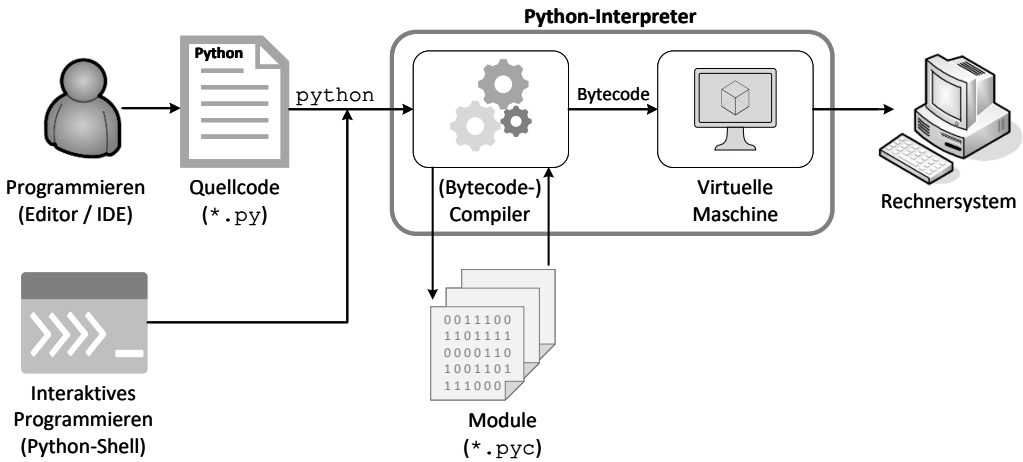


Abbildung 13.5: Schematische Darstellung zum „Übersetzung-Prozess“ bei der Programmiersprache Python.

sich gemerkt. Mit den Tastaturkommandos `Alt + p` kann rückwärts bzw. mit `Alt + n` vorwärts durch die Befehlshistory zugegriffen werden⁴.

Obwohl kein expliziter Befehl zur Übersetzung aufgerufen wurde, erfolgen „im Hintergrund“ verschiedene Bearbeitungsprozesse, welche in der Abbildung 13.5 dargestellt sind. Wir betrachten zunächst nur die Bearbeitungsfolge Python-Shell → (Bytecode-)Compiler → Virtuelle Maschine → Rechner-System. Die Anweisungen werden in einem ersten Schritt durch den Compiler auf (syntaktische) Korrektheit überprüft und, sofern keine Fehler aufgetreten sind, in einen Bytecode übersetzt. Dieses Vorgehen ist vergleichbar zur Ausführung der Anweisung `javac` bei der Programmiersprache Java. Es ist ein Kompromiss aus Plattformunabhängigkeit und einem schneller ausführbaren Code. Im nachfolgenden zweiten Schritt wird der Bytecode über die virtuelle Maschine (Python-Virtual-Machine) geladen und ausgeführt. In Java wurde dieser zweite Schritt durch die Anweisung `java` aufgerufen. In Python sind beide Schritte nicht getrennt – dies wird in der Abbildung 13.5 durch den Block „Python-Interpreter“ repräsentiert.

13.2.3 Entwicklungsumgebungen

Neben dem interaktiven Modus ist es in Python natürlich auch möglich, Programme zuerst mit einem Editor zu verfassen, diese in Quellcode-Dateien zu speichern und nachfolgend auszuführen. Python stellt nach der Installation bereits eine einfache integrierte Entwicklungsumgebung mit dem Namen IDLE (Integrated Development and Learning Environment) zur Verfügung. Für umfassendere Projekte stehen weitere Entwicklungsumgebungen zur Verfügung, beispielsweise

⁴ Unter MacOS ist die Befehlshistory über die Tastaturkommandos `Ctrl + p/n` zugreifbar.

Quellcode 13.3: Einfaches Python-Modul zur Ausgabe „Vorkurs!“

```

1 # Vorkurs Informatik - Modul
2 def vorkurs():
3     print('Vorkurs!')
```

PyCharm⁵ oder Anaconda⁶. Die integrierte Entwicklungsumgebung IDLE kann über den Menüpunkt `Python 3.10 | IDLE (Python 3.10 64-bit)` im Windows-Startmenü aufgerufen werden. Abbildung 13.6(a) zeigt, dass diese auch einen Prompt `>>>` zur interaktiven Programmierung anbietet – die Ausgabe ab den Gleichheitszeichen sollte zunächst ignoriert werden. Über den Pulldown-Menüpunkt `File >> New File` kann der Editor jetzt gestartet und der Quellcode

```

# Vorkurs Informatik
print('Vorkurs!')
```

eingegeben werden. Neben der bereits bekannten `print`-Anweisung ist in der ersten Zeile ein Kommentar eingefügt. Dieser beginnt mit dem `#`-Symbol. Der sicherlich noch recht einfache Programmcode kann über das Menü `File | Save As...` unter dem Namen `vorkurs.py` abgespeichert werden. Dateien mit Python-Quellcodes werden immer durch die Endung (das Suffix) `py` gekennzeichnet. Die Übersetzung und Ausführung folgt dem Prozess ab dem Punkt „Programmieren“ in der Abbildung 13.5. Recht einfach kann der Python-Interpreter über den Menüpunkt `Run >> Run Module` gestartet werden. Sofern keine Fehler aufgetreten sind, sollte sich eine vergleichbare Ausgabe zur Abbildung 13.6(b) ergeben. Zuerst wird der Name der Quellcode-Datei ausgegeben, nachfolgend die Ausgabe des Programms. Neben der Ausführung innerhalb der IDE kann dies auch über die Windows-Kommandozeile erfolgen, indem nach dem Befehl `python` als Argument der Name der Quellcodedatei angegeben wird. In unserem Beispiel würde sich somit die Anweisung

```
python vorkurs.py
```

ergeben.

13.2.4 Moduldateien

Im Zusammenhang mit der Erklärung zur Abbildung 13.5 wurde bisher der Block `Module` noch nicht besprochen. Vergleichbar zu Java und C++ ist es auch in Python sinnvoll, umfangreichere Programme nicht im interaktiven Modus zu verfassen, sondern diese in Python-Quellcodedateien zu schreiben. Hierzu sei auch vergleichend auf den Aufbau von Funktionen (vgl. Abschnitt 5) oder Klassen (vgl. Abschnitt 7) bei der Programmiersprache Java hingewiesen. In Python werden entsprechende Dateien als *Module* bezeichnet.

⁵ PyCharm (<https://www.jetbrains.com/pycharm>) ist als kostenlose Open-Source Community-Edition verfügbar.

⁶ Anaconda (<https://www.anaconda.com>) ist eine eigenständige Distribution für Python, welche die Entwicklungsumgebung Spyder und das Web-basierte Entwicklungssystem Jupyter Notebook beinhaltet.

Im Python-Quellcode 13.3 wird die bereits bekannte Ausgabe „Vorkurs!“ als Modul bzw. als Funktion definiert. Zu beachten ist dabei die Einrückung in Zeile 3, da in Python Blöcke durch Einrückungen definiert sind. Ohne auf die Details einzugehen merken wir uns zunächst, dass über das Schlüsselwort `def` die Funktion `vorkurs` definiert wird. Funktionen werden im Abschnitt 13.3 eingehender besprochen. Der Name des Moduls entspricht dem Dateinamen, im vorliegenden Fall somit „Vorkurs_Modul“. Um entsprechende Funktionen innerhalb von Modulen verwenden zu können, müssen diese zuerst mittels der Anweisung `import modulname` importiert werden. Dieser Mechanismus ist bereits über die Befehle `import` (Java) und `#include` (C++) bekannt. Entsprechend der Vorbemerkungen importiert der Befehl

```
>>> import vorkurs_modul
```

das Modul `Vorkurs_Modul`. Die Funktion `vorkurs` kann dann über die Anweisung

```
>>> vorkurs_modul.vorkurs()
```

ausgeführt werden. Resultierend sollte die Ausgabe „Vorkurs!“ zu sehen sein. Obwohl es für das vorliegende Programm sicherlich nicht notwendig ist, kann die Ausführung eines Programms mit der Tastenkombination `strg` + `c` abgebrochen werden.

Dieses Modul wurde insbesondere definiert, um einen Mechanismus zu verdeutlichen, der im Hintergrund erfolgt. Er besteht darin, dass Module beim ersten Import oder wenn sich ihr Inhalt ändert, kompiliert werden. Dieser Vorgang dient dazu, dass sie beim nächsten Aufruf schneller gestartet werden können. Die Ausführungsgeschwindigkeit wird dabei nicht verändert. Sie erkennen diesen Vorgang daran, dass im aktuellen Verzeichnis ein neues Verzeichnis mit der Bezeichnung „`__pycache__`“ angelegt wurde, in dem sich die kompilierten Dateien im Bytecode befinden. Diese sind an der Endung (dem Suffix) `.pyc` erkennbar. Neben dem Suffix folgen die Dateien dem Schema

Dateiname des Moduls (ohne Suffix) + `cpython-` + Python-Version + `.pyc`

zur Benennung. Die Moduldatei `vorkurs_modul.py` liegt in der kompilierten Fassung somit als Datei `vorkurs_modul.cpython-310.pyc` vor. Obwohl der Übersetzungsprozess automatisch nur für Module erfolgt, können auch andere Python-Daten mit einem speziellen Modul `compileall` manuell übersetzt werden. Dies könnte für die Python-Datei `vorkurs.py` mittels der Anweisung `python -m compileall vorkurs.py` erfolgen.

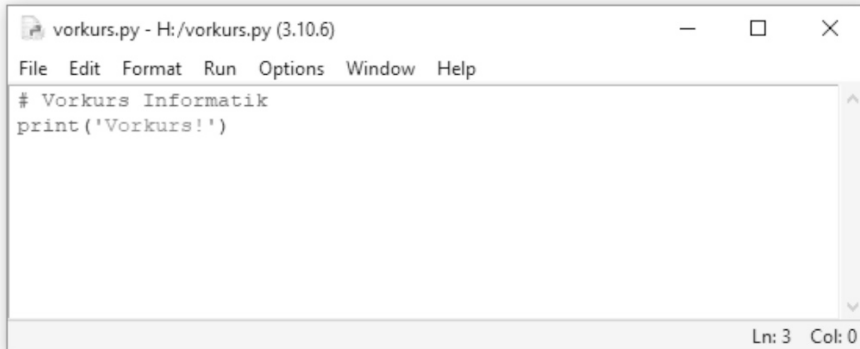
Aufgabe 13.3:

Erstellen Sie unter Verwendung des Editors einer integrierten Entwicklungsumgebung ein einfaches Python-Modul, welche die Anweisungen der Aufgabe 13.1 umfasst und führen Sie das Modul innerhalb der Entwicklungsumgebung aus.

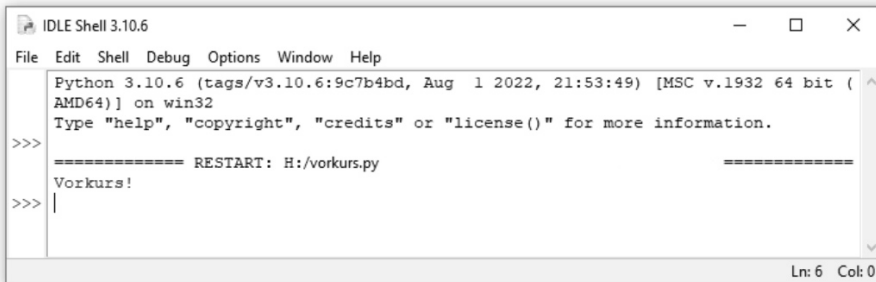


13.3 Funktionen

In den Java-Beispielen im Zusammenhang mit der Einführung in die Programmierung dieses Buches wurden Funktionen als statische Methoden realisiert. Im Unterschied zu Java stellt Python Funktionen unabhängig vom Klassenkonzept zur Verfügung. Funktionen in Python, welche



(a) Python-Shell von IDLE



(b) Programm vorkurs.py

Abbildung 13.6: Darstellung der integrierten Entwicklungsumgebung IDLE mit der (a) Python-Shell und dem (b) Programm-Editor.

außerhalb von Klassen deklariert werden, realisieren Funktionen im Sinne von statischen Java-Methoden (s. Kapitel 5). In Kontext des objektorientierten Programmierens von Python ergeben sich Methoden als Funktionen, welche innerhalb einer Klasse deklariert werden – auf die Details wird im Abschnitt 13.5 näher eingegangen.

Der Quellcode 13.4 zeigt am Beispiel der Berechnung einer Summe von Quadratzahlen die Realisierung einer Python-Funktion. Dabei leitet Schlüsselwort `def` eine Funktionsdefinition ein. Darauf folgt der Name der Funktion, die in Klammern gesetzte Liste der formalen Parameter und nach der Klammer ein Doppelpunkt. Die Anweisungen, die den Rumpf der Funktion bilden, beginnen in der nächsten Zeile und müssen linksausgerichtet eingerückt werden.

Zudem kann die erste Anweisung des Funktionskörpers optional eine Zeichenkette (String) sein (Zeile 2 in Quellcode 13.4), um das Programm bzw. die Funktion zu dokumentieren. Diese durch dreifache Anführungszeichen eingeschlossene Beschreibung wird als Dokumentations-

Quellcode 13.4: Python-Funktion zur Berechnung einer Summe von Quadratzahlen

```
1 def S2(n): # Summe der Quadrate der Zahlen von 1 bis n
2     """gibt Summe der Quadrate der Zahlen von 1 bis n zurueck"""
3     i=1
4     s=0
5     while i <= n:
6         s = s + i*i
7         i = i + 1
8     return s
```

Quellcode 13.5: Python-Prozedur zur Ermittlung einer Folge von Summen von Quadratzahlen

```
1 def S2Folge(n): # Folge von Summen von Quadratzahlen
2     i = 1
3     s = 0
4     while i <= n:
5         s = s + i*i
6         print(s, end=' ')
7         i = i + 1
```

String (Docstring) bezeichnet. Es gibt entsprechende Werkzeuge⁷, die Docstrings verwenden, um automatisch eine Online- oder gedruckte Dokumentation zu erstellen oder das interaktive Blättern durch den Programmcode zu ermöglichen.

Die `return`-Anweisung sorgt für die Rückkehr an die Aufrufstelle der Funktion im aufrufenden Programm und gibt dort den Wert zurück, welcher sich durch Auswertung des Ausdrucks in der `return`-Anweisung ergibt. Unter Verwendung der Funktion `S2` ergibt zum Beispiel die Anweisungsfolge

```
>>> maxn = 5
>>> m = 1
>>> while m <= maxn:
...     print(S2(m), end=' ')
...     m = m + 1
```

die Folge der Summen der Quadratzahlen von 1 bis 5 aus.

Sofern die `return`-Anweisung ohne ein Ausdrucksargument erfolgt, wird `None` zurückgegeben. Zudem erfolgt der Abbruch durch Erreichen des Endes einer Funktion. Derartige Funktionen mit dem Rückgabewert `None` werden auch als Prozeduren bezeichnet. Dabei kann die `return`-Anweisung auch weggelassen werden. Eine derartige Python-Prozedur zeigt der Quellcode 13.5. Unter Verwendung dieser Prozedur bewirkt die Ausführung von

⁷ Zur Überführung der Kommentare in Form von Python-Docstrings in eine strukturierte Dokumentation stehen verschiedene Werkzeuge zur Verfügung. Neben einer in der Entwicklungsumgebung *PyCharm* (<https://www.jetbrains.com/de-de/pycharm>) integrierten Verarbeitung seien mit *Swagger UI* (<https://swagger.io/tools/swagger-ui>) und *pdoc3* (<https://pdoc3.github.io/pdoc>) zwei exemplarische (semi)automatische Tools genannt.

Quellcode 13.6: Eine rekursive Python-Funktion zur Berechnung einer Summe von Quadratzahlen

```
1 def S2Rek(n): # Folge von Quadratzahlen
2     S2 = 1
3     if n > 1:
4         S2 = S2Rek(n-1) + n*n
5     return S2
```

Quellcode 13.7: Eine rekursive Python-Funktion zur Berechnung einer Folge von Summen von Quadratzahlen

```
1 def S2FolgeRek(n): # Summe der Quadrate der Zahlen von 1 bis n,
    rekursiv berechnet
2     S2 = 1
3     if n > 1:
4         S2 = S2FolgeRek(n-1) + n*n
5         print(S2, end=' ')
6     return S2
```

```
>>> maxn= 5
>>> S2Folge(maxn)
```

die Ausgabe der Folge der Summen der Quadratzahlen von 1 bis 5. Die Anzahl der durchgeführten Anweisungen ist hier deutlich geringer als im ersten Beispiel.

Vergleichbar zu Java erlaubt Python auch die rekursive Deklaration von Funktionen. Für eine ausführlichere Einführung in das Prinzip der Rekursion sei auf das Kapitel 6.2 verwiesen. Entsprechend berechnet der Quellcode 13.6 die Summe von Quadratzahlen auf rekursive Weise. Bei der Eingabe von $n=1$ wird der Wert 1 zurückgegeben. Für $n>1$ wird die Summe berechnet. Sofern n außerhalb des Definitionsbereichs liegt, d. h. $n<1$, wird ebenfalls 1 zurückgegeben. Eine weitere rekursive Funktion zeigt der Quellcode 13.7, welcher eine Folge von Summen von Quadratzahlen ausgibt. Dies geschieht durch Ausgabe des Ergebnisses am Ende jedes rekursiven Aufrufs durch eine `print`-Anweisung.

