

Python



Hochschule Kaiserslautern

Julian Bernhart, Manfred Brill, Eric Brunk, Mathias Fedder, Christopher Gross, Robin Guth, Rainer Haffner, Matthias Haselmaier, Fabian Kalweit, Lukas Kuhn, Kevin Konrad, Philipp Lauer, Miriam Lohmüller, Sebastian Morsch, Anatoli Schäfer, Denis Schlusche, Christoph Seibel, Marc Zintel

Vorwort

Das vom Stifterverband geförderte Projekt „Informatik studieren in der digitalen Gesellschaft (InfoStuDi)“ erprobt und evaluiert neue Lehr-, Lern- und Prüfungsformen in den Informatik-Studiengängen im Fachbereich Informatik und Mikrosystemtechnik der Hochschule Kaiserslautern.

Studiengänge an einer Hochschule für angewandte Wissenschaften bereiten die Studierenden auf die spätere Arbeitswelt vor. Diese Arbeitswelt wird von zeitlich und örtlich ungebundenem Tätigkeiten geprägt sein. Im Teilprojekt „Collaborative Writing“ wurde eine neue Form einer Lehrveranstaltung erprobt, die die Studierenden auf diese spätere Arbeitswelt vorbereiten soll. Ein Team aus Studierenden und Lehrenden verfasst ein Dokument zu einem Thema der Informatik. Dabei wird neben der Produktion von Texten auch Software entstehen. Die Produktion des vorliegenden Dokuments zum Thema Python wurde wie ein großes agiles Software-Projekt organisiert. Drei Sprints wurden durchgeführt, das Team organisierte sich selbst. Werkzeuge wie L^AT_EX, Git oder Jenkins wurden eingesetzt. Die Studierenden waren nicht nur Autoren, sondern auch Fachlektoren, Software-Entwickler und für die Qualität des Gesamtergebnisses mit verantwortlich.

Dieses Projekt wäre nicht zustande gekommen ohne die Studierenden, die sich auf dieses Abenteuer im Rahmen der Lehrveranstaltung „Aktuelle Themen aus Forschung und Praxis“ des Masterstudiengangs Informatik im Wintersemester 2018/19 eingelassen haben. An dieser Stelle ein herzliches „Danke schön!“ für das Vertrauen und den Mut, sich auf diese Form einer Lehrveranstaltung einzulassen. Miriam Lohmüller, als wissenschaftliche Mitarbeiterin im Projekt InfoStuDi tätig, brachte ihre Erfahrung aus dem Verlagswesen ein und hat die von den Studierenden verfassten Texte lektoriert. Fabian Kalweit, Mitarbeiter des Projektleiters im Fachbereich Informatik und Mikrosystemtechnik, hat das fachliche Lektorat unterstützt und insbesondere das Backend in GitHub organisiert und gestaltet.

Der vorliegende Text stellt den Stand im März 2019, nach Abschluss der Lehrveranstaltung, dar. Natürlich ist das Python-Tutorial nicht abgeschlossen. Das komplette Projekt steht in Form eines öffentlichen Git-Repositories ([?]) zur Verfügung und kann von interessierten Studierenden verwendet und vor allem weiterentwickelt werden. Alle Autoren hoffen, dass unsere Leser den Text für gut befinden.

Die Texte wurden nach bestem Wissen und Gewissen verfasst. Sollte der Text trotzdem Fehler enthalten, liegen diese in der alleinigen Verantwortung des Projektleiters!

Zweibrücken, im März 2019

Manfred Brill

Das Team



Das Projektteam nach dem letzten Sprint Meeting
im Januar 2019 von links nach rechts:

- | | |
|---------------|--|
| Hintere Reihe | Fabian Kalweit, Matthias Haselmaier, Marc Zintel, Robin Guth, Anatoli Schäfer, Denis Schlusche, Kevin Konrad, Miriam Lohmüller |
| Vordere Reihe | Mathias Fedder, Rainer Haffner, Lukas Kuhn, Sebastian Morsch, Julian Bernhart, Phillip Lauer, Christoph Seibel |
| Ganz vorne | Manfred Brill |

Die studentischen Autoren

Julian Bernhart



Eric Brunk



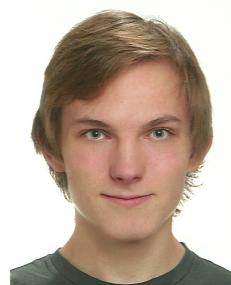
Mathias Fedder
Christopher Gross
Robin Guth