Unabhängig vom Prinzip der Rekursion wurden bisher klassische Funktionen, die einen Rückgabewert bestimmen, thematisiert und Prozeduren, die kein `return`-Argument haben oder `None` zurückgeben. In Python besteht zudem die Möglichkeit der Definition **anonymer Funktionen**. Anonyme Funktionen haben die Besonderheit, dass kein Funktionsname vorliegt. Dieser Typ von Funktionen wird auch als sogenannte Lambda-Funktion bezeichnet⁸. Lambda-Ausdrücke beginnen mit dem Schlüsselwort „`lambda`“. Daran schließt sich eine Folge von Parametern an, auf welche dann, durch einen Doppelpunkt getrennt, ein Ausdruck folgt. Sie sind also syntaktisch auf einen einzigen Anweisungsausdruck beschränkt. Das nachfolgende Beispiel

⁸ Die Formulierung der Lambda-Funktionen orientiert sich dabei an der Darstellung von Funktionen im Lambda-Kalkül der Mathematik [15].

```
>>> sum = lambda a, b: a+b
>>> print(sum(3, 4))
7
```

zeigt die Definition einer Lambda-Funktion mit zwei Parametern `a` und `b`, welche bei einem Aufruf die Summe der aktuellen Werte der Parameter `a` und `b` zurückgibt. Das Ergebnis wird der Variablen `sum` zugewiesen. Exemplarisch wird in dem Beispiel die Summe der Zahlen 3 und 4 berechnet sowie ausgegeben.

Derartige Lambda-Funktionen können insbesondere etwa dort verwendet werden, wo Funktionsobjekte benötigt werden. In dem vorliegenden Beispiel

```
>>> def make_increment(n):
...     return lambda x: x + n
>>> f = make_increment(10)
>>> print(f(1))
11
```

ist die Ausgabe des Aufrufs von `make_increment(10)` eine Referenz auf die `lambda`-Funktion mit dem Parameter `x`, welche der Variablen `f` zugewiesen wird. Bei einem Aufruf von `f(1)` wird dann „`lambda x: x + 10 (1)`“ ausgewertet, woraus sich $x + 10 = 1 + 10 = 11$ ergibt. Ein Aufruf von `f(2)` würde entsprechend 12 ergeben, da der Wert des Eingabeparameters von `f` stets um den konstanten Wert 10 erhöht wird. Das Beispiel illustriert, dass Funktionen Objekte sind und als solche über eine Referenz abgespeichert werden.

13.3.1 Variable Anzahl von Parametern

Bei der Definition von Funktionen wurden die Parameter bisher explizit bzgl. ihrer Anzahl und Position definiert. Diese werden daher auch als **Positionsparameter** bezeichnet. Zudem mussten beim Aufruf der Funktion für alle Parameter entsprechende Werte angegeben werden. In Python ist es aber auch möglich, Funktionen so zu deklarieren, dass sie mit einer variablen Anzahl von Parametern aufgerufen werden können. Die Tabelle 13.1 vermittelt eine vergleichende Übersicht an Beispielen zu den verschiedenen Formen der Parameterfestlegung, die zusätzlich kombiniert werden können:

- Angabe von Standardwerten
- Schlüsselwortparameter
- Parameterlisten.

Die erste Form ist die **Angabe von Standardwerten** (Default-Werten) für Parameter. Hierbei kann ein Standardwert für ein oder mehrere Parameter der formalen Parameterliste einer Funktion bei der Deklaration angegeben werden (s. Beispiel ② der Tabelle 13.1). Tritt ein solcher Parameter beim Aufruf der Funktion in der aktuellen Parameterliste nicht auf, wird der Standardwert verwendet. Dies bedeutet, dass der Parameter beim Aufruf der Funktion nicht angegeben werden muss.

Die zweite Form sind **Schlüsselwortparameter**, die auch als benannte Parameter bezeichnet werden. Bei Schlüsselwortparametern (Keyword Arguments) erfolgt der Aufruf durch Angabe

des formalen Parameters *Param* in der Form *Param = Wert*. Im zweiten Beispiel in Tabelle 13.1 trifft dies für die Parameter *jahr* und *name* bei den Aufrufen zu.

Grundsätzlich können die aktuellen Parameter an eine Python-Funktion entweder nach Position (Positionsparameter) oder explizit nach Schlüsselwort (Schlüsselwortparameter) übergeben werden. Positionsparameter müssen dabei vor Schlüsselwortparametern angegeben werden. Das nachfolgende Beispiel

```
>>> def printsum(a=3, b=4, c=5):
...     print(a+b+c)
>>> printsum()
12
>>> printsum(b=2, c=1, a=4)
7
>>> printsum(5, c=1, b=4)
10
```

vermittelt die Kombination von Positionsparametern, Schlüsselwortparametern und Defaultwerten an der Funktion `printsum` (s. a. Beispiel ② in der Tabelle 13.1).

Python bietet zudem eine Möglichkeit, die Art und Weise, wie Argumente übergeben werden können, einzuschränken. Dies kann durch zusätzliches, optionales Einfügen von Symbolparametern „/“ und „*“ in die formale Parameterliste bei der Funktionsdeklaration erfolgen, z. B.

```
>>> def f(pos1, pos2, /, pos_or_sw, *, sw1, sw2): ...
```

Bei den Parametern vor „/“ handelt es sich um Positionsparameter und bei jenen nach „*“ um Schlüsselwortparameter, dazwischen ist beides möglich. Durch diese Angabe kann die Lesbarkeit des Programmcodes verbessert werden.

Die dritte Form sind **Parameterlisten** (s. Beispiel ③ in der Tabelle 13.1). Hierfür gibt es zwei Varianten. Bei der ersten Variante, *Erstellen von Parameterlisten*, kann eine Funktion mit einer beliebigen Anzahl von aktuellen Parametern aufgerufen werden. In der formalen Parameterliste der Deklaration steht `*args` an der Stelle, an der beim Aufruf eine beliebige Anzahl aktueller Parameterwerte auftreten können. Davor und dahinter können keine oder mehrere normale Parameter stehen. Die dahinterstehenden Parameter müssen Schlüsselwortparameter sein. Beim Aufruf werden die durch `*args` repräsentierten aktuellen Parameter in ein Tupel (s. Abschnitt 13.4.1.2) verpackt und dieses dann an die Funktion übergeben. Das nachfolgende Beispiel

```
>>> def printargs(*args):
...     print(args)
>>> printargs(1, 5, 2, 3)
(1, 5, 2, 3)
```

zeigt die Übergabe einer beliebigen Anzahl an Parametern an die Funktion `printargs`, welche nachfolgend nur ausgegeben werden. Das Beispiel ④ in der Tabelle 13.1 zeigt zudem den Funktionsaufruf mit zwei oder drei Parametern. Bei der zweiten Variante, *Entpacken von Parameterlisten*, liegen die Parameterwerte als Liste (s. Abschnitt 13.4.1.1) oder Tupel vor. Bei einem Funktionsaufruf, der separate Positionsparameter erfordert, wird anstelle der Einzelparаметerwerte ein Verweis auf die Liste übergeben, wozu ebenfalls der `*`-Operator verwendet wird. Beim Aufruf werden die Parameterwerte aus der Liste oder dem Tupel ausgepackt und einzeln übergeben. Das Beispiel ⑤ in der Tabelle 13.1 zeigt diese Möglichkeit.

①	Deklaration	<code>def stud(nummer, jahr, name):</code> <code> return (nummer, name, jahr)</code>
	Aufrufe	<code>stud11 = stud(88188, 1980, 'Thorsten Meier')</code>
②	Deklaration	<code>def stud(nummer, jahr = 0, name = ""):</code> <code> return (nummer, name, jahr)</code>
	Aufrufe	<code>stud12 = stud(88633)</code> <code>stud13 = stud(88633, name = 'Mona Schmid')</code> <code>stud14 = stud(88633, 1981, 'Mona Schmid')</code>
③	Deklaration	<code>def stud(nummer, /, jahr, *, name = ""):</code> <code> return (nummer, name, jahr)</code>
	Aufrufe	<code>stud15 = stud(88633, 1981, name = 'Mona Schmid')</code> <code>stud16 = stud(88633, jahr = 1981, name = 'Mona Schmid')</code>
④	Deklaration	<code>def stud(nummer, *persdat):</code> <code> return persdat</code>
	Aufrufe	<code>stud17 = stud(88633, 'Mona Schmid')</code> <code>stud18 = stud(88633, 1981, 'Mona Schmid')</code>
⑤	Deklaration	<code>def stud(nummer, jahr, name):</code> <code> return (nummer, name, jahr)</code>
	Aufrufe	<code>persdat = (1981, 'Mona Schmid')</code> <code>stud19 = stud(nummer, *persdat)</code>

Tabelle 13.1: Exemplarische Übersicht zur Verwendung einer variablen Anzahl von Parametern innerhalb von Funktionen.

Das Konzept der Parameterlisten gibt es nicht nur für Parameterwerte, sondern auch für Parameterwerte mit Schlüsselwort. Hierzu wird der `**`-Operator genutzt. Anstelle von Listen oder Tupeln werden Wörterbücher verwendet, wobei die Schlüsselworte die Schlüssel bilden. Eine Anwendung des `**`-Operators vermittelt der Abschnitt 13.4.3. Für weitere Details sei zudem auf die Literatur verwiesen, z. B. [16, 17].

13.3.2

Gültigkeitsbereich von Variablen

Die Programmiersprache Python erlaubt, vergleichbar zu Java, Bezeichner innerhalb eines Programms mehrfach zu nutzen. Beispielsweise können verschiedene Variablen mit dem gleichen Bezeichner deklariert werden, wenn sie nicht in einem gleichen Gültigkeitsbereich liegen. Diese Gültigkeitsbereiche werden in Java durch die Blockstruktur eines Programms festgelegt (s. Abschnitt 5.3). In Python gibt es einen entsprechenden Mechanismus, welcher anhand von *Namensräumen* (name spaces, Kontexte) definiert wird. Unter einem Namensraum wird somit der Bereich eines Programms verstanden, in dem Namen definiert werden. Ein Gültigkeitsbereich eines Namens ist ein Bereich des Programms, in dem auf einen ihn enthaltenden Namensraum di-

Typ	Bedeutung	Lebensdauer
<i>Lokale Namensräume</i>	Namen, die innerhalb einer Methode oder Funktion definiert sind	während der Aktivierung einer Funktion oder Methode
<i>Modul-Namensräume</i>	Namen sind innerhalb einer Datei definiert	erzeugt beim Einlesen eines Moduls
<i>Eingebauter Namensraum</i>	von Python vordefinierte Namen	permanent vorhanden

Tabelle 13.2: Übersicht zur Bedeutung und Lebensdauer der unterschiedlichen Typen von Namensräumen

rekt zugegriffen werden kann, ohne einen Namens-Präfix zu verwenden. Dies trifft zum Beispiel innerhalb des Rumpfes einer Funktion zu. Python unterscheidet dabei zwischen lokalen, Modul-bezogenen und eingebauten Namensräumen. Dabei kann sich die Existenz von Namensräumen während der Programmausführung ändern. Innerhalb der Tabelle 13.2 werden die Bedeutung und Lebensdauer der drei Typen von Namensräumen vergleichend gegenübergestellt.

Der Gültigkeitsbereich eines Namens wird durch den ersten Namensraum seines Auftretens in der folgenden Reihenfolge festgelegt:

1. der lokale Namensraum des Namens,
2. die Namensräume aller umschließenden Funktionen, die beginnend mit dem nächstgelegenen umschließenden Namensraum durchsucht werden,
3. der Namensraum des aktuellen Moduls,
4. der eingebaute Namensraum.

Die Bezeichner eines Namenraums werden im Python-Programmiersystem in einer sogenannten Symboltabelle gespeichert. Diese Verzeichnisse verwenden den Datentyp Wörterbuch (Dictionary, s. Abschnitt 13.4.3 zur Erläuterung dieses Datentyps). Sie sind während der Programmausführung zugänglich.

Der Aufruf einer Funktion bewirkt die Einrichtung einer neuen Symboltabelle, welche für die lokalen Variablen der Funktion verwendet wird. Dabei wird der zugewiesene Wert unter dem Namen der Variablen für alle Variablenzuweisungen in einer Funktion in der lokalen Symboltabelle gespeichert. Für Variablenreferenzen wird, wie oben beschrieben, zuerst in der lokalen Symboltabelle, dann in den lokalen Symboltabellen der umschließenden Funktionen, dann in der globalen Symboltabelle und schließlich in der Tabelle der eingebauten Namen nachgesehen. Daher kann globalen Variablen und Variablen von umschließenden Funktionen nicht direkt ein Wert innerhalb einer Funktion zugewiesen werden, obwohl sie referenziert werden können. Ausnahmen sind globale Variablen, die in einer globalen Anweisung, und Variablen umschließender Funktionen, die in einer nicht-lokalen Anweisung benannt werden.

Bei einem Funktionsaufruf werden die aktuellen Parameterwerte in die lokale Symboltabelle der aufgerufenen Funktion eingefügt. Parameterwerte werden somit mit „Call-by-Value“ (Wertübergabe) übergeben. Allgemein bedeutet „Call-by-Value“, dass nur der Wert eines Parameters an die Funktion übergeben wird. Der Wert des Parameters in der aufrufenden Instanz wird somit

nicht verändert. Davon abgrenzend verhält es sich bei „Call-by-Reference“ (Referenzübergabe). In dem Fall wird eine Referenz auf den Parameter der Funktion übergeben. Jede Änderung wirkt sich direkt auf den Wert des Parameters aus. Somit erfolgt auch eine Änderung in der aufrufenden Instanz. Da der Wert in Python aufgrund der Klasseneigenschaft aller Datentypen immer eine Objektreferenz ist, bedeutet dies für Python, dass es sich hierbei letztendlich um „Call-by-Reference“ handelt. Wenn eine Funktion eine andere Funktion aufruft oder sich selbst rekursiv aufruft, wird eine neue lokale Symboltabelle für diesen Aufruf erstellt.

Eine Funktionsdeklaration assoziiert den Funktionsnamen mit dem Funktionsobjekt in der aktuellen Symboltabelle. Das Python-System erkennt das Objekt, auf welches dieser Name verweist, als benutzendendeklarierte Funktion.

Aufgabe 13.4:

Die nachfolgende Anweisungsfolge verwendet die Funktion `s2(n)` gemäß dem Quellcode 13.4 :

```
>>> print(s2(5))
>>> s = s2
>>> print(s(5))
```

Erläutern Sie, was bei Durchführung der Anweisungsfolge erfolgt.

Aufgabe 13.5:

Geben Sie die Ergebnisse der Funktionsaufrufe für die Objekte `stud1` bis `stud9` innerhalb der Tabelle 13.1 an.

Aufgabe 13.6:

Deklariieren Sie eine Python-Funktion `s3FolgeRek(n, b=True)`, die bei einem Aufruf `s3FolgeRek(m)` die Folge der Summe der Quadrate von 1 bis `m` ausdrückt und bei einem Aufruf `s3FolgeRek(m, b=False)` die Summe der Quadrate von 1 bis `m` zurückgibt.

Hinweis: Die Lösung kann durch einfache Modifikation des Quellcodes 13.7 umgesetzt werden.

13.4 Eingebaute Sammeldatentypen

Sammeldatentypen oder auch Container-Typen bezeichnen Datenstrukturen, welche mehrere Datenelemente speichern können. In Java ist das Array der wesentliche in die Sprache eingebaute Sammeltyp. Dies spielte in den entsprechenden Kapiteln eine wesentliche Rolle bei den Sortieralgorithmen, aber auch bei der Abspeicherung und dem Suchen nach Studierendenobjekten. Arrays ermöglichen den effizienten Zugriff auf ihre Elemente über die Indizes, wohingegen das Einfügen und Entfernen von Elementen aufwendig sind. Eine Alternative sind verzeigte lineare Listen, die in Java durch Verwendung von Verweisen bei der objektorientierten Programmierung möglich werden (s. Abschnitt 7.3.4). Verzeigte Listen erlauben das effiziente Einfügen und Entfernen von Elementen, wohingegen der gezielte Zugriff auf Elemente aufwendig ist. Im Unterschied zu Arrays sind verzeigte Listen und die damit verbundenen Operationen nicht in die Sprache Java eingebaut, sondern müssen selbst programmiert oder über eine Klassenbibliothek importiert werden.

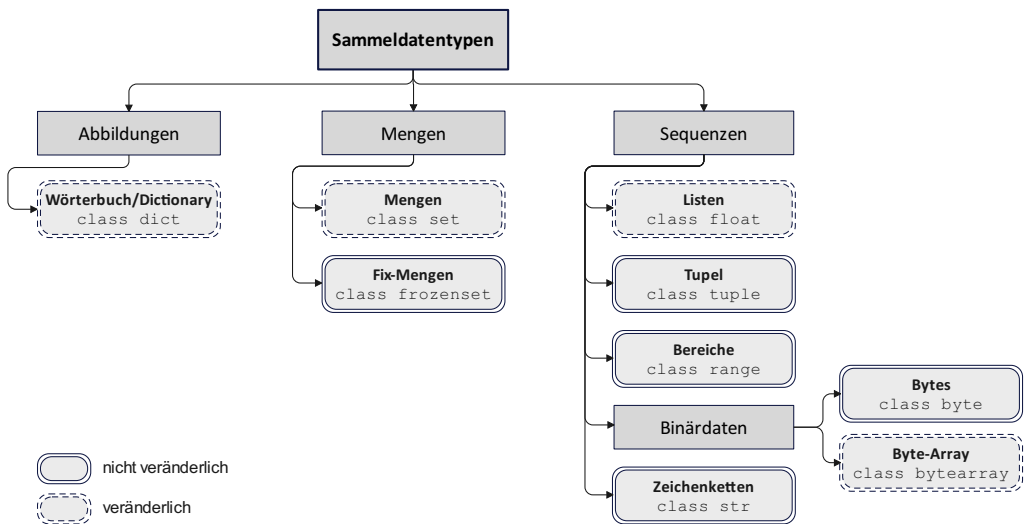


Abbildung 13.7: Schematische Übersicht zu den (eingebauten) Sammeltypen von Python.

Python stellt, anders als Java, ein recht umfangreiches Konzept von Klassen von Sammeltypen als Bestandteil der Sprachdefinition bereit (s. Abbildung 13.7). Das Konzept deckt den wesentlichen Bedarf in Anwendungsaufgaben ab. Es umfasst drei Arten von eingebauten Sammeltypen: **Sequenztypen** (engl. *sequence types*), **Mengentypen** (engl. *set types*) und **Abbildungstypen** (engl. *mapping types*). Sequenzen zeichnen sich durch die Anordnung ihrer Elemente aus, die, wie bei Java-Arrays, durch eine Indizierung mit ganzen Zahlen festgelegt ist. Dabei können Elemente auch mehrmals auftreten. Mengen sind hingegen, wie in der Mathematik, ungeordnet und ihre Elemente können nicht mehrmals in der Menge enthalten sein. Der einzige Abbildungstyp in Python ist das Wörterbuch (engl. *dictionary*), alternativ als Lexikon bezeichnet. Wörterbücher sind Sammlungen von Elementen, welche einem sogenannten Schlüssel einen Wert zuordnen. Bei klassischen Wörterbüchern ist der Schlüssel ein Wort, der Wert etwa die Übersetzung des Worts in eine andere Sprache. Aus Sicht der Mathematik können Wörterbücher als Funktionen oder „Familien“ aufgefasst werden.

Zudem wird bei Sequenztypen und Mengentypen zwischen **veränderlichen** (engl. *mutable*) und **unveränderlichen** (engl. *immutable*) Varianten unterschieden. Bei den unveränderlichen Varianten kann ein entsprechendes Objekt nach seiner Generierung nicht mehr verändert werden. Veränderliche Varianten stellen hingegen weitreichende Operationen zur Aktualisierung ihrer Objekte bereit. Dies umfasst die Änderung von gespeicherten Werten durch Zugriff über Indizes wie bei Arrays. Hinzu kommen aber auch Operationen wie das Einfügen neuer Elemente und das Entfernen von Elementen wie bei verzeigerten Listen. Veränderbare Python-Sequenztypen kombinieren damit die zu Beginn geschilderten Möglichkeiten dieser beiden Datenstrukturen.

Unveränderliche Sammeltypen sind bei einer weiteren Kategorisierung bedeutungsvoll, der **Hash-Fähigkeit**. Hashing ist eine Methode zur Speicherung von Datenelementen in einer indizierten Datenstruktur, beispielsweise in einem Array. Es ermöglicht das Einfügen von neuen Elementen in eine Datensammlung sowie das Auffinden und eventuell Entfernen von vorhandenen Elementen in der Datensammlung auf häufig effiziente Weise. In Python ist Hashing insbesondere für die Implementierung von Mengen und von Wörterbüchern durch das Python-System relevant.

Das Einfügen und Auffinden eines Elements durch Hashing geschieht mithilfe einer Hash-Funktion, die jedem möglichen Element einen Wert im Indexbereich der indizierten Datenstruktur (z. B. eines Array) zuordnet. Das Element wird dann unter seinem mit der Hash-Funktion ermittelten Index in der Datenstruktur abgelegt beziehungsweise gesucht. Für weitere Information sei auf den Abschnitt 17.5 verwiesen, der im Zusammenhang mit der Mengenverwaltung in die Methode des Hashing für ganze Zahlen als Datenelemente einführt.

Ein Datentyp in Python ist Hash-fähig, wenn eine Hash-Funktion für seine Elemente zur Verfügung steht. Die Hash-Funktion muss zudem die Eigenschaft haben, dass sich der von ihr zugewiesene Wert über die gesamte Lebensdauer eines Elements nicht ändert. Neben den elementaren Datentypen ist dies bei den Sammeltypen am ehesten bei den unveränderlichen Varianten gegeben. Die Voraussetzung der Hash-Fähigkeit der Elemente des Sammeldatentyps „Menge“ in Python ermöglicht die Python-interne Implementierung von Mengen durch Hashing, wofür Python über eine eingebaute Hash-Funktion verfügt.

Eine weit verbreitete Anwendung des Hashing ist der Fall von komplexeren Datenelementen, welche über ein sie identifizierendes Attribut verfügen. Ein Beispiel ist die Matrikelnummer in Studierendenobjekten aus Abschnitt 7.1. Ein solches identifizierendes Attribut kann als sogenannter Schlüssel dienen. Diese Gegebenheit trifft in Python auf Wörterbücher zu. Für solche Datenelemente kann Hashing dann durch eine Hash-Funktion auf dem Definitionsbereich der möglichen Schlüsselwerte realisiert werden, z. B. den ganzen Zahlen bei den Matrikelnummern. Ein Element wird über den Index seines Schlüssels gespeichert beziehungsweise gefunden. Die Voraussetzung der Hash-Fähigkeit der Schlüssel von Wörterbüchern in Python ermöglicht deren Python-interne Implementierung durch Hashing.

Der Hash-Wert erlaubt somit auch zu überprüfen, ob zwei Hash-fähige Objekte `obj1` und `obj2` gleich sind, somit der Ausdruck `obj1 == obj2` den Wert `True` liefert. Intern nutzt Python zur Überprüfung eine der sogenannten **magischen Funktionen**⁹, im vorliegenden Fall die Methode `__eq__` (`obj1`, `obj2`).

Ein weiterer bedeutungsvoller Begriff im Zusammenhang mit den Sammeltypen ist die **Iterierbarkeit** von Objekten. Iterierbar (engl. *iterable*) ist ein Objekt, welches mehrere Datenelemente umfasst, die es nacheinander zurückgeben kann. Python bietet hierzu die Möglichkeit, iterierbare Objekte auf elegante Weise in einer `for`-Schleife zu durchlaufen. Unter den eingebauten Sammeltypen ist die Iterierbarkeit bei Objekten eines Sequenztyps auf natürliche Weise und bei den Mengen- und Abbildungstypen durch Bedingungen an die Elemente bzw. Schlüssel gegeben. Darüber hinaus bietet Python einen Mechanismus zur Erstellung eigener iterierbarer Objekte an (s. Abschnitt 13.5 zur objektorientierten Programmierung). Die nachfolgenden Abschnitte gehen genauer auf die eingebauten Sammeltypen ein.

⁹ Die sogenannten magischen Funktionen sind durch zwei Unterstriche (`__`) vor und nach dem Namen der Methode gekennzeichnet. Es sind Methoden mit spezieller Bedeutung.

13.4.1 Sequenztypen

In Python gibt es zwei grundlegende allgemeine Sequenztypen, Listen (`class list`) und Tupel (`class tuple`). Ferner existieren spezialisierte Sequenztypen, insbesondere für Ganzzahlbereiche (`class range`) und für Zeichenketten (`class str`). Listen sind veränderlich, wohingegen Tupel, Ganzzahlbereiche und Zeichenketten unveränderlich sind. Weitere spezialisierte eingebaute Sequenztypen dienen der Verarbeitung von Binärdaten (`class bytearray`, `class byte`), wobei der Typ `bytearray` veränderlich und der Typ `byte` nicht veränderlich ist. Im Fokus der nachfolgenden Unterabschnitte stehen die grundlegenden Sequenztypen.

Abgesehen von Ausnahmen, etwa bei den Ganzzahlbereichen, gibt es allgemeine Sequenzoperatoren und Methoden, die für alle Sequenztypen anwendbar sind, unabhängig davon, ob sie veränderlich oder unveränderlich sind. Es lassen sich vier Arten von Operatoren unterscheiden: Test auf Enthaltensein, Konkatenation, lesender Zugriff über Indizes sowie die Ermittlung von Länge, Minimum und Maximum. Ferner gibt es noch Methoden, wie `index` und `count`. In der Tabelle 13.3 sind die allgemeinen Operationen für den Sammeldatentyp Sequenz zusammengestellt. Dabei sind die Sequenzoperationen nach aufsteigender Priorität angeordnet. Die Operatoren `in` und `not in` haben die gleichen Prioritäten wie Vergleichsoperatoren und die Operatoren `+` (Verkettung) und `*` (Wiederholung) wie entsprechende numerische Operatoren. Bei der Konkatenation unveränderlicher Sequenzen ist zu beachten, dass diese zu einem neuen Objekt führt.

Allen Sequenztypen ist gemeinsam, dass es sich um Folgen von Elementen handelt, die mit ganzen Zahlen indiziert sind. Analog zu Java-Arrays ist die Vorwärtsindizierung, aufsteigend mit Index 0 beginnend, gebräuchlich. Es gibt jedoch auch andere Möglichkeiten, insbesondere die Rückwärtsindizierung, beginnend mit -1 an dem letzten Element. Entsprechend ergibt sich im nachfolgenden Beispiel

```
>>> x = [3, 7, 8, 1]
>>> x[2]
8
>>> x[-2]
8
```

für beide Zugriffe auf die Liste `x` (Details s. nachfolgenden Abschnitt 13.4.1.1) das identische Ergebnis, da die Zahl 8 bei einer Indizierung von vorne (Indizes: 0, 1, 2, 3) an der Position 2 und bei einer Infizierung von hinten (Indizes: -1, -2, -3, -4) an der Position -3 angeführt ist.

Ergänzend zu dieser Indizierung einzelner Elemente stehen auch Operationen für den gleichzeitigen Zugriff auf mehrere Elemente zur Verfügung. Entsprechend definiert die Anweisung `s[i:j]` das Teilstück von `s` von `i` bis `j` als die Folge derjenigen Elemente von `s`, deren Index `l` die Bedingung $i \leq l < j$ erfüllt. Wenn `i` größer oder gleich `j` ist, ist das Teilstück leer. Zudem erlaubt die Anweisung `s[i:j:k]` ein Teilstück von `i` bis `j` mit Schrittweite `k` von `s` als die Folge von Elementen mit Index $l = i + n \cdot k$ zu definieren, sodass $0 \leq l < (j-i)/k$ gilt. Für die einleitend definierte Liste `x` sind somit die Anweisungen

```
>>> x[1:3]
[7, 8]
>>> x[0:len(x)+1:2]
[3, 8]
```

Operation	Beispiel	Bedeutung
<code>x in s</code>	<code>libsp1 = ('zwei' in Liste1)</code>	True, wenn ein Eintrag in <code>s</code> gleich <code>x</code> ist, sonst False
<code>x not in s</code>	<code>libsp2 = ('zwei' not in Liste1)</code>	False, wenn ein Eintrag in <code>s</code> gleich <code>x</code> ist, sonst True
<code>s + t</code>	<code>libsp3 = Liste2 + Liste3</code> <code>libsp4 = (Liste1 != Liste3)</code>	Konkatenation von <code>s</code> und <code>t</code>
<code>s * n</code> oder <code>n * s</code>	<code>libsp5 = 3 * Liste2</code>	<code>n</code> -malige Konkatenation von <code>s</code> mit sich selbst
<code>s[i]</code>	<code>libsp6 = Liste4[1]</code>	<code>i</code> -ter Eintrag von <code>s</code> , gezählt ab 0
<code>s[i:j]</code>	<code>libsp7 = Liste1[0,8]</code>	Teilstück von <code>s</code> von <code>i</code> bis <code>j</code>
<code>s[i:j:k]</code>	<code>libsp8 = Liste1[0,7,2]</code>	Teilstück von <code>s</code> von <code>i</code> nach <code>j</code> mit Schrittwerte <code>k</code>
<code>len(s)</code>	<code>libsp9 = len(Liste1)</code>	Länge von <code>s</code>
<code>min(s)</code>	<code>libsp10 = min(Liste2)</code>	kleinstes Element von <code>s</code>
<code>max(s)</code>	<code>libsp11 = max(Liste2)</code>	größtes Element von <code>s</code>
<code>s.index(x[,i[,j]])</code>	<code>libsp12 = Liste2.index(1)</code> <code>libsp13 = Liste2.index(2,2)</code>	Index des ersten Vorkommens von <code>x</code> in <code>s</code> bei oder nach Index <code>i</code> und vor Index <code>j</code>
<code>s.count(x)</code>	<code>libsp14 = Liste1.count('eins')</code>	Anzahl des Auftretens von <code>x</code> in <code>s</code>
<code>sort(*, key=None, reverse=False)</code>	<code>libsp15 = Liste1.sort()</code>	Sortieren einer Liste in aufsteigender Reihenfolge, optional anhand des Vergleichsschlüssels <code>key</code> und in absteigender Reihenfolge (<code>reverse=True</code>)

Tabelle 13.3: Übersicht zu den allgemeinen Operationen für Sequenztypen. Dabei sind `s` und `t` Sequenzen desselben Typs, `n`, `i`, `j` und `k` sind ganze Zahlen und `x` ist ein beliebiges Objekt, das alle von `s` auferlegten Typ- und Werteinschränkungen erfüllt. Im Hinblick auf die Definitionen von `Liste1` bis `Liste4` sei auf die Abbildung 13.8 verwiesen.

```

>>> Liste1 = ['eins', 1, 'zwei', 2, 'drei', 3, 'vier', 4, 'eins', 1]
>>> Liste2 = [1, 2, 3, 4, 1]
>>> Liste3 = ['eins', 'zwei', 'drei', 'vier', 'eins']
>>> Liste4 = [['eins', 1], ['zwei', 2], ['drei', 3]]
>>> Liste5 = [['zwei', 2], ['drei', 3], ['vier', 4]].

```

Abbildung 13.8: Exemplarische Instanziierung der Listenobjekte *Liste1* bis *Liste5*

möglich. Da für den Index l die Bedingung $i \leq l < j$ gilt, erlaubt der Ausdruck $\text{len}(x)+1$ für den letzten zu betrachtenden Index j auch das letzte Element der Sequenz beachten zu können.

Sequenzen desselben Typs unterstützen auch Vergleiche. Insbesondere Tupel und Listen werden lexikografisch verglichen, indem die entsprechenden Elemente verglichen werden. Das bedeutet, dass bei Gleichheit beide Sequenzen die gleiche Länge haben und die verglichenen Elemente jeweils gleich sein müssen.

Für veränderbare Sequenztypen gibt es weitere Operatoren, welche die Änderung von gespeicherten Werten und der Sequenzstruktur über die Indizierung ermöglichen. Ferner gibt es Methoden für weitere Veränderungen. Die Tabelle 13.4 stellt die Operationen für veränderbare Sequenztypen zusammen. Entsprechende Beispiele werden im Zusammenhang mit den im Folgenden dargestellten Sequenztypen angegeben.

Die einzige Operation, die unveränderliche Sequenztypen im Allgemeinen implementieren und die nicht auch von veränderlichen Sequenztypen implementiert wird, ist die Unterstützung für die eingebaute Hash-Funktion. Diese Unterstützung erlaubt es, unveränderlichen Sequenzen, wie z. B. Tupelinstanzen, als dict-Schlüssel zu verwenden und in Mengenobjekten zu speichern. Eine Voraussetzung für die Hash-Fähigkeit einer unveränderlichen Sequenz ist, dass sie Hash-fähige Werte enthält.

13.4.1.1 Listen

Der in Python eingebaute Listentyp gehört zu den veränderlichen Sequenztypen und ist durch `class list` vordeclariert. Listenobjekte können, analog zu Arrays in Java, explizit durch Angabe ihrer Elemente erstellt werden. Dabei werden die einzelnen Elemente, durch Kommas getrennt, in eckigen Klammern aufgeführt. Die Abbildung 13.8 vermittelt einige exemplarische Instanziierungen von Listenobjekten. Dabei repräsentiert beispielsweise `[1, 2, 3, 4, 1]` eine Liste mit fünf ganzzahligen Elementen. Befinden sich zwischen den eckigen Klammern keine Elemente, handelt es sich um eine leere Liste. Anders als bei einem Array in Java können die Listenelemente in Python einen unterschiedlichen Typ haben. So enthält die Liste `['eins', 1, 'zwei', 2, 'drei', 3, 'vier', 4, 'eins', 1]` ganzzahlige Elemente und Zeichenkettenelemente. Solche Listenobjekte werden „inhomogen“ genannt. Eher gebräuchlich sind homogene Listen, bei denen alle Listenelemente vom gleichen Typ sind.

Eine weitere Möglichkeit der Instanziierung von Listenobjekten ist der Aufruf der Klasse durch `list()` oder `list(iterable)`. Dabei erzeugt der Aufruf `list()` eine leere Liste. Im Aufruf `list(iterable)` steht `iterable` für ein iterierbares Objekt, z. B. vom Sequenztyp. Wenn

`iterable` bereits vom Typ `list` ist, wird eine Kopie hiervon erstellt und ein Verweis darauf zurückgegeben. Ist `iterable` keine Liste, wird daraus ein Objekt vom Typ `list` erstellt, deren Elemente die gleichen wie in `iterable` sind und dessen Reihenfolge mit jener in `iterable` übereinstimmt. Beispielsweise gibt `list({1, 2, 3})` für die Menge $\{1, 2, 3\}$ eine Liste `[1, 2, 3]` zurück.

Listenobjekte können sich auch als Ergebnis von Operationen ergeben. Beispielsweise ist ein Teilstück, welches aus einer Liste extrahiert wurde, wieder ein Listenobjekt. Daher ist gemäß der nachfolgenden Anweisungen

```
>>> x = [1, 'zwei', 3, 'vier']
>>> y = x[1:3]
>>> y
['zwei', 3]
```

`y` Teilstück als Listenobjekt des gegebenen Listenobjekts `x`.

Eine komfortable Möglichkeit zum Erstellen von Listen ist die Listenabstraktion (engl. *list comprehension*). Listenabstraktionen, und analog Mengenabstraktionen und Wörterbuchabstraktionen, ähneln der in der Mathematik gebräuchlichen Notation zur Definition von Teilmengen, z. B. $\{n^2 : 1 \leq n \leq 5, n \neq 3\}$. Eine Abstraktion dieses Beispiels für Listen ist

```
>>> squares = [n*n for n in [1,2,3,4,5] if n != 3]
>>> squares
[1, 4, 16, 25].
```

Hierbei nutzt die `for`-Schleife die Eigenschaft der Iterierbarkeit von Listen (s. Abschnitt 13.4.4).