Rainer Haffner



Matthias Haselmaier
Kevin Konrad
Lukas Kuhn



Philipp Lauer
Sebastian Morsch
Anatoli Schäfer
Denis Schlusche



Christoph Seibel



Marc Zintel



Inhaltsverzeichnis

| | | |
|----------|--|----------|
| 1 | Grundlagen | 1 |
| 1.1 | Was ist Python? | 1 |
| 1.2 | Installation | 1 |
| 1.2.1 | Hinweis zur Installation unter Windows | 2 |
| 1.2.2 | Hinweis zur Installation unter Mac | 2 |
| 1.2.3 | Hinweis zur Installation unter Linux(Ubuntu) | 4 |
| 1.3 | Python Interpreter | 4 |
| 1.4 | Syntax | 5 |
| 1.4.1 | Allgemeine Strukturen | 6 |
| 1.4.2 | Leerzeichen und Einrückung | 6 |
| 1.4.3 | Kommentare | 6 |
| 1.4.4 | Typsicherheit | 7 |
| 1.5 | Beispiel „Hello World!“ | 7 |
| 1.6 | IDEs | 8 |
| 1.6.1 | Einige IDEs vorgestellt | 9 |
| 1.7 | Elementare Datentypen | 11 |
| 1.7.1 | Zahlenoperatoren | 11 |
| 1.7.2 | ENUMs | 13 |
| 1.7.3 | NUL oder NONE | 13 |
| 1.7.4 | Referenz, Identität und Kopie | 14 |
| 1.8 | Kontrollstrukturen | 14 |
| 1.8.1 | If-then-else | 15 |
| 1.8.2 | Schleifen | 15 |
| 1.8.3 | Ausdrücke und Operatoren | 17 |
| 1.9 | Fehler- und Ausnahmebehandlung | 17 |
| 1.9.1 | Mögliche Fehlerquellen | 18 |
| 1.9.2 | try, except, else und finally | 19 |
| 1.9.3 | raise | 22 |
| 1.9.4 | Selbst definierte Ausnahmen | 22 |
| 1.9.5 | Zusammenfassung | 23 |

| | |
|--|-----------|
| 1.10 Collections | 25 |
| 1.10.1 List | 25 |
| 1.10.2 Tuple | 31 |
| 1.10.3 Set | 33 |
| 1.10.4 Dictionary | 38 |
| 1.11 Klassen und Objekte | 45 |
| 1.11.1 Klassen und Objekte erstellen | 46 |
| 1.11.2 Die <code>__init__()</code> Methode | 47 |
| 1.11.3 Vererbung | 48 |
| 1.11.4 Die <code>__del__()</code> Methode | 49 |
| 1.11.5 Klassenattribute | 52 |
| 1.11.6 Statische Methoden | 53 |
| 1.12 Iteratoren | 55 |
| 1.12.1 Iterator und Iterable | 55 |
| 1.12.2 Benutzung von Iteratoren | 55 |
| 1.12.3 Erzeugung eigener Iteratoren | 57 |
| 1.13 Generatoren | 58 |
| 1.14 Zusammenfassung | 59 |
| 2 Ein- und Ausgabe | 63 |
| 2.1 Konsolenausgabe mit <code>print()</code> | 63 |
| 2.2 Formatierung von Strings | 65 |
| 2.2.1 Formatierung mit <code>format()</code> | 67 |
| 2.3 Konsoleneingabe mit <code>input()</code> | 70 |
| 2.4 Dateien lesen und schreiben | 72 |
| 2.4.1 Dateitypen | 72 |
| 2.4.2 Open-Methode | 72 |
| 2.4.3 Methoden | 74 |
| 2.4.4 With-Statement | 78 |
| 2.4.5 Attribute | 78 |
| 2.5 JSON | 79 |
| 2.6 Zusammenfassung | 82 |
| 3 Funktionen und Module | 85 |
| 3.1 Vorteile von Funktionen | 85 |
| 3.1.1 Aufteilen von komplexen Aufgaben | 86 |
| 3.1.2 Reduktion von Code-Duplikationen | 87 |
| 3.1.3 Bessere Lesbarkeit, Erweiterbarkeit, Veränderbarkeit | 87 |
| 3.2 Gültigkeitsbereich von Variablen und Funktionen | 88 |
| 3.2.1 Statements zu Gültigkeitsbereichen - global und nonlocal | 89 |