Listen implementieren alle allgemeinen und veränderbaren Sequenzoperationen. Darüber hinaus bieten sie eine Methode `sort(*, key=None, reverse=False)` an. Über `key` kann eine Funktion mit einem Parameter zur Extraktion eines Vergleichsschlüssels aus jedem Listenelement übergeben werden, z. B. als `lambda`-Ausdruck (s. Abschnitt 13.3). Der Standardwert `None` bedeutet, dass die Listenelemente direkt sortiert werden, ohne dass ein separater Schlüsselwert berechnet wird. Zudem ist `reverse` ein boolescher Wert, der bewirkt, dass für `True` nach fallender Größe des Schlüssels sortiert wird. Die Methode `sort()` ist stabil, d. h. die relative Reihenfolge von Elementen, welche gleich sind, bleibt unverändert.

13.4.1.2 Tupel

Der Datentyp `Tupel` bestimmt unveränderliche Sequenzen, welche durch die Klasse `class tuple` in Python vordefiniert sind. Sie implementieren alle allgemeinen Sequenzoperationen. `Tupel` können explizit durch Angabe einer Folge von Elementen angegeben werden, die durch Kommas getrennt sind, z. B. `1, 2, 3` oder `(1, 2, 3)`. Hierbei ist zu beachten, dass es das Komma ist, das ein `Tupel` bildet, nicht die Klammern. Die Klammern sind optional, außer im Fall eines leeren `Tupels` „`()`“, oder wenn sie benötigt werden, um syntaktische Mehrdeutigkeit zu vermeiden. Zum Beispiel ist `f(a, b, c)` ein Funktionsaufruf mit drei Parametern, während `g((a, b, c))` ein Funktionsaufruf mit einem 3-`Tupel` als einzigem Parameter ist. Bei einelementigen `Tupeln` wird ein nachgestelltes Komma verwendet, also z. B. `a,` bzw. `(a,)`.

Operation	Beispiel	Bedeutung
<code>s[i] = x</code>	<code>Liste2[4] = 5</code>	Element <code>i</code> von <code>s</code> wird durch <code>x</code> ersetzt
<code>s[i:j] = t</code>	<code>Liste1[8:10] = ['fünf', 5]</code>	ersetzt das Teilstück von <code>s</code> von <code>i</code> bis <code>j</code> durch den Inhalt von <code>t</code>
<code>del s[i:j]</code>	<code>del Liste1[8:10]</code>	entspricht <code>s[i:j] = []</code>
<code>s[i:j:k] = t</code>	<code>Liste1[1:10:2] = ['one', 'two', 'three', 'four', 'one']</code>	ersetzt die Elemente von <code>s[i:j:k]</code> durch jene von <code>t</code>
<code>del s[i:j:k]</code>	<code>del Liste1[0:9:2]</code>	entfernt die Elemente <code>s[i:j:k]</code> aus der Sequenz
<code>s.append(x)</code>	<code>Liste4.append(['vier', 4])</code>	fügt <code>x</code> an das Ende der Sequenz an (entspricht <code>s[len(s):len(s)] = [x]</code>)
<code>s.clear()</code>	<code>Liste4.clear()</code>	entfernt alle Elemente von <code>s</code>
<code>s.copy()</code>	<code>libsp16 = Liste4.copy()</code>	erzeugt eine flache Kopie von <code>s</code>
<code>s.extend(t)</code> oder <code>s += t</code>	<code>Liste2.extend(Liste3)</code>	erweitert <code>s</code> um den Inhalt von <code>t</code>
<code>s *= n</code>	<code>libsp17 *= Liste2</code>	aktualisiert <code>s</code> mit seinem Inhalt <code>n</code> -mal wiederholt
<code>s.insert(i, x)</code>	<code>Liste3.insert(0, ['eins', 1])</code>	fügt <code>x</code> an dem durch <code>i</code> gegebenen Index in <code>s</code> ein
<code>s.pop()</code> oder <code>s.pop(i)</code>	<code>libsp18 = Liste1.pop()</code>	holt das Element an <code>i</code> und entfernt es auch aus <code>s</code>
<code>s.remove(x)</code>	<code>Liste2.remove(1)</code>	entfernt das erste Element aus <code>s</code> , bei dem <code>s[i]</code> gleich <code>x</code> ist
<code>s.reverse()</code>	<code>Liste4.reverse()</code>	kehrt die Elemente von <code>s</code> an Ort und Stelle um

Tabelle 13.4: Übersicht zu den ergänzenden Operationen für veränderliche Sequenztypen. Die Variable `s` ist dabei eine Instanz eines veränderlichen Sequenztyps, `t` ein beliebiges iterierbares Objekt und `x` ein beliebiges Objekt, welches alle von `s` auferlegten Typ- und Werteinschränkungen erfüllt (z. B. akzeptiert `bytearray` nur Ganzzahlen, welche die Einschränkung $0 \leq x \leq 255$ erfüllen). Im Hinblick auf die Definitionen der Listenobjekte `Liste1` bis `Liste4` sei auf die Abbildung 13.8 verwiesen.

Ferner können Tupelobjekte durch Aufruf der entsprechenden Klasse instanziiert werden. Der Aufruf `tuple(iterable)` erzeugt ein Tupelobjekt, dessen Elemente die gleichen wie in `iterable` sind und deren Reihenfolge mit jener in dem iterierbaren Objekt `iterable` übereinstimmt. Wenn `iterable` bereits ein Tupel ist, wird es unverändert zurückgegeben. Zum Beispiel liefert `tuple({1, 2, 3})` für die Menge `{1, 2, 3}` ein Tupel `(1, 2, 3)`. Wenn kein Argument angegeben wird, erzeugt der Aufruf ein neues leeres Tupel `()`.

Tupel können wie Listen homogen oder heterogen sein. Sie werden typischerweise zum Speichern von Sammlungen heterogener Daten verwendet. So können etwa Studierendendaten, bestehend aus Name, Matrikelnummer und Geburtsjahr, auch durch Tupel gespeichert werden, z. B. `'Thorsten Meier', 88188, 1980` oder `('Thorsten Meier', 88188, 1980)`.

13.4.1.3 Bereiche (Range)

Der Bereichstyp stellt eine unveränderliche Folge von Zahlen dar und ist durch die Klasse `range` in Python vordefiniert. Er implementiert alle allgemeinen Sequenzoperationen mit Ausnahme von Verkettung und Wiederholung. Diese würden zu Sequenzen führen, die nicht dem Muster von Bereichen entsprechen. Eine häufige Anwendung des Bereichstyps ist, eine `for`-Schleife für eine bestimmte Anzahl zu durchlaufen.

Erzeugt werden kann ein Objekt des Bereichstyps mit einem Klassenaufruf. Bei einem Aufruf von `range(start, stop)` mit ganzzahligen Parameterwerten bewirkt dies die Instanziierung eines Bereichsobjekts aus den ganzen Zahlen, die größer oder gleich `start` und kleiner als `stop` sind. Erfolgt der Aufruf mit nur einem Parameter, `range(stop)`, beginnt die Folge der Zahlen bei 0. Zudem kann der Aufruf mit drei Parametern erfolgen, `range(start, stop, step)`. Dann gibt der dritte Parameter, der ungleich 0 sein muss, eine Schrittweite an, d. h. zwei aufeinanderfolgende Zahlen unterscheiden sich durch den Wert von `step`. Die Parameter können auch negativ sein, wodurch etwa rückwärts angeordnete Folgen generiert werden können. Es ist jedoch auf Sinnhaftigkeit der Werte zu achten, da ansonsten eine Fehlermeldung ausgelöst werden kann. In den nachfolgenden Anweisungen

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(0, 28, 4))
[0, 4, 8, 12, 16, 20, 24]
>>> list(range(0, -5, -1))
[0, -1, -2, -3, -4]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

wird die Verwendung der Anweisungen exemplarisch verdeutlicht.

Zudem erlaubt der Bereichstyp `range`, Listen kompakter zu definieren. Damit kann z. B. die im Abschnitt 13.4.1.1 angeführte Anweisung `squares = [n*n for n in [1,2,3,4,5] if n != 3]` auch verkürzt in der Form `squares = [n*n for n in range(1,6) if n != 3]` geschrieben werden.

Die Prüfung von Bereichsobjekten auf Gleichheit mit `==` und `!=` vergleicht sie als Sequenzen. Das heißt, zwei Bereichsobjekte werden als gleich angesehen, wenn sie dieselbe Folge von Werten darstellen. Dabei können zwei Bereichsobjekte, welche gleich sind, auf unterschiedliche Weise instanziiert worden sein. In dem nachfolgenden Beispiel

```
>>> range(0) == range(2, 1, 3)
True
>>> range(0, 3, 2) == range(0, 4, 2)
True
```

wird für die erste Überprüfung auf Gleichheit durch beide Anweisungen eine leere Liste erzeugt und bei der zweiten Überprüfung jeweils die Liste `[0, 2]`.

Der Vorteil von Bereichsobjekten gegenüber einer normalen Liste oder einem Tupel besteht darin, dass sie unabhängig von der Größe des Bereichs, den sie repräsentieren, immer nur den gleichen (geringen) Speicherplatz für die Parameter benötigen, da die Bereiche nach Bedarf berechnet werden können.

13.4.1.4 Zeichenketten (Strings)

Textuelle Daten können in Python, ähnlich wie in Java, durch Zeichenkettenobjekte (String-Objekte) gespeichert und verarbeitet werden. Zeichenkettenobjekte sind Instanziierungen der vordefinierten Python-Klasse `str` und repräsentieren unveränderliche Sequenzen von Unicode-Zeichen¹⁰. Zeichenkettenliterals können dabei auf verschiedene Arten geschrieben werden:

- *einfache Anführungszeichen*:
'erlaubt eingebettete "doppelte" Anführungszeichen'
- *doppelte Anführungszeichen*:
"erlaubt eingebettete 'einfache' Anführungszeichen"
- *dreifache Anführungszeichen*:
'''Drei einfache Anführungszeichen''' oder
"""Drei doppelte Anführungszeichen"""

Zeichenfolgenliterals in dreifachen Anführungszeichen können sich über mehrere Zeilen erstrecken – alle zugehörigen Leerzeichen werden in das Zeichenfolgenliteral aufgenommen. Neben Textzeichen können in den Zeichenketten zudem Steuerzeichen (sogenannte Escape-Zeichen) enthalten sein, z. B. „\n“ für einen Zeilenwechsel.

Mit dem Aufruf der Klasse `str` können auch aus anderen Objekten Zeichenkettenobjekte erstellt werden. Dies wird hier nicht weiter verfolgt.

Die Klasse `str` implementiert alle allgemeinen Sequenzoperationen. Hierzu ist zu erwähnen, dass es in Python keinen separaten Zeichentyp, wie etwa `char` in Java, gibt, sodass ein Zugriff der Form `s[i]` für eine Zeichenkette eine Zeichenkette der Länge 1 liefert. Darüber hinaus

¹⁰ Unicode bezeichnet eine internationale Standardisierung zur Repräsentation (Kodierung) von Zeichen auf elektronischen Medien (z. B. Rechnersystemen). Die Festlegung erfolgt durch das gemeinnützige Unicode-Konsortium (www.unicode.org).

```
>>> Menge1 = {'eins', 1, 'zwei', 2, 'drei', 3, 'vier', 4}
>>> Menge2 = {1,2,3,4}
>>> Menge3 = {'eins', 'zwei', 'drei', 'vier'}
>>> Menge4 = {('eins',1), ('zwei',2), ('drei', 3)}
>>> Menge5 = {('zwei',2), ('drei', 3), ('vier',4)}.
```

Abbildung 13.9: Exemplarische Instanziierung der Mengenobjekte *Menge1* bis *Menge5*

gibt es weitere, zeichenkettenspezifische Methoden zur Zeichenkettenformatierung. Insbesondere werden zwei Arten von Zeichenkettenformatierungen unterstützt. Die erste Art ist durch die Methode `str.format()` verfügbar und bietet ein hohes Maß an Flexibilität und Anpassungsfähigkeit. Zudem existiert eine zweite Form, welche sich am Formatierungsstil von `printf` in C/C++ orientiert [8]. Auf dieses Thema wird hier nicht weiter eingegangen.

13.4.2 Mengentypen

Nachdem in den vorhergehenden Abschnitten Sequenzen besprochen wurden, widmen sich die folgenden Erklärungen verschiedenen Typen von Mengen. Mengentypen dienen zur Speicherung und Verarbeitung von Datensammlungen, welche die Eigenschaften einer Menge in der Mathematik haben. Das bedeutet, dass im Unterschied zu Listen jedes Element nur einmal auftritt und die Elemente nicht angeordnet sind. Daraus ergibt sich als wesentlicher Unterschied, dass Mengenobjekte in Python im Gegensatz zu Listenobjekten keine Indizes haben, über die auf die Elemente zugegriffen werden kann.

Python verfügt über zwei eingebaute Mengentypen, die „Mengen“ (engl. *sets*), vordeklariert durch die Klasse `class set`, und die „eingefrorenen Mengen“, vordeklariert durch die Klasse `class frozenset`. Objekte beider Typen repräsentieren eine ungeordnete Sammlung eindeutiger Hash-fähiger Elemente. Ein Grund für die Bedingung der Hash-Fähigkeit ist, dass Mengen dadurch systemintern effizient als Hash-Datenstruktur implementiert werden können. Objekte des Typs `set` sind veränderlich, des Typs `frozenset` sind nicht veränderlich. Dies ähnelt den Gegebenheiten bei Listen bzw. Tupeln. Da die Elemente von Mengen vom Typ `frozenset` zudem Hash-fähig sind, sind Objekte vom Typ `frozenset` Hash-fähig. Sie können damit auch Elemente von Mengen sein und als Wörterbuchschlüssel verwendet werden.

Objekte vom Typ `set` und `frozenset` können durch Aufruf der entsprechenden Klasse instanziiert werden. Die Aufrufe `set()` bzw. `frozenset()` erzeugen eine leere Menge. Die Aufrufe `set(iterable)` bzw. `frozenset(iterable)` geben ein neues `set`- bzw. `frozenset`-Objekt zurück, dessen Elemente aus `iterable` entnommen werden.

Nicht leere Mengen vom Typ `set` können auch durch eine durch Kommas getrennte Folge von Elementen in geschweiften Klammern explizit angegeben werden, z. B. `{ 'Thorsten Meier', 'Monika Schmidt', 'Monika Schneider' }`. Ferner gibt es analog zur Listenabstraktion auch eine Mengenabstraktion (engl. *set comprehension*) zur Erzeugung von Mengenobjekten. Die Abbildung 13.9 zeigt hierzu einige exemplarische Instanziierungen von Mengenobjekten.

Zu den allgemeinen Mengenoperationen, die für die Mengentypen `set` und `frozenset` implementiert sind, gehören insbesondere Zugehörigkeitstests sowie mathematische Operationen wie Schnittmenge, Vereinigung, Differenz und symmetrische Differenz. Es sei noch einmal darauf hingewiesen, dass im Unterschied zu Sequenztypen keine Indizierung vorliegt. Die Tabelle 13.5 repräsentiert eine Übersicht über allgemeine Mengenoperationen.

Dabei akzeptieren die in der Tabelle 13.5 angeführten Methoden `union()`, `intersection()`, `difference()`, `symmetric_difference()`, `issubset()` und `issuperset()` beliebige iterierbare Objekte als Parameter. Im Gegensatz dazu müssen die Parameter ihrer ebenfalls angegebenen, auf Operatoren basierenden Gegenstücke, Mengen sein. Dies schließt fehleranfällige Konstruktionen wie beispielsweise `set('xyz') & 'yza'` zugunsten der besser lesbaren `set('xyz').intersection('yza')` aus. Binäre Operationen, die `set`-Instanzen mit `frozenset` mischen, geben den Typ des ersten Operanden zurück. So gibt `frozenset('xy') | set('yz')` eine Instanz von `frozenset` zurück.

Das flache Kopieren eines zusammengesetzten Objektes kreiert ein neues zusammengesetztes Objekt und fügt dann, soweit möglich, Verweise auf die im Original gefundenen Objekte ein.

Sowohl `set` als auch `frozenset` unterstützen Vergleiche zwischen Mengen, z. B.

```
>>> set('Vorkurs') == {'V', 'o', 'k', 'u', 'r', 's'}
True
>>> set('Vorkurs') > set('Vor')
True
>>> set('Vor') < set('Vorkurs')
True
```

Zwei Mengen sind nur dann gleich, wenn jedes Element jeder Menge in der anderen enthalten ist (jede Menge ist eine Teilmenge der anderen). Zudem gilt, dass eine Menge nur dann kleiner/größer als eine andere Menge, wenn die erste Menge eine echte Teilmenge/Obermenge der zweiten Menge ist. Instanzen von `set` werden mit Instanzen von `frozenset` anhand ihrer Mitglieder verglichen. Dies bedeutet beispielsweise, dass `set('xyz') == frozenset('xyz')` und `set('xyz') in set([frozenset('xyz')])` beide den Wahrheitswert `True` liefern.

Die Tabelle 13.6 stellt für die veränderlichen Mengen der Klasse `set` verfügbare Operationen zusammen, welche nicht für die unveränderlichen Instanzen der Klasse `frozenset` anwendbar sind. Sie betreffen insbesondere Operationen zur Aktualisierung von Mengen durch Mengenoperationen und durch Einfügen und Entfernen von Elementen. Auch hier akzeptieren die Nicht-Operator-Versionen der Methoden `update()`, `intersection_update()`, `difference_update()` und `symmetric_difference_update()` iterierbare Objekte als Parameter.

Der Parameter `elem` bei den (magischen) Methoden `__contains__()`, `remove()` und `discard()` kann eine Menge sein (vgl. auch Abschnitt 13.4). Um die Suche nach äquivalenten eingefrorenen Mengen zu unterstützen, wird eine temporäre eingefrorene Menge aus `elem` erstellt.