| | | |
|----------|---|------------|
| 3.3 | Input-Parameter | 91 |
| 3.3.1 | Arten von Input-Parametern | 92 |
| 3.4 | Lambda-Funktionen | 94 |
| 3.5 | Modularisierung | 94 |
| 3.5.1 | Erstellung eines lokalen Moduls | 95 |
| 3.5.2 | Module verwenden | 95 |
| 4 | Testen | 101 |
| 4.1 | Grundlegende Testmöglichkeiten | 101 |
| 4.1.1 | doctest | 101 |
| 4.1.2 | unittest | 105 |
| 5 | Benutzeroberflächen | 109 |
| 5.1 | Tkinter | 109 |
| 5.2 | Einbindung | 110 |
| 5.3 | Hello World mit grafischer Benutzeroberfläche | 110 |
| 5.4 | Die Layout-Manager | 112 |
| 5.4.1 | Pack | 113 |
| 5.4.2 | Grid | 117 |
| 5.4.3 | Place | 118 |
| 5.4.4 | Zusammenfassung | 118 |
| 5.5 | Widgets | 119 |
| 5.5.1 | Frame | 119 |
| 5.5.2 | Label | 120 |
| 5.5.3 | Button | 120 |
| 5.5.4 | Entry | 122 |
| 5.5.5 | Listbox | 123 |
| 5.5.6 | Colorchooser | 124 |
| 5.5.7 | Canvas | 125 |
| 6 | Python Bibliotheken | 129 |
| 6.1 | NumPy | 129 |
| 6.1.1 | Arrays | 129 |
| 6.1.2 | Konstanten und Funktionen | 131 |
| 6.1.3 | Erzeugen und Manipulieren von Arrays | 133 |
| 7 | Dokumentation | 141 |
| 7.1 | Epydoc | 141 |
| 7.2 | Docstrings | 142 |
| 7.3 | Epytext | 143 |

| | | |
|----------|---|------------|
| 7.4 | Zusammenfassung | 144 |
| 8 | Weiterführende Themen | 145 |
| 8.1 | Maschinelles Lernen in Python | 145 |
| 8.1.1 | Bibliotheken | 146 |
| 8.1.2 | Daten laden | 149 |
| 8.1.3 | Plots erzeugen | 153 |
| 8.1.4 | Beispiel: knn-Klassifikation | 158 |
| 8.1.5 | Beispiel: Naive Bayes | 161 |
| 8.1.6 | Beispiel: Lineare Regression | 163 |
| 8.2 | Datenbanken | 165 |
| 8.2.1 | Relationale Datenbanken | 165 |
| 8.2.2 | NoSQL Datenbanken | 168 |
| 8.3 | Nebenläufigkeit | 172 |
| 8.3.1 | Parallelität in Python | 172 |
| 8.3.2 | Threads | 173 |
| 8.3.3 | Prozesse | 192 |
| | Literaturverzeichnis | 203 |
| A | Lösungshinweise | 203 |
| A.1 | Lösungen zu Grundlagen | 203 |
| A.2 | Lösungen zu Datentypen und Kontrollstrukturen | 204 |
| A.3 | Lösungen zu Collections | 210 |
| A.4 | Lösungen zu Iteratoren | 215 |
| A.5 | Lösungen zu Testen | 216 |
| A.6 | Lösungen zu Klassen und Objekte | 217 |
| A.7 | Lösungen zu Funktionen und Module | 221 |
| A.8 | Lösungen zur Ein- und Ausgabe | 225 |
| A.9 | Lösungen zu Benutzeroberflächen | 226 |
| A.10 | Lösungen zu Bibliotheken | 232 |
| A.11 | Lösungen zu Maschinelles Lernen | 233 |
| A.12 | Lösung zu Datenbanken | 241 |
| A.13 | Lösungen zu Nebenläufigkeit | 243 |

Kapitel 1

Grundlagen

1.1 Was ist Python?

Die Programmiersprache Python wurde Anfang der 1990er Jahre von Guido van Rossum als Universalprogrammiersprache entwickelt. Der Name der Sprache beruht auf der Komikergruppe Monty Python. Hierzu lassen sich auch zahlreiche Anspielung in der Dokumentation von Python finden. Python wurde mit dem Ziel entworfen, eine einfache, gut verständliche und übersichtliche Programmiersprache zur Verfügung zu stellen. Dies soll nicht nur durch eine übersichtliche Standardbibliothek erreicht werden, sondern auch durch die Modulare Erweiterbarkeit. Im Folgenden wird die Programmiersprache Python in der Version 3 behandelt.

1.2 Installation

Python kann auf der Webseite <https://www.python.org> für eine Vielzahl von Betriebssystemen bezogen werden. Es stehen 32- und 64-Bit Versionen zur Verfügung. Nach dem Start des Installationsassistenten wird der Nutzer durch die Installation geführt. Der Ablauf der Installation unterscheidet sich je nach Betriebssystemen nur gering bis gar nicht. Nach erfolgreichem Abschluss stehen dem Anwender verschiedene Programme für die Arbeit mit Python zur Verfügung. Im vom Nutzer gewählten Installationsverzeichnis befinden sich nun folgende Programme:

IDLE Standard IDE¹ für Python

¹Integrated Development Environment

Python Standard Konsolen Interpreter

Pythonw Standard Interpreter ohne Konsolenausgabe

Diese Programme reichen aus, um Code mit Python zu entwickeln. Der Python Interpreter kann in der Konsole mit dem Befehl `python` aufgerufen werden. Die folgenden Abschnitte beschreiben Besonderheiten bei der Installation auf einzelnen Betriebssystemen.

1.2.1 Hinweis zur Installation unter Windows

Windows Nutzer müssen die Systemvariable für Python im Installationsassistenten hinzufügen lassen. Andernfalls kann Python nur im Installationsverzeichnis bzw. durch die Angabe des kompletten Pfads aufgerufen werden. In Abbildung 1.1 ist die notwendige Auswahl zu sehen.

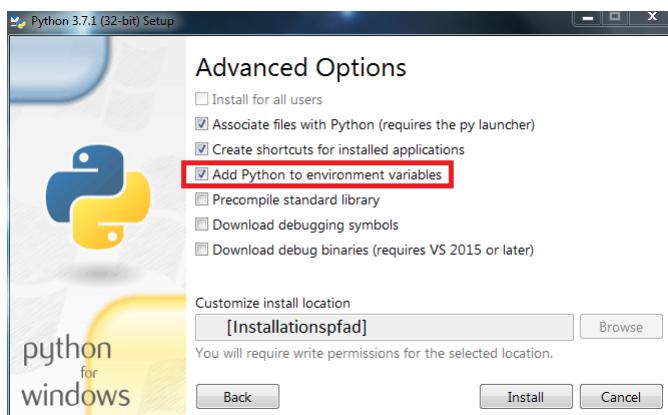


Abbildung 1.1: Start des Installationsassistenten

1.2.2 Hinweis zur Installation unter Mac

Im Folgenden wird die Installation unter macOS X gezeigt.

Nach Ende der Installation befindet sich der Python Ordner im Finder (Dateiexplorer).

Zuletzt wurde durch den Assistent unter `/Library/Frameworks/Python.framework` noch das Python Framework abgelegt. Ohne das wäre die Arbeit mit Python unter Mac nicht möglich.