Operation	Beispiel	Bedeutung
len(s)	mebsp1 = len(Menge1)	gibt die Anzahl der Elemente in der Menge s zurück (Kardinalität von s).
x in s	mebsp2 = ('zwei' in Menge1)	testet x auf Zugehörigkeit zu s.
x not in s	mebsp3 = ('zwei' not in Menge2)	testet x auf Nicht-Zugehörigkeit zu s.
isdisjoint(other)	mebsp4 = Menge1.isdisjoint(Menge3)	gibt True zurück, wenn ihre Schnittmenge die leere Menge ist.
issubset(other) oder set <= other	mebsp5 = Menge2.issubset(Menge1) mebsp6 = (Menge2 <= Menge1)	prüfen, ob jedes Element der Menge in der anderen Menge enthalten ist.
set < other	mebsp7 = (Menge2 < Menge1)	prüft, ob die Menge eine echte Teilmenge von other ist.
issuperset(other) oder set >= other	mebsp8 = Menge1.issuperset(Menge2) mebsp9 = (Menge1 >= Menge2)	prüfen, ob jedes Element in other in der Menge enthalten ist.
set > other	mebsp10 = (Menge1 > Menge2)	prüft, ob die Menge eine echte Obermenge von other ist
union(*other) oder set other ...	mebsp11 = Menge2.union([Menge3, Menge4, Menge5]) mebsp12 = Menge2 Menge3	geben eine neue Menge mit Elementen aus der Menge und allen other-Mengen zurück.
intersection(*other) oder set & other & ...	mebsp13 = Menge2.intersection(Menge5) mebsp14 = Menge4 & Menge5	Liefern eine neue Menge mit Elementen, die der Menge und allen other-Mengen gemeinsam sind.
difference(*other) oder set - other - ...	mebsp15 = Menge2.difference(Menge2) mebsp16 = Menge1 - Menge2	geben eine neue Menge zurück, deren Elemente in der Menge set enthalten sind, die nicht in den other-Mengen enthalten sind.
symmetric_difference(other) oder set ^ other	mebsp17 = Menge4.symmetric_difference(Menge5) mebsp18 = Menge4 ^ Menge5	geben eine neue Menge zurück, deren Elemente entweder in der Menge set oder in der Menge other enthalten sind, aber nicht in beiden.
copy()	mebsp19 = Menge1.copy()	gibt eine flache Kopie der Menge zurück.

Tabelle 13.5: Übersicht zu den allgemeinen Operationen für Mengen. Die Variablen `set` und `other` sind dabei Instanzen eines Mengentyps. Im Hinblick auf die Definition der Mengenobjekte `Menge1` bis `Menge4` sei auf die Abbildung 13.9 verwiesen.

```

>>> WB1 = {88188:('Thorsten_Meier', 1980), 88633:('Monika_Schmidt', 1981)}
>>> WB1[88755] = ('Monika_Schneider', 1980)
>>> WB2 = {88266:('Tobias_Müller', 1979), 88711: ('Anna_Schulz', 1982)}
>>> WB1[88633]
('Monika_Schmidt', 1981)
>>> (WB1[88633])[0]
'Monika_Schmidt'

```

Abbildung 13.10: Exemplarische Instanziierung und Verwendung der Wörterbuchobjekte *WB1* und *WB2*

13.4.3 Abbildungen (Mappings)

Der Datentyp Mapping beschreibt eine Möglichkeit zum Zugriff auf Werte (engl. *values*) über einen Schlüssel (engl. *key*). Dies ist ein abweichender Zugriff auf Objekte gegenüber dem Datentyp Liste (s. Abschnitt 13.4.1.1), bei dem der Zugriff auf die Elemente (Werte) der Liste über die Position (Index) erfolgte. Somit entsprechen die Kombinationen aus Schlüssel-Werte-Paaren einer Abbildung (engl. *mapping*) von einem Schlüssel (key) auf seinen zugehörigen Wert (value). Etwas formaler ausgedrückt repräsentiert ein Mapping-Objekt eine Funktion $w(\cdot)$ durch eine endliche Menge von Paaren $(s, w(s))$. Bei Mapping-Objekten besteht der Definitionsbereich, dem s entstammt, aus Hash-fähigen Objekten und der Wertebereich, in dem $w(s)$ liegt, aus beliebigen Objekten. Als Beispiel sei die Abbildungsfunktion genannt, die einer Matrikelnummer (Schlüssel) weitere Daten (Werte) der Studierenden, wie Name und Geburtsjahr, zuordnet.

Derzeit bietet Python nur einen eingebauten Mapping-Typ in Form der veränderlichen Klasse `class dict` an. Der Datentyp `dict` steht dabei für „Dictionary“ (deutsch Wörterbuch oder Lexikon). Die Einträge eines Wörterbuchs ordnen einem Hash-fähigen Schlüssel einen Wert zu.

Wörterbuchobjekte können explizit durch Angabe einer kommasetrennten Liste von Schlüssel:Wert-Paaren in geschweiften Klammern erstellt werden. Zum Zugriff auf die Werte (value) in einem Wörterbuch wird der Schlüssel (key) in eckigen Klammern hinter dem Namen des Wörterbuchs angegeben, z. B. `d[key]`. In dem Beispiel ist der Schlüssel die Matrikelnummer und der Wert ein Tupel aus dem Namen und dem Geburtsjahr. Natürlich kann auch auf die Einzelelemente der Tupel mit den bereits bekannten Methoden zugegriffen werden, z. B. um ein neues Element hinzuzufügen (`d[key]=value`). Analog zur Listenabstraktion und zur Mengenabstraktion steht zudem auch das Konstrukt der Wörterbuchabstraktion zur Verfügung.

Ferner können Wörterbuchobjekte durch einen Klassenaufruf erzeugt werden. Dieser hat zwei Parameter, einen Positionsparameter und eine möglicherweise auch leere Menge `kwargs` von Schlüsselwortparametern (zur Terminologie der Parameter s. Abschnitt 13.3.1). Es gibt drei Aufrufmöglichkeiten, `dict(mapping, **kwargs)`, `dict(**kwargs)` und `dict(iterable, **kwargs)`. Der Positionsparameter bewirkt die Erstellung eines Wörterbuchs aus dem übergebenen Objekt. Handelt es sich bei dem vorliegenden Positionsparameter um ein Mapping-Objekt, d. h. `dict(mapping, **kwargs)`, wird mit denselben Schlüssel-Wert-Paaren wie im Mapping-Objekt ein Wörterbuch erstellt. Fehlt er beim Aufruf, d. h. `dict(**kwargs)`, wird ein leeres Wörterbuch erstellt.

Operation	Beispiel	Bedeutung
update(*others) oder set = other 	Menge2.update([Menge3,Menge4,Menge5 Menge2 = Menge3 Menge4 Menge5	aktualisiert die Menge, indem Elemente aus allen Mengen other hinzugefügt werden.
intersection_update(*others) oder set &= other & ...	mebsp22 = Menge4.intersection_update(Menge5) Menge4 &= Menge5	aktualisiert die Menge, wobei nur die in ihr und allen Mengen other gefundenen Elemente beibehalten werden.
difference_update(*others) oder set -= other ...	Menge1.difference_update(Menge2) Menge1 -= Menge2	aktualisiert die Menge und entfernt Elemente, die in den Mengen other gefunden wurden.
symmetric_difference_update(other) oder set ^ = other	Menge4.symmetric_difference_update(Menge5) Menge4 ^= Menge5	aktualisiert die Menge, wobei nur die Elemente beibehalten werden, die in einer der beiden Mengen, aber nicht in beiden enthalten sind.
add(elem)	Menge5.add(('eins',1))	fügt das Element elem in die Menge ein.
remove(elem)	Menge3.remove('zwei')	entfernt das Element elem aus der Menge, wenn es in ihre enthalten ist, ansonsten gibt es eine Fehlermeldung.
discard(elem)	Menge3.discard('zwei')	entfernt das Element elem aus der Menge, wenn es vorhanden ist.
pop()	mebsp31 = Menge4.pop()	entfernt ein beliebiges Element aus der Menge und gibt es zurück. Es gibt eine Fehlermeldung, wenn die Menge leer ist.
clear()	Menge1.clear()	entfernt alle Elemente aus der Menge.

Tabelle 13.6: Übersicht zu den nur für die veränderlichen Mengen der Klasse `set` verfügbaren Operationen. Die Variablen `set` und `other` sind dabei Instanzen des Mengentyps `set`. Im Hinblick auf die Definition der Mengenobjekte `Menge1` bis `Menge4` sei auf die Abbildung 13.9 verwiesen.

Sofern der Positionsparameter ein iterierbares Objekt ist, d.h. `dict (iterable, **kwargs)`, muss jedes seiner Elemente selbst ein iterierbares Objekt mit genau zwei Objekten sein. Das erste Objekt jedes Elements wird zu einem Schlüssel im neuen Wörterbuch, das zweite Objekt zum entsprechenden Wert. Kommt der Schlüssel mehr als einmal vor, wird der letzte Wert für diesen Schlüssel der entsprechende Wert im neuen Wörterbuch.

Wenn Schlüsselwortparameter angegeben werden, d.h. `kwargs` nicht leer ist, werden die Schlüsselwortparameter und ihre Werte dem Wörterbuch hinzugefügt, welches aus dem Positionsparameter erstellt wurde. Wenn ein hinzugefügter Schlüssel bereits vorhanden ist, ersetzt der Wert aus dem Schlüsselwortparameter den Wert aus dem Positionsparameter.

Die folgenden Beispiele

```
>>> a = dict(one=1, two=2, three=3)
>>> b = dict({'three': 3, 'one': 1, 'two': 2})
>>> c = dict({'one': 1, 'three': 3}, two=2)
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> a
{'one': 1, 'two': 2, 'three': 3}
```

generieren alle das `dict`-Objekt `{'one':1, 'two':2, 'three':3}`, siehe z.B. die Ausgabe zum Objekt `a`. Dabei repräsentieren Wörterbuch `a` die Aufrufmöglichkeit `dict(**kwargs)`, Wörterbücher `b` und `c` `dict(mapping, **kwargs)` sowie Wörterbuch `d` `dict(iterable, **kwargs)`.

Eine Zusammenfassung von Wörterbüchern unterstützenden Operationen vermitteln die Tabellen 13.7 und 13.8. Hierbei ist zu beachten, dass Wörterbuchobjekte die Einfügereihenfolge beibehalten und die Aktualisierung eines Eintrags die Reihenfolge beeinflusst. Zudem werden nach einer Entfernung hinzugefügte Schlüssel am Ende eingefügt. Zwei Wörterbuchobjekte sind gleich, wenn sie die gleichen (Schlüssel, Wert)-Paare haben, unabhängig von der Reihenfolge.

Die von `dict.keys()`, `dict.values()` und `dict.items()` zurückgegebenen Objekte sind sogenannte Wörterbuchansichtsobjekte (engl. *dictionary view objects*). Sie bieten eine dynamische Sicht auf die Wörterbucheinträge, d.h. wenn sich das Wörterbuch ändert, spiegelt die Ansicht diese Änderungen wider. Darauf wird hier aber nicht weiter eingegangen.

13.4.4 for-Anweisung

Den in den vorhergehenden Abschnitten beschriebenen Sammeldatentypen ist gemeinsam, dass in Anwendungsprogrammen vielfach die Anforderung besteht, die Elemente nacheinander oder in speziell definierter Form nacheinander aufzulisten. In der Informatik wird in diesem Fall auch davon gesprochen, dass über die Elemente iteriert wird. In Python erlaubt insbesondere die `for`-Anweisung eine Iteration über iterierbare Objekte, zu denen die eingebauten Datentypen Liste, Tupel, Menge und das Wörterbuch gehören. Daher wird die `for`-Anweisung im Folgenden mit Bezug zu diesen Sammeldatentypen besprochen. Hierbei ist zu beachten, dass das Konzept der `for`-Anweisung von Python spezieller ist als die Konzepte der `for`-Anweisungen von Java oder C++.

Operation	Beispiel	Bedeutung
<code>list(d)</code>	<code>wbsp1 = list(WB1)</code>	gibt eine Liste aller im Wörterbuch <code>d</code> verwendeten Schlüssel zurück.
<code>len(d)</code>	<code>wbsp2 = len(WB1)</code>	gibt die Anzahl der Elemente im Wörterbuch <code>d</code> zurück.
<code>d[key]</code>	<code>wbsp3 = WB1[88755]</code>	gibt das Element von <code>d</code> mit dem Schlüssel <code>key</code> zurück bzw. löst einen <code>KeyError</code> aus, wenn <code>key</code> nicht vorhanden ist.
<code>d[key] = value</code>	<code>WB1[88266] = ('Tobias Müller', 1979)</code>	setzt <code>d[key]</code> auf <code>value</code> .
<code>del d[key]</code>	<code>del WB1[88633]</code>	entfernt <code>d[key]</code> aus <code>d</code> . Löst einen <code>KeyError</code> aus, wenn <code>key</code> nicht vorhanden ist.
<code>key in d</code>	<code>wbsp6 = (88755 in WB1)</code>	gibt <code>True</code> zurück, wenn <code>d</code> einen Schlüssel <code>key</code> hat, sonst <code>False</code> .
<code>key not in d</code>	<code>wbsp7 = (88922 not in WB1)</code>	äquivalent zu <code>not key in d</code> .
<code>iter(d)</code>	<code>wbsp8 = iter(WB1)</code>	gibt einen Iterator über die Schlüssel des Wörterbuchs zurück. Dies ist eine Abkürzung für <code>iter(d.keys())</code> .
<code>clear()</code>	<code>WB1.clear()</code>	entfernt alle Einträge aus dem Wörterbuch.
<code>copy()</code>	<code>wbsp10 = WB1.copy()</code>	gibt eine flache Kopie des Wörterbuchs zurück.
<code>classmethod fromkeys(iterable[, value])</code>	<code>wbsp11 = fromkeys(['eins', 'zwei', 'drei'],0)</code>	erzeugt ein neues Wörterbuch mit Schlüsseln aus <code>iterable</code> und Werten, die auf <code>value</code> gesetzt sind. <code>fromkeys()</code> ist eine Klassenmethode, die ein neues Wörterbuch zurückgibt.
<code>get(key[, standard])</code>	<code>wbsp12 = WB1.get(88633)</code> <code>wbsp13 = WB1.get(88400, ('NN',0))</code>	gibt den Wert für <code>key</code> zurück, wenn <code>key</code> im Wörterbuch enthalten ist, sonst <code>default</code> . Wenn <code>default</code> nicht angegeben wird, ist es standardmäßig <code>None</code> .
<code>items()</code>	<code>wbsp14 = WB1.items()</code>	gibt eine neue Ansicht der Elemente des Wörterbuchs zurück.

Tabelle 13.7: Übersicht 1/2 zu den nur für die Klasse `dict` (Wörterbuch) verfügbaren Operationen. Im Hinblick auf die Definition der Wörterbuchobjekte `WB1` und `WB2` sei auf die Abbildung 13.10 verwiesen.

Operation	Beispiel	Bedeutung
keys()	wbbp15 = WB1.keys()	gibt eine neue Ansicht der Wörterbuchschlüssel zurück.
pop(key[, standard])	wbbp16 = WB1.pop(88633)	Wenn der Schlüssel im Wörterbuch ist, wird er entfernt und sein Wert zurückgegeben, ansonsten default. Wenn default nicht angegeben und key nicht im Wörterbuch ist, wird ein KeyError ausgelöst.
popitem()	wbbp18 = WB1.popitem()	entfernt ein (Schlüssel, Wert)-Paar aus dem Wörterbuch und gibt es zurück. Die Paare werden in LIFO(Last In First Out)-Reihenfolge zurückgegeben.
reversed(d)	wbbp20 = reversed(WB1)	gibt einen umgekehrten Iterator über die Schlüssel des Wörterbuchs zurück. Dies ist eine Abkürzung für reversed(d.keys()).
setdefault(key[, standard])	wbbp21 = WB1.setdefault(88400, ('NN', 0))	wenn key im Wörterbuch enthalten ist, wird sein Wert zurückgegeben. Falls nicht, fügt key den Wert default ein und gibt default zurück. default ist standardmäßig None.
update([other])		aktualisiert das Wörterbuch mit den Schlüssel/Wert-Paaren aus other, wobei vorhandene Schlüssel überschrieben werden. Gibt None zurück.
update()	WB1.update(WB2)	akzeptiert entweder ein anderes Wörterbuchobjekt oder eine Iterable von Schlüssel/Wert-Paaren.
values()	wbbp24 = WB1.values()	gibt eine neue Ansicht der Werte des Wörterbuchs zurück.
d other	wbbp25 = WB1 WB2	erstellt ein neues Wörterbuch mit den zusammengeführten Schlüsseln und Werten von den Wörterbüchern d und other. Die Werte von other haben Vorrang, wenn d und other gemeinsame Schlüssel haben.
d = other	WB1 = WB2	aktualisiert das Wörterbuch d mit den Schlüsseln und Werten von other, das entweder ein Mapping oder eine Iterable von Schlüssel/Wertpaaren sein kann. Die Werte von other haben Vorrang, wenn d und other gemeinsame Schlüssel haben.

Tabelle 13.8: Übersicht 2/2 zu den nur für die Klasse dict (Wörterbuch) verfügbaren Operationen. Im Hinblick auf die Definition der Wörterbuchobjekte WB1 und WB2 sei auf die Abbildung 13.10 verwiesen.

Beispielsweise gibt die `for`-Anweisung über die nachfolgenden Anweisungen

```
>>> studliste = ['Thorsten_Meier', 'Monika_Schmidt', 'Monika_Schneider']
>>> for x in studliste:
...     print(x)
```

die Elemente der Liste `studliste` nacheinander aus.

Die `for`-Anweisung ähnelt auf dem ersten Blick einer `while`-Anweisung, welche schon von Java bekannt ist (s. Abschnitt 4.4). Wir werden aber im Folgenden sehen, dass sich die `for`-Anweisung in Python deutlich davon unterscheidet.