Abbildung 1.2: Start des Installationsassistenten

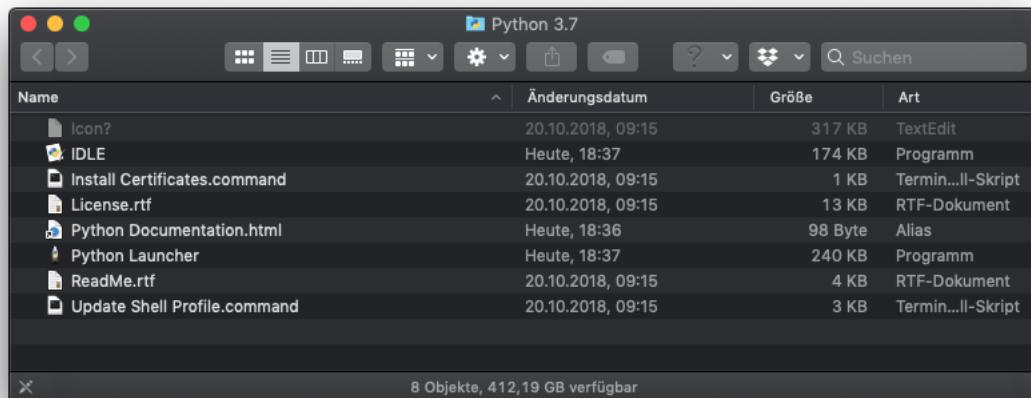


Abbildung 1.3: Fertige Installation

1.2.3 Hinweis zur Installation unter Linux(Ubuntu)

In diesem Kapitel wird die Installation für Ubuntu Version 18.04.1 LTS erläutert. Im Gegensatz zu den anderen Betriebssystemen, wird hier nur der Python Interpreter in der Version 3.6.6 mitgeliefert Standard Entwicklungsumgebung IDLE ist nicht vorinstalliert. Diese kann über das Paket IDLE nachträglich installiert werden. Sollte die Arbeit mit Python noch nicht möglich sein, kann durch den folgenden Befehl die Installation manuell angestoßen werden.

```
sudo apt-get install python3 python-doc
```

Diese Installation umfasst unter anderem auch die Dokumentation von Python. Nach Abschluss der Installationsroutine kann wie mit jedem anderen Betriebssystem mit Python gearbeitet werden.

1.3 Python Interpreter

Die einfachste Möglichkeit, Python Code auszuführen, ist die direkte Übergabe des Codes an den sogenannten Python Interpreter. Dabei handelt es sich um eine Konsolenanwendung, die Code ausführen und gegebenenfalls auftretende Ergebnisse anzeigen kann. Dabei kann ein Nutzer den Code entweder direkt in die Konsole eingeben oder aus einer Datei auslesen lassen. Wie bei anderen Programmiersprachen auch, stehen für Python verschiedene IDEs zur Verfügung, welche in Kapitel 1.6 behandelt werden. Für die ersten Versuche mit Python reicht der Interpreter jedoch völlig aus. Dieser wird standardmäßig mit Python installiert.

In Abbildung 1.4 ist der Interpreter zu sehen. Zusätzlich zur Version werden auch noch der Herausgeber von Python sowie die Uhrzeit angezeigt. Bereits jetzt kann erster Code ausgeführt werden. Im Folgenden werden zu einzelnen Bestandteilen von Python Beispiele beigelegt, die leicht im Interpreter ausführbar sind. Es wird dem Leser empfohlen, sie zum besseren Verständnis nachzuvollziehen, falls möglich durch selbständiges Ausprobieren.

```
Python 3.6.5 |Anaconda, Inc.| (default, Apr 26 2018, 08:42:37)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

Abbildung 1.4: Ansicht nach Start des Interpreters

Interaktiver Modus

Wird der Interpreter ohne Angabe einer Quellcodedatei gestartet, befindet dieser sich im interaktiven Modus. Der Nutzer kann hier direkt Anweisungen eingeben. Durch die Ausgabe der Zeichen >>> zeigt die Konsole an, dass sie eine Anweisung erwartet. In Python existieren auch mehrzeilige Anweisungen. Nach der Eingabe der ersten Zeile werden die Zeichen ... ausgegeben, was bedeutet, dass Folgeanweisungen erwartet werden.

Einlesen einer Datei

Auf Dauer ist die direkte Übergabe des Codes an den Interpreter sehr unpraktisch. Um Code erneut nutzen zu können und zu speichern, kann dieser in Dateien abgelegt werden. Dies kann mit einem simplen Texteditor erfolgen. Dateien, die Python Code enthalten, werden mit der Dateiendung ".py" gekennzeichnet. Sie können direkt mit dem der Konsole ausgeführt werden. Dazu muss Python der relative Pfad der Pythondatei (.py-Endung) übergeben werden.