Allgemeiner haben `for`-Anweisungen von Python die Form

```
for Variable in Ausdruck: Anweisungen_1
[else: Anweisungen_2]
```

Die Ausführung beginnt mit der Abarbeitung der Befehle von *Anweisungen_1*, wobei das Ergebnis iterierbar sein muss. Daran schließt sich der Durchlauf der Schleife an. Dabei wird, beginnend mit dem ersten Element, in jedem Iterationsschritt das nächste Element ermittelt, dieses der Variablen *Variable* zugewiesen und dann *Anweisungen_1* durchgeführt. Dieser Durchlauf (Iteration) endet, sofern es kein Element (gemäß dem Durchlauf in der z.B. Liste) mehr gibt. Dann würde *Anweisungen_2* abgearbeitet, falls der optionale `else`-Zweig vorhanden ist. Nachfolgend endet die Ausführung der Schleife.

Zur Steuerung von `for`-Schleifen stehen zusätzliche Anweisungen, `break` und `continue`, zur Verfügung. Die Angabe einer `break`-Anweisung in *Anweisungen_1* ermöglicht es, die Ausführung einer `for`-Anweisung vorzeitig zu beenden. In der Anweisung

```
>>> for x in studliste:
...     if x == 'Monika_Schmidt':
...         break
...     print(x)
```

geschieht dies im Durchlauf für „Monika Schmidt“, in welchem die Schleife vor der Ausführung der `print`-Anweisung beendet wird. Die Ausführung einer `continue`-Anweisung in *Anweisungen_1* bewirkt, dass der Rest von *Anweisungen_1* übersprungen wird. Wenn es ein nächstes Element gibt, wird die Ausführung der `for`-Schleife mit diesem Element fortgesetzt, ansonsten mit dem `else`-Teil, falls dieser vorhanden ist. Würde in dem Beispiel `continue` anstelle von `break` stehen, würden alle drei Listenelemente bearbeitet, aber nur „Thorsten Meier“ und „Monika Schneider“ ausgegeben werden.

Für Listen, Tupeln und Mengen kann mittels `for`-Anweisungen auf die beschriebene Weise über die Elemente iteriert werden. Bei Zeichenketten (Strings) erfolgt die Iteration über die einzelnen Zeichen, die als Listen der Länge 1 zurückgegeben werden. Beispielsweise liefert

```
>>> for char in 'ABC':
...     print(char)
```

als Ausgabe die einelementigen Listen A, B und C. Die Iteration bei Wörterbüchern erfolgt über den Schlüssel, z.B.

```
>>> for key in {'one':1, 'two':2}:
...     print(key)
```

mit der Ausgabe von `one` und `two`. Eine weitergehende Erklärung des Python-Konzepts iterierbarer Objekte und die damit assoziierte Ausführung von `for`-Anweisungen erfolgt im Abschnitt 13.5.5.

Aufgabe 13.7:

Geben Sie die Ergebnisse der Beispielausdrücke zu den Operationen der Sammeldatentypen in den Tabellen 13.3 bis 13.6 an.

Aufgabe 13.8:

Im Python-Quellcode 13.2 sind die Daten, über welcher das Minimum zu suchen ist, in der Variablen `a` vom Typ `list` gegeben.

- Wenn `a` durch eine Menge vom Typ `set` ersetzt wird, d.h. `a = {11, 7, 8, 3, 15, 13, 9, 19, 18, 10, 4}`, ist das Programm nicht mehr korrekt. Geben Sie einen Grund dafür an.
- Formulieren Sie ein vollständiges Python-Programm, welches für `a` als Menge arbeitet.

Aufgabe 13.9:

In Java gibt es mehrdimensionale Arrays, z.B. solche der Dimension 2 mit zwei Indizes, `a[i, j]`. Hingegen ist die mit Arrays verwandte Liste in Python nur eindimensional, d.h. es ist nur ein Index verfügbar, `a[i]`.

- Eine Möglichkeit zur Realisierung mehrdimensionaler Arrays ist die Verwendung geschachtelter Listen, z.B. gemäß `2dliste = [[11,7,8], [3,15,13], [9,19,18]]` für den zweidimensionalen Fall. Auf welche Weise kann mittels Indizes auf den Wert 3 in dieser Struktur zugegriffen werden?
- Die Python-Programmbibliothek `numpy` (Numerical Python, <https://numpy.org>) ist eine etablierte Erweiterung für Python, welche u.a. umfassende Datenstrukturen zur Verfügung stellt, z.B. auch mehrdimensionale Arrays. Die Funktionalität erfolgt nicht auf Basis von Python-Sammeldatentypen, sondern ist zur effizienteren Ausführung direkt implementiert. Die `numpy`-Arrays sind von großer Bedeutung für Verfahren des Maschinellen Lernens der Künstlichen Intelligenz, die sogenannte Tensoren verwenden. Zu nennen ist hier auch `PyTorch` (<https://pytorch.org>), eine Entwicklungsumgebung des Maschinellen Lernens [18].

Installieren Sie die Programmbibliothek `numpy`, importieren Sie diese (`import numpy as np`), erstellen Sie ein zweidimensionales Array, z.B. gemäß `a=np.array([[11,7,8], [3,15,13], [9,19,18]])` und geben Sie das Element 3 durch Indexzugriff auf das Array aus.

Hinweis: Sie können die vorliegenden Teilaufgaben auch zunächst auslassen, da die Teilschritte evtl. etwas kompliziert sind.

Aufgabe 13.10:

Ersetzen Sie die `while`-Schleife innerhalb des Quellcodes 13.2 durch eine `for`-Schleife.

Hinweis: Beispielsweise könnte eine `for`-Schleife über eine `range`-Instanz oder eine `for`-Schleife über die Liste `a` verwendet werden. `for`-Schleifen erlauben unter anderem das sukzessive Ablaufen der Elemente einer `range`-Instanz in der Form „`for i in range[aktuelle Parameter]: Anweisung`“.

Aufgabe 13.11:

Es soll ein Wörterbuchobjekt für eine Vorlesungsverwaltung erstellt werden. Der Schlüssel eines Eintrags soll ein Tupel sein, welches sich aus einer Zeichenkette des Vorlesungsnamens und einer ganzen Jahreszahl

zusammensetzt, z. B. (`'Mathematik', 2022`). Der Wert eines Eintrags soll eine Menge vom Typ `set` sein, welche die Menge der Matrikelnummern von Teilnehmenden der Vorlesung enthält.

- a) Begründen Sie die Zulässigkeit der vorgeschlagenen Schlüsseldefinition.
- b) Legen Sie ein Wörterbuchobjekt `vorlsgadmin` an, welches als initial anzulegendes Objekt folgende Information enthält:
 - Vorlesung „Mathematik“, 2022, Teilnehmende 88329, 88431, 88349,
 - Vorlesung „Programmierung“, 2021, Teilnehmende 88329, 88431, 88421, 88212,
 - Vorlesung „Rechnerstrukturen“, 2022, Teilnehmende 88678, 88631, 88721.

13.5 Objektorientierte Programmierung

Vergleichbar zu den Programmiersprachen Java und C++ erlaubt auch Python das Konzept der objektorientierten Programmierung umzusetzen. Die Idee ist es, zusammengehörige Daten und Methoden zu kapseln und als Klassen zu definieren. Über diese Klassendefinition können dann verschiedene Objekte von einem konsistenten Grundtypus erzeugt werden. Für eine weitergehende Einführung in das Programmierparadigma der „Objektorientierten Programmierung“ (OOP) sei auf das Kapitel 9 verwiesen.

13.5.1 Klassen

Vergleichbar zu den Programmiersprachen Java und C++ erlaubt auch Python, für Strukturen mit gleicher Funktionalität eine Klasse zu definieren, um entsprechend identisch aufgebaute Objekte (Instanzen) erzeugen zu können. Schauen wir uns dieses Prinzip der objektorientierten Programmierung für das bereits bekannte Beispiel von Studierenden an (vgl. Abschnitt 7.1 für Java). Der Quellcode 13.8 setzt dies exemplarisch für Python um. Eingeleitet wird die grundlegende Definition einer Klasse durch das Schlüsselwort `class`, gefolgt von dem Namen der Klasse. Abgeschlossen wird die Zeile mit einem Doppelpunkt (s. Zeile 1). Die Zugehörigkeit zum Block der Klassendefinition wird auch in diesem Fall durch die Einrückung festgelegt.

Um die nachfolgenden Erklärungen besser verstehen zu können, werden die verschiedenen Mechanismen innerhalb des Quellcodes 13.9 am Beispiel einer Klasse `Studierende` exemplarisch verdeutlichen. Damit wir diese Funktionalitäten der Klasse `Studierende` innerhalb der `main`-Funktion (s. Abschnitt 13.3) nutzen können, müssen sie zuerst importiert werden. Dies erfolgt über die vielleicht etwas eigentümlich erscheinende Anweisung in der ersten Zeile. Die Anweisung besagt, dass die Klasse `Studierende` aus der Datei `Studierende.py` importiert werden soll. Da es in Python erlaubt ist, mehrere Klassen innerhalb einer Datei zu definieren, ist diese genaue Angabe notwendig. Sofern einfach alle Klassen importiert werden sollen, die innerhalb einer Datei definiert sind, wäre das auch mittels der Anweisung `„from Studierende import *“` möglich.

Quellcode 13.8: Definition der Klasse Studierende

```

1 class Studierende:
2
3     def __init__(self, name='', nummer=0, jahr=0):
4         self.studname = name
5         self.matrikelnummer = nummer
6         self.geburtsjahr = jahr
7         print('Erzeuge Studierendenobjekt')
8
9     def __del__(self):
10        print('Loesche Studierendenobjekt')
11
12    def gibStudname(self):
13        return self.studname
14
15    def setzeStudname(self, name):
16        self.studname = name
17
18    def gibMatrikelnummer(self):
19        return self.matrikelnummer
20
21    def setzeMatrikelnummer(self, nummer):
22        self.matrikelnummer = nummer
23
24    def info(self):
25        print('Name:', self.studname,
26            ' Matrilnummer:', self.matrikelnummer,
27            ' Geburtsjahr:', self.geburtsjahr)

```

13.5.2 Instanzmethode

Nach den Erklärungen zum grundlegenden Aufbau einer Klasse widmen sich die nachfolgenden Erklärungen den wesentlichen Definitionen, die über allgemeine und spezielle Instanzmethoden erfolgen. Unter Instanzmethoden werden die bisher von Java und C++ bekannten Methoden verstanden, welche auf Instanzen/Objekte einer Klasse aufgerufen werden können. Im Folgenden wird zunächst einfach der Begriff Methode genutzt, welches aber im Abschnitt 13.5.3.1 noch genauer charakterisiert wird. Methoden werden dabei vergleichbar zu Funktionen durch das einleitende Schlüsselwort `def` definiert, welches auch durchgängig in dem Beispiel zu sehen ist.

Eine Besonderheit ist der Aufbau der Parameterliste von Methoden, welcher dem Schema

```
def Methodenname (self, Parameter_1, ..., Parameter_n):
```

folgt. Der erste Parameter ist immer eine Referenz auf das aktuell instanziierte Objekt. Obwohl der Name dieses ersten Parameters prinzipiell frei gewählt werden kann, ist es in Python eine durchgehend eingehaltene Konvention, diesen mit `self` zu bezeichnen. Nach diesem speziellen

Parameter folgen die „normalen“ Parameter. Abgeschlossen wird die einleitende Deklaration wieder mit einem Doppelpunkt.

Die Zuweisung von Parameterwerten an die Objektvariablen erfolgt dann wieder unter Verwendung des Schlüsselworts `self` und der Adjunktion per Punkt. Es ergibt sich der folgende Aufbau `self.Objektvariable = Parameter`. Beispielsweise wird in der Zeile 4 der Wert des Parameters `name` der Objektvariablen `studname` zugewiesen. Die Definition von Objektmethoden erfolgt nach dem gleichen Prinzip. Nach dem Schlüsselwort `def` wird der Name der Methode angeführt, nachfolgend in Klammern die Parameter. Der erste Parameter ist wieder die Referenz auf das aktuelle Objekt (Bezeichner `self`). Entsprechend ist z. B. in der Zeile 15 der Parameter `name` an zweiter Position aufgelistet. Die Zuweisung an das Objektattribut `studname` erfolgt erneut über die Referenz `self` per Punktnotation.

Die Instantiierung einer Klasse geschieht, ähnlich einem Funktionsaufruf, durch Aufruf der Klasse. Es besteht jedoch die Möglichkeit, die Instantiierung mit einer Initialisierung des erzeugten Objekts zu verbinden. Im Unterschied zu Java und C++ (s. Abschnitt 7.2) und C++ (s. Abschnitt 12.4) gibt es in Python aber keine explizite Deklarationsmöglichkeit von Konstruktoren. Hierfür steht die eingebaute Initialisierungsmethode `__init__` zur Verfügung (Zeile 4 des Beispiels). Sie wird nach erfolgter Instantiierung des Objekts durch den Klassenaufruf automatisch aufgerufen. Die notwendigen Parameterwerte erhält sie über eine entsprechende Parameterangabe beim Klassenaufruf (s. auch Abschnitt 13.5.1). In Analogie zu Programmiersprachen wie Java oder C++ wird teilweise aber trotzdem von einem Konstruktor gesprochen.

Im Gegensatz zu Java oder C++, bei denen mehrere Konstruktoren definiert werden konnten, ist in Python nur die Definition einer `__init__`-Methode vorgesehen. Es gibt aber verschiedene Mechanismen um die Instantiierung einer Klasse situationsabhängig zu erstellen. Zur situationsabhängigen Instantiierung einer Klasse können die gleichen Parameterübergabemechanismen wie für Funktionen genutzt werden, s. Abschnitt 13.3.1. Ein Mechanismus ist die Festlegung von Defaultwerten, siehe Zeile 3 des Beispiels. Hinter den Parametern wird über ein Gleichheitszeichen ein Wert festgelegt, der dann der Objektvariablen zugewiesen wird, falls der Parameter nicht angegeben wurde. Bitte beim Parameter `name` beachten, dass Strings durch einfache Anführungszeichen ausgedrückt werden (s. Abschnitt 13.4.1.4). Da die Reihenfolge der Parameter festgelegt ist, können Parameter nur ab einer Position weggelassen werden, nicht aber zwischendurch. Im Hinblick auf weitere Details sei auf umfassende Einführungen in die Programmiersprache Python verwiesen [16, 19].

Nach den Ausführungen zum grundlegenden Aufbau von allgemeinen und speziellen Instanzmethoden wird im Folgenden die Instanziierung von Objekten vom Typ `Studierende` besprochen – dies am Beispiel des Quellcodes 13.9. Innerhalb der Zeile 4 wird eine Instanz der Klasse `Studierende` erstellt und ihre Referenz analog zu Java (s. Kapitel 7), der Objektvariablen `stud_1` zugewiesen. In Klammern werden die drei Werte (ohne `self`) der Parameter übergeben. Mit dem Aufruf der Klasse wird dann die spezielle `__init__`-Methode ausgeführt. Damit die Wertzuweisung an die Objektattribute besser nachzuvollziehen ist, wird in Zeile 5 die Methode `info` für Objekte aufgerufen. Es bietet sich jetzt an, die `main`-Funktion einmal aufzurufen, um die Erklärungen besser verstehen zu können. Durch die Festlegung von Defaultwerten für die Parameter innerhalb der `__init__`-Methode ist es aber auch möglich, beim Aufruf keine Werte

Quellcode 13.9: Exemplarische Methode zur Verwendung der Klasse Studierende

```
1 from Studierende import Studierende
2
3 def main():
4     stud_1 = Studierende('Thorsten Meier', 88188, 1980)
5     stud_1.info()
6
7     stud_2 = Studierende()
8     stud_2.info()
9     stud_2.setzeStudname('Monika Schmidt')
10    stud_2.setzeMatrikelnummer(88633)
11    stud_2.geburtsjahr = 1981
12    stud_2.info()
13
14    stud_3 = Studierende('Monika Schneider')
15    stud_3.info()
16    stud_3.setzeMatrikelnummer(1980)
17    stud_3.info()
18
19    del stud_1, stud_2, stud_3
20
21 main()
```

zu übergeben (leere Klammer). In der Zeile 7 wird nach diesem Mechanismus eine zweite Instanz der Klasse mit dem Namen `stud_2` erstellt. Die Objektparameter entsprechen somit den Defaultwerten. Beispielsweise hat der Parameter `geburtsjahr` den Wert „0“. Wir können aber jetzt im nächsten Schritt (Zeilen 9-11) über die Methoden der Klasse den Objektparametern entsprechende Werte zuweisen. Der Mechanismus von Defaultwerten kann auch genutzt werden, um nur einige (der ersten) Parameter anzugeben. Dies zeigt exemplarisch Zeile 14. Dem Objekt `stud_3` wird ein Name zugewiesen. Die Matrikelnummer und das Geburtsjahr werden hingegen über die Defaultwerte gesetzt.

Analog zu `__init__` gibt es eine eingebaute Methode `__del__`. Sie wird vom Python-System aufgerufen, wenn es ein Objekt entfernt. Objekte können entfernt werden, wenn sie nicht mehr referenziert werden. Eine explizite Minderung der Anzahl der Referenzen wird insbesondere durch die `del`-Anweisung bewirkt. Nach Anwendung auf einen Variablennamen ist diese Variable nicht mehr existent (sie wird aus der aktuellen Symboltabelle entfernt). Da damit auch die Referenz der Variable auf das Objekt verloren geht, vermindert sich die Anzahl der Referenzen um 1. Die Anwendung von `del` in Zeile 19 des Beispiels reduziert die Anzahl der Referenzen auf die betroffenen Objekte auf 0, sodass dadurch der Aufruf von `__del__` bewirkt wird und das Objekt gelöscht wird. Zeile 19 zeigt zudem, dass die aufeinanderfolgende Anwendung einer gleichartigen Anweisungsart in Python (Tupel-artig) zusammengefasst werden kann. Es wäre auch möglich gewesen, jede der drei Variablen einzeln zu entfernen..

Quellcode 13.10: Erweiterte Variante der Klasse Studierende

```

1 class Studierende:
2     __sumStudierende = 0
3
4     def __init__(self, name='', nummer=0, jahr=0):
5         self.__studname = name
6         self.__matrikelnummer = nummer
7         self.geburtsjahr = jahr
8         Studierende.__sumStudierende += 1
9
10    def __del__(self):
11        Studierende.__sumStudierende -= 1
12
13    # ...
14    # Zeilen 12-27 aus dem Quellcode 13.3
15    # ...
16
17    @classmethod
18    def anzahlStudierende(cls):
19        return cls.__sumStudierende

```

13.5.3 Spezielle Eigenschaften

Im Folgenden wird noch auf einige erweiterte Konzepte im Zusammenhang mit Klassen und Objekten eingegangen. Hierzu zeigt der Quellcode 13.10 eine Abwandlung der bereits bekannten Klasse Studierende (s. Quellcode 13.8). Dabei wurden die identischen Zeilen 12-27 im Sinne der Übersichtlichkeit weggelassen. An dieser Stelle noch einmal der Hinweis, dass alle im Buch besprochenen Quellcodes über die Internetseite <http://www.vorkurs-informatik.de/buch> zum Buch verfügbar sind.

13.5.3.1 Instanz- und Klassen-bezogene Attribute und Methoden

Ein erster Aspekt, der genauer betrachtet werden soll, ist die Unterscheidung zwischen *Instanzattribute* und *Klassenattribute*. Alle bisher in der Klasse Studierende vorliegenden Attribute `studname`, `matrikelnummer` und `geburtsjahr` sind *Instanzattribute*. Diese Attribute werden jeweils individuell bei der Generierung (Instanziierung) für ein neues Objekt einer Klasse erstellt. Die Werte der *Instanzattribute* können sich somit zwischen den Objekten unterscheiden. Somit verändern sich ihre Werte mit Bezug zum jeweiligen Objekt. Daher wird auch von dynamischen Attributen gesprochen. Im Gegensatz dazu gibt es sogenannte statische Attribute, die auch als *Klassenattribute* bezeichnet werden. Diese Attribute haben einen Wert pro Klasse und unterscheiden sich nicht zwischen den Objekten der Klasse. Klassenattribute müssen nicht im Vergleich zu Java mit dem Schlüsselwort `static` gekennzeichnet werden (vgl. Abschnitt 5). Attribute außerhalb der Definitionsblöcke einer Klasse haben automatisch diese Eigenschaft. Ei-

ne Konvention ist es nur, diese nach der Zeile mit dem Schlüsselwort `class` anzuführen. In dem Beispiel (s. Quellcode 13.10) ist das Attribut `sumStudierende` (Zeile 2) statisch (auf die Bedeutung der beiden Unterstriche wird erst in Abschnitt 13.5.3.3 eingegangen). Sie soll dazu dienen, die jeweils aktuelle Anzahl an Objekten vom Typ `Studierende` abzuspeichern. Entsprechend müssen wir den Wert des Attributs `sumStudierende` bei der Erstellung eines Objektes innerhalb der Methoden `__init__` immer um eins erhöhen (Zeile 8).

Erkennbar ist auch ein Unterschied im Zugriff. Während der Zugriff bei Instanzattributen über das Schlüsselwort `self` (Referenz auf das jeweils aktuelle Objekt) erfolgt, kommt bei Klassenattributen der Name der Klasse zum Tragen – in diesem Fall somit `Studierende`. In der Deklaration der Methode `__del__` wird der Wert des Attributs `sumStudierende` ebenfalls entsprechend angepasst (Zeile 11).

Vergleichbar zu den Attributen ist es auch bei Methoden möglich, diese mit Bezug zur Instanz oder zur Klasse zu deklarieren. Zudem können wir in Erweiterung von den Attributen auch Methoden definieren, welche unabhängig von der Klasse und den Instanzen eine Klasse sind. Entsprechend unterscheiden wir zwischen den nachfolgenden Ausprägungen:

- Instanzmethode („`def Methodenname`“): Methode mit Referenz zum jeweiligen Objekt (Instanz) der Klasse
- Klassenmethode („`@classmethod def Methodenname`“): Methode mit Referenz zur Klasse aber ohne Referenz zum jeweiligen Objekt
- Statische Methode („`@staticmethod def Methodenname`“): Methode ohne Referenz zur Klasse und zu den Objekten der Klasse.

Instanzmethoden wurden schon im Abschnitt 13.5.2 besprochen. Alle Methoden, z.B. `gib-Studname`, welche wir bisher in der Klasse `Studierende` definiert hatten, sind von diesem Typus. Sie zeichnen sich dadurch aus, dass über den ersten Parameter `self` ein Zugriff auf die jeweiligen Attribute des individuellen Objektes möglich ist.

Klassenmethoden verhalten sich vergleichbar zu Klassenvariablen. Daher bietet es sich an, eine Klassenmethode zu schreiben, welche den Wert unseres Klassenattributs `sumStudierende` als Rückgabewert beinhaltet. In den Zeilen 17-19 wird eine entsprechende Klassenmethode `anzahlStudierende` definiert. Klassenmethoden werden dabei durch das vorangestellte Schlüsselwort `@classmethod` gekennzeichnet. Die Schlüsselwörter `@classmethod` und `@staticmethod` werden auch als *Dekorator* bezeichnet. Eine zweite Unterscheidung ist der erste Parameter. Dieser mit `cls` bezeichnete Parameter erlaubt einen Zugriff auf die Attribute der Klassen. Diesen Mechanismus nutzen wir in der Zeile 19 zur Festlegung des Rückgabeparameters. Die Ausführung der Zeilen 4 bis 9 im zugehörigen Testprogramm (Quellcode 13.11) macht die Anwendung und Wirkung der neuen Konzepte deutlich.

Statische Methoden, welche durch das vorangestellte Schlüsselwort `@staticmethod` gekennzeichnet werden, erlauben es zudem Methoden zu definieren, welche unabhängig von einem Bezug den Attributen der Klasse und der Objekte sind. Beispielsweise könnte eine Methode definiert werden, welche zwei Objekte der Klasse als Parameter hat und einen Vergleich zwischen diesen ausführt.

Quellcode 13.11: Testprogramm zur Erweiterung der Klasse Studierende

```
1 from Studierende_2 import Studierende
2
3 def main():
4     print('Anzahl Studierender:', Studierende.anzahlStudierende())
5     stud_1 = Studierende('Thorsten Meier', 88188, 1980)
6     print('Anzahl Studierender:', Studierende.anzahlStudierende())
7
8     stud_1.abschluss = True
9     print(stud_1.__dict__)
10    print(stud_1)
11
12 main()
```

Zum Abschluss dieses Abschnitts soll noch eine besondere Eigenschaft im Zusammenhang mit Attributen betrachtet werden. In der Klasse Studierende wurden bekanntlich die Instanzattribute `studname`, `matrikelnummer` und `geburtsjahr` definiert. In Python hat jede erzeugte Instanz einer Klasse einen eigenen Gültigkeitsbereich für Attribute (s. Abschnitt 5.3 für Java). Daher haben wir auch von Instanzattributen gesprochen. Im Gegensatz zu Java und C++ ist es dadurch in Python aber auch möglich, dynamisch jede Instanz um neue (Instanz-)Attribute zu ergänzen. Beispielsweise wird im Testprogramm das Objekt `stud_1` um das Attribut `abschluss` erweitert (Zeile 8). In den meisten Anwendungsfällen sollte diese Möglichkeit nicht zum Tragen kommen. Besser ist es, einen konsistenten Aufbau von Instanzen vorliegen zu haben.

13.5.3.2 Objektinformationen

Python bietet zudem noch recht komfortabel die Option, eine Übersicht zu den Attributen und aktuell vorliegenden Werten für jede Instanz zu erhalten. Hierzu steht das spezielle Attribut `__dict__` zur Verfügung. Das Attribut ist in Python integriert. Wir müssen uns somit nicht weiter um die Definition kümmern. Wenn wir uns in unserem Testprogramm das Attribut für das Objekt `stud_1` mittels der Methoden `print` ausgeben lassen (s. Zeile 9), resultiert die Ausgabe

```
{'_Studierende__studname': 'Thorsten Meier',
 '_Studierende__matrikelnummer': 88188,
 '_Studierende__geburtsjahr': 1980}.
```

Wir sehen z.B., dass das Objekt `stud_1` eine Instanz der Klasse `Studierende` ist, welche u. a. das Attribut `studname` beinhaltet und dieses Attribut z.Z. den Inhalt „Thorsten Meier“ aufweist. Mit der Anweisung können wir auch kontrollieren, ob evtl. Instanzattribute hinzugefügt wurden. Entsprechend liefert die Anweisung die Ausgabe, dass das Objekt `stud_1` durch die Anweisung in Zeile 8 das zusätzliche Attribut `abschluss` beinhaltet. Der Aufbau ist schon von den Ausführungen zu den Sammeldatentypen im Abschnitt 13.4 bekannt. Die Repräsentation erfolgt als Wörterbuch (*Attributlexikon*). Entsprechend könnten wir damit auch Operationen für Wörterbücher nutzen.

Einen zweiten Mechanismus, um Informationen über das jeweilige Objekt zu erhalten, zeigt beispielhaft die Zeile 10 des Testprogramms. Wenn der Name eines Objekts einer Klasse als Argument der Methoden `print` übergeben wird, resultiert als Ausgabe der Name der Klasse inklusive dem Dateinamen, in dem die Klasse definiert ist als auch eine Information zur Speicheradresse, ab der sich das Objekt befindet. Für das vorliegende Beispiel könnte somit die Ausgabe

```
<Studierende_2.Studierende object at 0x00000197DCFF5000>.
```

ergeben. Die Angabe zur Speicheradresse ist individuell und wird daher bei einer erneuten Programmausführung höchst wahrscheinlich eine andere sein. Zum Verständnis des Zusammenhangs zwischen Objekten und Speicher sei auf die Erklärungen zur Programmiersprache C++ verwiesen (s. Abschnitt 12.3) und auf das Kapitel 19 zur Einführung in die allgemeine Rechnerarchitektur. Im Zusammenhang mit der Anweisung in der Zeile 10 ist noch eine weitere Besonderheit zu beachten. Prinzipiell „im Hintergrund“ wird die (magischen) Methode `__repr__()` aufgerufen. Es handelt sich dabei um eine weitere interne Methode, welche für jedes Objekt zur Verfügung steht. Sofern wir die Methode nicht selber individuell definieren, erhalten wir die angeführte Defaultausgabe.

13.5.3.3 Sichtbarkeit

Ein Mechanismus der objektorientierten Entwicklung ist auch, die Sichtbarkeit und den Zugriff auf Instanzattribute einzuschränken oder nur einen kontrollierten Zugriff zu erlauben. Dies war auch die Motivation für die `get`- und `set`-Methoden in der Klasse `Studierende`. Obwohl diese Methoden vorliegen, war es gemäß der Definition im Quellcode 13.8 möglich, den Wert der Attribute auch direkt zu ändern, z. B. mittels der Anweisung `stud_1.matrikelnummer = 12345`. Um die Sichtbarkeit (lesender und schreibender Zugriff) von Informationen einzuschränken, wurden im Abschnitt 9.4 bereits am Beispiel von Java verschiedene Konzepte eingeführt. Im Gegensatz zur Programmiersprache Java, in der bestimmte Schlüsselwörter, z. B. `private`, als Modifikatoren zur Beeinflussung der Sichtbarkeit notwendig waren, fungiert das Vorhandensein von ein oder zwei Unterstrichen als Modifikator. Eine Übersicht der drei verschiedenen Ausprägungen vermittelt die Tabelle 13.9. Innerhalb des Quellcodes 13.10 sehen wir hierzu entsprechende Beispiele. Sowohl dem Klassenattribut `sumStudierende` und dem Instanzattribut `studname` sind zwei Unterstriche vorangestellt, wodurch für beide Attribute die Sichtbarkeit *private* festgelegt ist. Für das Attribut `matrikelnummer` gilt durch das Voranstellen von nur einem Unterstrich die Sichtbarkeit *protected*. Keine Einschränkung der Sichtbarkeit (*public*) gilt hingegen für das Attribut `geburtsjahr`, da kein Modifikator vorliegt. In einer realen Anwendung würde es sich vermutlich anbieten, alle Attribute als *private* zu definieren. Die vorliegende Ausprägung sollte auch nur als Beispiel zur Verwendung unterschiedlicher Modifikatoren dienen.

13.5.4 Vererbung

Ein weiteres Prinzip der objektorientierten Modellierung ist das Prinzip der Vererbung, welches im Abschnitt 9.3 unter Verwendung der Programmiersprache Java eingeführt wurde. Die Idee ist es, Klassen in einer hierarchischen Beziehung zu definieren. Dabei sind die abgeleiteten Klassen

Modifikator	Konstruktion	Beispiel	Bedeutung
public	kein Unterstrich	bezeichner	Kein expliziter Modifikator (Unterstrich) bedeutet, dass (von außen) auf Attribute in allen Instantiierungen des Objektes zugegriffen werden kann.
protected	ein Unterstrich	_bezeichner	Wird ein Unterstrich angegeben, ist der Zugriff (lesen und schreiben) auf die Attribute der instanziierten Objekte weiterhin möglich. Die Programmierenden bringen mit diesem Modifikator aber zum Ausdruck, dass der Zugriff nicht gewünscht ist.
private	zwei Unterstriche	__bezeichner	Der Modifikator (zwei Unterstriche) bewirkt die stärkste Einschränkung. Die Attribute sind nicht sichtbar und es ist kein lesender sowie schreibender Zugriff (von außen) erlaubt.

Tabelle 13.9: *Einschränkung der Sichtbarkeit für Attribute und Methoden*

(Unterklassen) eine Erweiterung oder Spezialisierung von einer (allgemeinen) Oberklasse bzw. mehreren Oberklassen. Die Unterklassen „erben“ die Attribute und Methoden der Oberklassen. Diese können aber auch verändert oder ergänzt werden. Um die damit verbundenen Konzepte von Python zu verdeutlichen, gehen wir von der bereits bekannten Klasse `Studierende` aus. Da es eine sehr allgemeine und wenig spezifische Form aller Studierenden ist, fungiert sie als Oberklasse. Zwei typische Spezialisierungen sind `Studierende` im Bachelor oder Master. Somit sind die Klassen `Bachelor` und `Master` zwei Unterklassen der Oberklasse `Studierende`. Der Quellcode 13.12 zeigt eine mögliche Umsetzung in Python. Da es uns um die Verdeutlichung der Vererbung geht, sind die Klassen recht einfach aufgebaut. Grundsätzlich wird eine Vererbung dadurch zum Ausdruck gebracht, dass die `class`-Deklaration der Unterklasse um den Namen der Oberklasse in Klammern gemäß der Anweisungsform

```
class Unterklassenname (Oberklassenname):  
  
    erweitert wird.
```

Neben der Einfachvererbung (eine Oberklasse) unterstützt Python auch die Mehrfachvererbung. Die Oberklassen werden dann durch Komma getrennt in der Klamme eingeführt. Soll beispielsweise die Klasse `Bachelor` von den Klassen `Studierende` und von einer weiteren Klasse `HiWi` erben, kann das über die Anweisung `class Bachelor(Studierende, HiWi)` realisiert werden.

Quellcode 13.12: Prinzip der Vererbung zwischen der Oberklasse Studierende und den Unterklassen Bachelor und Master

```
1 class Studierende:
2     def __init__(self, name=''):
3         self.studname = name
4
5     def gibStudname(self):
6         return self.studname
7
8     def pruefungen(self):
9         print('Nicht bekannt')
10
11 class Bachelor(Studierende):
12     def __init__(self, name=''):
13         super().__init__(name)
14
15     def pruefungen(self):
16         print('Pruefungsthemen')
17
18     def bachelorarbeit(self):
19         print('Titel der BA')
20
21 class Master(Studierende):
22     pass
```

Im Folgenden werden nur einfache Vererbungen betrachtet. Im entsprechenden Quellcode 13.12 wird in der Zeile 11 für die Unterklasse `Bachelor` zur Oberklasse `Studierende` verdeutlicht bzw. in der Zeile 18 für die Klasse `Master`. Da wir die Struktur der Klasse nicht weiter deklariert haben, können wir für diesen Fall das Schlüsselwort `pass` nutzen (Zeile 19). Es weist den Python-Interpreter an, direkt fortzufahren. Zu beachten ist, dass die Oberklasse am Ort der Deklaration der abgeleiteten Klasse bekannt ist. Im vorliegenden Fall sind alle drei Klassen in einer Datei `Studium.py` definiert. Wären die drei Klassen auch in drei separaten Dateien definiert, müsste die Oberklasse in den Dateien der Unterklassen per `import`-Anweisung eingebunden werden.

Obwohl der Aufbau von Klassen bei der Vererbung dem bekannten Aufbau folgt, gibt es einige ergänzende Konzepte, um eine Verbindung zwischen Ober- und Unterklasse zu gewährleisten. Teilweise ist es z. B. notwendig, von einer Unterklasse explizit auf die Oberklasse zuzugreifen. Hierzu stellt Python die Funktion `super()` zur Verfügung. Da die Funktion jeweils auf die direkte Oberklasse verweist, ist auch kein Parameter notwendig. Dieser Mechanismus wird in der Zeile 13 des Quellcodes 13.12 genutzt, um innerhalb der `__init__`-Methode der Unterklasse `Bachelor` die entsprechende `__init__`-Methode der Oberklasse `Studierende` aufzurufen. In einem komplexeren Beispiel würde über dieses Prinzip die Instanzattribute der Oberklassen deklariert, um sie auch in der Unterklasse nutzen zu können. In vergleichbarer Form könnte auch auf andere Methoden der Oberklasse zugegriffen werden. Im Falle einer mehrstufigen Ver-

bungshierarchie, z. B. `Mensch` → `Studierende` → `Bachelor`, ist aber zu beachten, dass die Funktion `super` innerhalb der Klasse `Bachelor` auf die direkte Oberklasse `Studierende` verweist und nicht auf Klasse `Mensch`.

13.5.4.1 Polymorphismus

Neben dem Zugriff auf Methoden der Oberklasse über die Funktion `super()` ist die unterschiedliche Verfügbarkeit und Funktionalität von Methoden ein weiteres Prinzip der Vererbung. Dieses, als Polymorphismus bezeichnete Prinzip wurde im Abschnitt 9.5 eingeführt. Bei der Deklaration polymorpher Methoden wurden die Konzepte Überladung und Überschreibung unterschieden. Überladung bedeutet, dass zwei Methoden (in der Ober- und Unterklasse) den selben Namen haben, sich aber bzgl. der Parameterlisten unterscheiden. Sofern zwei Methoden (in der Ober- und Unterklasse) sowohl den selben Namen als auch identische Parameterlisten aufweisen, wird von Überschreibung gesprochen – die Methoden haben dann im Allgemeinen aber eine unterschiedliche Funktionalität. Sofern Ihnen die Begrifflichkeiten nicht mehr komplett bekannt sind, sollten Sie sich evtl. die Einführung am Beispiel der Programmiersprache Java ab Seite 170 noch einmal durchlesen.

Im Gegensatz zu Java erlaubt Python nur das Überschreiben, nicht das Überladen. Im Folgenden schauen wir uns dies am Beispiel des Quellcodes 13.13 an. Nach der Instanziierung eines Objekts vom Typ `Bachelor` und Referenzierung an die Variable `stud_2` können Methoden aus der Ober- und Unterklasse genutzt werden. Objekte vom Typ `Bachelor` erben in dem vorliegenden Beispiel die Methode `gibStudname()` der Oberklasse `Studierende`. Daher ist der Methodenaufruf `stud_2.gibStudname()` in der Zeile 8 erlaubt. Die Methoden `bachelorarbeit()`, um welche die Unterklasse `Bachelor` erweitert wurde, ist für Objekte der Klasse selbstverständlich auch gültig (s. Zeile 9). Mittels der Anweisung `stud_2.pruefungen()` in der Zeile 10 wird schließlich das Prinzip des Überschreibens von Methoden deutlich. Die Methode `pruefungen` wurde in der Oberklasse `Studierende` bereits definiert, in der Unterklasse `Bachelor` überschrieben. Somit steht Objekten vom Typ `Bachelor` nur die Funktionalität der Methode gemäß der Definition in der Unterklasse zur Verfügung. Die Anweisung `stud_2.pruefungen()` generiert daher die Ausgabe „Prüfungsthemen“.

13.5.4.2 Typüberprüfung

Im Zusammenhang mit der Einführung der eingebauten Sammeldatentypen in Python (s. Abschnitt 13.4) wurden bereits verschiedene Möglichkeiten zur Überprüfung und zum Vergleich von Datentypen besprochen. Hierzu wurden im Kern die Funktionen `type` und `isinstance` genutzt. Diese sind auch im Kontext der objektorientierten Entwicklung von Bedeutung, um nähere Informationen zu Klasseninstanzen zu erhalten. Der Quellcode 13.14 verdeutlicht dies exemplarisch. Importiert werden wieder die Klasse `Studierende` und `Bachelor` gemäß dem Quellcode 13.13. In den Zeilen 4 und 5 werden zwei Instanzen der Klassen erstellt. Für beide Objekte kann mit der Funktion `type` (ohne weitere Parameter) der entsprechende Typ abgefragt werden. Die Anweisungen in den Zeilen 7 und 8 generieren die nachfolgende Ausgabe:

Quellcode 13.13: Testprogramm zum Prinzip der Vererbung am Beispiels des Quellcodes 13.12

```

1 from Studium import *
2
3 def main():
4     stud_1 = Studierende('Thorsten Meier')
5     stud_1.pruefungen()
6
7     stud_2 = Bachelor('Monika Schmidt')
8     print(stud_2.gibStudname())
9     stud_2.bachelorarbeit()
10    stud_2.pruefungen()
11
12 main()

```

```

<class 'Studium.Studierende'>
<class 'Studium.Bachelor'>.

```

Ergänzend zum Namen der Klasse (z. B. Studierende) wird der Dateiname (z. B. Studium) angeführt, in der diese definiert ist.

Eine zweite Option zur Ermittlung der Klasse eines Objektes ist über die `isinstance`-Funktion gegeben. Als Parameter erwartet die Funktion ein Objekt und den Bezeichner einer Klasse. Als Rückgabe wird der Wahrheitswert `True` ermittelt, sofern das Objekt eine Instanz der bezeichneten Klasse oder einer Unterklasse von dieser ist, ansonsten `False`. Daher ergeben beide Abfragen in der Zeile 10 den Wert `True` – für die Instanz der Klasse `Bachelor` und die Oberklasse `Studierende`. Sofern für ein Objekt eine genauere Differenzierung der Klasse notwendig ist, kann dies über die `type`-Funktion erfolgen (s. Zeile 11). In dem Fall ist die erste Abfrage `False` und die zweite `True`.

Neben der Überprüfung der Klasseninstanz für ein Objekt ist es teilweise auch von Bedeutung, ob zwei Objekte gleich sind. Leider ist dieser Zusammenhang etwas komplizierter. Daher steht neben dem Vergleich mit dem booleschen `==`-Operator ein spezieller `is`-Operator zur Verfügung. Der `==`-Operator überprüft, ob die Werte der Instanzattribute von zwei Objekten gleich sind und der `is`-Operator, ob zwei Objekte wirklich identisch sind – sprich die gleiche Adresse im Speicher repräsentieren. Um dieses zu verstehen, schauen wir uns die Anweisungen in den Zeilen 5 und 13 im Quellcode etwas genauer an. In der Zeile 5 wird auf der rechten Seite vom Gleichheitszeichen ein Objekt vom Typ `Bachelor` instanziiert und dann mittels des Gleichheitszeichens der Variablen `stud_2` zugewiesen. Über die Variable `stud_2` wird ein spezifisches Objekt vom Typ `Bachelor` referenziert, welches bei der Instanziierung eine eindeutige Identität erhalten hat. In der Zeile 13 wird diese Referenz auch der Variablen `stud_3` zugewiesen. Daher zeigen (referenzieren) beide Variablen auf das gleiche Objekt mit der gleichen Identität. Entsprechend dieser Vorbemerkungen erhalten beide Anweisungen in den Zeilen 14 und 15 für die erste Abfrage den Wert `False` und für die zweite Abfrage den Wert `True`.

Quellcode 13.14: Testprogramm zur Typ-Überprüfung

```

1 from Studium import *
2
3 def main():
4     stud_1 = Studierende()
5     stud_2 = Bachelor()
6
7     print(type(stud_1))
8     print(type(stud_2))
9     print('-----')
10    print(isinstance(stud_2, Studierende), isinstance(stud_2, Bachelor))
11    print(type(stud_2) == Studierende, type(stud_2) == Bachelor)
12    print('-----')
13    stud_3 = stud_2
14    print(stud_1 is stud_2, stud_2 is stud_3)
15    print(stud_1 == stud_2, stud_2 == stud_3)
16
17 main()

```

Interessant wird diese Unterscheidung der beiden Operatoren auch dadurch, dass es möglich ist, den `==`-Operator individuell pro Klasse selber zu definieren. In dem Fall ist die spezielle `__eq__`-Methode, welche per Default definiert ist und alle Instanzattribute vergleicht, entsprechend anzupassen. Beispielsweise würde die Definition

```

def __eq__(self, other):
    return self.studname == other.studname

```

für die Klasse `Studierende` dafür sorgen, dass der `==`-Operator nur die Gleichheit der Namen der Studierenden überprüft. Abhängig vom Einsatz der Operatoren kann sich die Semantik des Programms grundlegend unterscheiden.

13.5.5 Iterierbare Objekte und Generatorfunktionen

Im Abschnitt 13.4.4 wurde bereits das Grundprinzip der Iteration, dem Durchlaufen einer Menge von Elementen, mithilfe der `for`-Schleife besprochen. Nachdem das Prinzip von Objekten über den Abschnitt 13.5 vermittelt wurde, kann im Folgenden das Prinzip der Iteration vertieft besprochen werden.

13.5.5.1 Iterierbare Objekte

Ein iterierbares Objekt von Python ist ein Objekt, das aufzählbare Elemente enthält. „Aufzählbar“ bedeutet, dass von dem Objekt ein sogenannter *Iterator* erhalten werden kann. Ein Iterator ist ein Objekt, mit dessen Hilfe die Elemente nacheinander aufgezählt werden können. In einer Liste (iterierbares Objekt) können die (aufzählbaren) Elemente beispielsweise nacheinander (mittels des Iterators) ausgegeben werden.

Quellcode 13.15: Deklaration einer Klasse von iterierbaren Objekten

```

1 class Intervall:
2     def __init__(self, start = 0, stop = 0, step = 1):
3         self.a = start
4         self.b = stop
5         self.s = step
6
7     def __iter__(self):
8         return self
9
10    def __next__(self):
11        if self.a < self.b:
12            x = self.a
13            self.a += self.s
14            return x
15        else:
16            raise StopIteration

```

Eine Klasse, die iterierbare Objekte spezifiziert, muss auch die vordefinierten Python-Methoden `__iter__()` und `__next__()` implementieren. Dabei muss die Methode `__iter__()` die Eigenschaft haben, dass ihr Aufruf für ein Objekt, welches die Klasse instanziiert, ein Iterator-Objekt zurückgibt. Die Methode `__next__()` muss bei dem Aufruf für das Iterator-Objekt das nächste Element des iterierbaren Objekts zurückgeben.

Ausgehend von diesen Erklärungen zeigt der Quellcode 13.15 ein Beispiel zur Deklaration einer Klasse, deren instanziierte Objekte iterierbar sind.

Die nachfolgenden Anweisungen zeigen exemplarisch die Verwendung der Klasse `Intervall`:

```

>>> eininterv = Intervall(start = 0, stop = 3)
>>> eininterviterator = iter(eininterv)
>>> next(eininterviterator)
0
>>> next(eininterviterator)
1
>>> next(eininterviterator)
2

```

Dabei ist die Anweisung „`eininterv = intervall(start = 0, stop = 3)`“ ein Beispiel für die Instanziierung eines iterierbaren Objekts. Die Rückgabe eines Iteratorobjekts wird nachfolgend mittels „`eininterviterator = iter(eininterv)`“ geleistet, wobei die eingebaute `iter`-Funktion von Python den Aufruf der `__iter__`-Methode von `eininterv` bewirkt. Das resultierende Iteratorobjekt wird der Variablen `eininterviterator` auf der linken Seite zugewiesen. Die Iteration durch ein iterierbares Objekt erfolgt durch sukzessive Anwendung der `__next__`-Methode des Iterators, der auch durch Aufruf der eingebauten `next`-Funktion ausführbar ist. So gibt der erste Aufruf `next(eininterviterator)` den Wert 0 zurück. Die nächsten beiden Aufrufe liefern 1 und 2. Ein weiterer Aufruf führt zur Beendigung der Iteration

mit Rückgabe einer `StopIteration`-Ausnahme. Diese wird mit dem Befehl `raise` erzwungen. Vordefinierte Ausnahmen (engl. *exceptions*) können in Python und anderen Programmiersprachen beim Eintreten den normalen Programmablauf unterbrechen und dadurch eine gezielte Behandlung auslösen.

Alternativ kann mit einer `for`-Anweisung über ein iterierbares Objekt iteriert werden. Die Anweisung

```
>>> for i in eininterv:
...     print(i)
```

gibt beispielsweise die Werte 0, 1 und 2 aus. Die Ausführung einer `for`-Anweisung beginnt mit dem Aufruf der `__iter__`-Methode des iterierbaren Objekts, das dem Schlüsselwort „in“ folgt, im Beispiel `eininterv`. Der zurückgegebene Iterator wird dann verwendet, um durch sukzessive Aufrufe seiner `__next__`-Methode über das iterierbare Objekt zu iterieren. Der durch einen `__next__`-Aufruf zurückgegebene Wert wird jeweils der vor „in“ stehenden Variablen, im Beispiel `i`, zugewiesen. Dann wird die Anweisungsfolge ausgeführt, im Beispiel `print(x)`. Sofern keine weiteren Elemente mehr vorliegen, was der Fall ist, wenn der Iterator eine `StopIteration`-Ausnahme auslöst, werden, sofern vorhanden, die Anweisungen im `else`-Teil ausgeführt und die Schleife wird beendet.

Für die eingebauten Python-Sammeltypen geschieht dies in der Regel analog. Die Wirkung der `for`-Anweisung

```
>>> studliste = ['Thorsten_Meier', 'Monika_Schmidt', 'Monika_Schneider']
>>> for stud in studliste:
...     print(stud)
```

kann in Iteratorform auch mit einer Ausnahmebehandlung über ein `try/except`-Konstrukt realisiert werden. In dem folgenden Beispiel

```
>>> studiterator = iter(studliste)
>>> while True:
...     try:
...         stud = next(studiterator)
...         print(stud)
...     except StopIteration:
...         break
```

führt die Ausnahmebehandlung zunächst die Anweisungen hinter „try“ aus. Sofern in deren Verlauf keine Ausnahme auftritt, wird der `except`-Teil übersprungen. Beim Eintreten einer Ausnahme wird die Ausführung der Anweisungen dort abgebrochen und mit dem `except`-Teil fortgefahren. Falls es sich bei der aufgetretenen Ausnahme um die angeführte Ausnahmeart handelt, im vorliegenden Beispiel die `StopIteration`, werden die darauffolgenden Anweisungen abgearbeitet, im Beispiel also die `break`-Anweisung, welche die `while`-Schleife beendet.

13.5.5.2 Generatorfunktionen

Eine weitere Möglichkeit der Erstellung von Iteratoren sind sogenannte **Generatorfunktionen**. Iteratoren, die auf diese Weise erzeugt werden, können auch objektbasiert erhalten werden. In vielen Fällen sind jedoch Generatorfunktionen der einfachere Weg.

Quellcode 13.16: Deklaration einer Generatorfunktion

```
1 def Intervall(start = 0, stop = 0, step = 1):
2     x = start
3     while x < stop:
4         yield x
5         x += step
6     return
```

Der Quellcode 13.16 zeigt ein Beispiel für die Deklaration einer Generatorfunktion. Generatorfunktionen werden wie normale Funktionen geschrieben, verwenden aber die `yield`-Anweisung zur Rückgabe von Daten. Diese besteht aus dem Schlüsselwort `yield`, welchem ein Ausdruck folgt, der zur Ermittlung der auszugebenden Werte dient. Im Beispiel repräsentiert die Variable `x` diesen Ausdruck.

Wird eine Generatorfunktion aufgerufen, gibt sie einen Iterator zurück, der als **Generator** oder Generatoriterator bezeichnet wird. Der Generator steuert dann die Ausführung der Generatorfunktion. Die Ausführung beginnt beim Aufruf der `next`-Methode des Generators. Beim ersten auftretenden `yield`-Ausdruck wird sie unterbrochen und der Wert des Ausdrucks der `yield`-Anweisung an den Aufrufer des Generators zurückgegeben. Bei der Unterbrechung wird der gesamte lokale Zustand der Ausführung beibehalten, d. h. insbesondere die Werte aller Variablen zu diesem Zeitpunkt. Durch den nächsten Aufruf der `next`-Methode wird die Ausführung mit der Anweisung fortgesetzt, die auf die `yield`-Anweisung folgt. Die Ausführung des Generators endet, wenn die Generatorfunktion vollständig abgearbeitet oder eine `return`-Anweisung der Generatorfunktion erreicht wurde. Dabei bewirkt die `return`-Anweisung die Aktivierung einer `StopIteration`-Ausnahme. Unter Verwendung des Quellcodes 13.16 gibt die nachfolgende `for`-Schleife

```
>>> for i in Intervall(start = 0, stop = 3):
...     print(i)
```

beispielsweise 0, 1 und 2 aus. Diese Ausgabe ergibt sich ebenfalls bei der direkten Anwendung des Generators:

```
>>> intervgenerator = Intervall(start = 0, stop = 3)
>>> while True:
...     try:
...         i = next(intervgenerator)
...         print(i)
...     except StopIteration:
...         break
```

Die Verwendung einer Generatorfunktion im Ausdruck nach „in“ einer `for`-Anweisung erfüllt die Anforderung der Fähigkeit der Iterierbarkeit durch Bereitstellung eines Iterators durch diesen Ausdruck. Die Klasse `Intervall` (s. Quellcode 13.15) und die Generatorfunktion `Intervall` (s. Quellcode 13.16) zeigen offensichtlich eine gewisse Ähnlichkeit zum eingebauten Python-Sammeltyp `range` (s. Abschnitt 13.4.1.3).

Eine weitere Möglichkeit der Erzeugung von insbesondere einfachen Generatoren sind **Generatorausdrücke**. Die Syntax von Python-Generatorausdrücken ähnelt jener von Listenabstraktionen (engl. *list comprehensions*, s. Abschnitt 13.4.1.1), wobei jedoch runde anstelle von eckigen Klammern verwendet werden. Generatorausdrücke sind für Situationen gedacht, in denen der Generator sofort von einer einschließenden Funktion verwendet wird. Sie sind kompakter, aber weniger vielseitig als Generatorfunktionen. Generatorausdrücke werden hier nicht weiter behandelt.

Aufgabe 13.12:

Legen Sie ein Wörterbuchobjekt `studadmin` an. Als Schlüssel eines Eintrags des Wörterbuchs soll die Matrikelnummer dienen und der Wert soll ein Studierendenobjekt gemäß der Klassendeklaration von Quellcode 13.8 sein. Das Wörterbuch soll mit den drei Studierendenobjekten initialisiert werden, deren Daten in Quellcode 13.9 verwendet werden.

Aufgabe 13.13:

Geben Sie, falls noch nicht geschehen, eine Lösung der Aufgabe 13.7 unter Anwendung der `for`-Anweisung mit dem implizit verfügbaren Iterator des Datentyps `set` an.