```
python <Relativer Dateipfad der Pythondatei>
```

Da nur der relative Pfad angegeben werden muss, muss lediglich der Dateiname angegeben werden, falls die Konsole sich bereits im selben Verzeichnis wie die zu öffnende Datei befindet.

1.4 Syntax

Im Folgenden werden wichtige Grundkonzepte der Programmiersprache Python erläutert.

Syntax

```
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16)
Type "help", "copyright", "credits" or "license" for more information.
>>> 1+1
2
>>> 2*4
8
>>> 18/6
3.0
>>>
```

Abbildung 1.5: Einfache Ausdrücke im Python Interpreter

1.4.1 Allgemeine Strukturen

Anders als bei anderen Programmiersprachen wie beispielsweise Java oder C++, benötigt Python keine Klassenkonstrukte oder main-Methoden zur Ausführung. Das bedeutet, dass einzelne Anweisungen in Python korrekt ausgeführt werden können. Sollten für die Bearbeitung von komplexeren Problemen objektorientierte Ansätze benötigt werden, kann auch dies mit Python umgesetzt werden. Weitere Informationen zur Objektorientierung mit Python finden sich im weiteren Verlauf des Python Tutorials. Für den Moment reicht es für den Leser zu wissen, dass Anweisungen bereits zeilenweise ausgeführt werden können. Das kann sehr simpel getestet werden, indem der Python Interpreter als einfacher Taschenrechner genutzt wird. In Abbildung 1.5 wird dies gezeigt. Einfache Ausdrücke wie Summen, Subtraktionen, Multiplikationen und Divisionen können direkt eingegeben werden. Nach Betätigung der Eingabe-Taste liefert Python das Ergebnis des Ausdrucks.

1.4.2 Leerzeichen und Einrückung

Um in Python Blöcke auszuzeichnen, werden im Gegensatz zu Java oder C++ keine geschweiften Klammern genutzt. In Python werden Blöcke durch das Einrücken der Zeilen markiert. Hierfür sind entweder der Tabulator oder vier aufeinander folgende Leerzeichen vorgesehen.

1.4.3 Kommentare

Innerhalb der Programmiersprache Python wird zwischen Zeilen- und Blockkommentaren unterschieden. Zeilenweise Kommentare werden durch das Rautensymbol `\#` eingeleitet. Blockkommentare hingegen werden durch drei

aufeinander folgende Anführungszeichen „„“ jeweils zu Beginn und am Ende des Kommentars markiert. Hier wird jeweils ein Beispiel gezeigt:

```
# Zeilenweiser Kommentar
# muss vor jeder Zeile Wiederholt werden!

"""
Kommentarblock
Gilt so lange bis der Block durch
Wiederholen der Zeichen beendet wird!
"""


```

1.4.4 Typsicherheit

Anders als bei Java und C++ ist Python eine nur schwach typisierte Sprache. Somit ist bei der Initialisierung keine Typangabe erforderlich. Der Datentyp wird beim Initialisieren dynamisch ermittelt und automatisch zugewiesen. Um einer Variable trotzdem einen gewünschten Typ zuzuweisen, kann man sie einfach mit dem entsprechenden Typ initialisieren. Weitere Informationen zu Datentypen werden in Abschnitt 1.7 erläutert.

1.5 Beispiel „Hello World!“

Einfache Ausgaben können mit der print()-Anweisung gemacht werden. Innerhalb der Klammern muss ein String übergeben werden, sprich eine einfache Zeichenkette. Diese wird durch umschließende einfache oder doppelte Anführungszeichen markiert ("EinString"/'EinString'). Der Einfachheit halber, wird hier noch auf die genaue Erklärung der einzelnen Bestandteile der Anweisung verzichtet. Ein einfaches "Hello-World!"-Programm benötigt in Python nur eine Zeile:

```
print("Hallo World!")
```

Beispiel

Es handelt sich dabei um ein vollständiges Python-Programm, das in dieser Form ausgeführt werden kann. Wie bereits erläutert, werden anders als bei Java oder C++ keine Klassenkonstrukte oder main-Methoden benötigt.



Übungsaufgaben

Aufgabe 1.5.1

Installieren sie Python und lassen sie sich im Anschluss über `--version` die Version ihres Python Interpreters ausgeben.

Aufgabe 1.5.2

Schreiben sie ihr eigenes Programm, welches `Hello Python-World!` ausgibt. Das Programm soll über eine eigene Python Datei erstellt und ausgeführt werden.

Die Lösungen zu den Aufgaben finden Sie im Anhang A.1.

1.6 IDEs

Python Programmierung mit der IDLE (in Python integrierte Entwicklungsumgebung) oder Python Shell sind gute Möglichkeiten um den Einstieg in Python zu vereinfachen. Ein grundsätzliches Verständnis der Sprache und erste kleine Programme lassen sich so bewältigen. Sobald jedoch größere Programme oder Projekte anstehen kann es mit diesen Tools schnell frustrierend werden.

Eine passende IDE (Integrierte Entwicklungsumgebung) oder selbst ein einfacher Code-Editor kann einem das Leben deutlich vereinfachen. Im folgenden werden einige geeigneten IDEs für Python vorgestellt und für jede einige Vor- und Nachteile aufgezeigt.

Was ist eine IDE?

Integrierte Entwicklungsumgebungen vereinen wichtige Tools für das Erstellen von Software unter einer Oberfläche. Dazu zählen Editor mit Syntax-Hervorhebung, Compiler, Debugger, Interpreter und weitere Werkzeuge, die dem Entwickler die Arbeit erleichtern.

Durch den Austausch zwischen Werkzeugen innerhalb der IDE können Arbeitsgänge erleichtert und beschleunigt werden. Fehler im Quelltext beispielsweise können direkt in der entsprechenden Zeile markiert werden.

Anforderungen an eine Python Entwicklungsumgebung

Es gibt ein paar Grundanforderungen an eine geeignete oder gute Python Entwicklungsumgebung:

Debugging Unterstützung

Schrittweise durch den Code zu wandern, während dieser ausgeführt wird, ist eine weiter Grundaufgabe einer IDE.

Syntax Highlighting

Farbliche Markierungen erleichtern die Suche nach bestimmten Keywords. Die Lesbarkeit des Codes wird hierdurch verbessert.

Automatische Codeformatierung

Eine gute IDE erkennt das Zeilenende beispielsweise nach einem *While-Statement* und rückt die nächste Zeile automatisch ein.

Ausführen des Codes innerhalb der IDE

Wenn der Code außerhalb der IDE ausgeführt werden muss, ist es eher ein Text-Editor als eine IDE.

Interaktive Console

Live Ein- und Ausgabe einzelner Codezeilen.

Fehlererkennung

Syntaxfehler sollten automatisch markiert werden und Runtime-Fehler genannt werden.

1.6.1 Einige IDEs vorgestellt

Eclipse

Wer schon mit Java programmiert hat, ist wohl schon mal auf Eclipse gestoßen. Durch die Installation von PyDeth lässt sich Eclipse gut (und kostenlos) für die Python-Entwicklung erweitern und bietet dabei wichtige Features, wie zum Beispiel Code Completion, Python Debugging, eine interaktive Python Konsole oder das Einbinden von Django.

Vorteile: Wenn Eclipse bereits auf dem Rechner vorhanden ist, genügt der Download der PyDeth Erweiterung, welche nach einem Neustart von Eclipse sofort eingebunden ist.

Nachteile: Der Einstieg in die Python Programmierung für Eclipse-Neulinge kann in der sehr großen Entwicklungsumgebung Eclipse zu Schwierigkeiten führen.

Visual Studio

Die IDE aus dem Hause Microsoft bietet viele eigene Erweiterungen und Entwicklungs-Features an, welche dem Entwickler eine gute Individualisierungsmöglichkeit bieten. Die Visual Studio Python Tools ermöglichen es, alle üblichen Entwicklungsmöglichkeiten bei Programmierung mit Python zu nutzen. Visual Studio ist sowohl in der Community-Version umsonst, aber auch in einem Bezahlmodell verfügbar.

Visual Studio Community kann kostenlos über folgenden Link heruntergeladen werden: <https://visualstudio.microsoft.com/de/vs/community/>

Nachteile: Der Download ausschließlich für die Python-Programmierung ist recht groß (3-4GB). Es müssen sowohl das Programm an sich, als auch die Python-Erweiterung heruntergeladen werden. Visual Studio ist ausschließlich für Windows und MacOs verfügbar. Eine Linux-Variante wird allerdings von Visual Studio Code angeboten. Vergleichbar mit Eclipse ist auch Visual Studio nicht sonderlich einsteigerfreundlich.

Atom

Etwas schlichter geht es im Atom Editor zu. Das klare und strukturierte Interface ist auch für Einsteiger gut verständlich und dient daher als geringere Hürde, die ein Neuling überwinden muss, als die überladenen Interfaces von Eclipse oder Visual Studio. Python kann durch eine Erweiterung nachträglich installiert werden, welche während der Laufzeit hinzugefügt werden kann.

Vorteile: Einsteigerfreundliche Alternative mit geringem Download- und Installationsumfang, sowie Plattformunabhängigkeit.

Nachteile: *Build-* und *Debugging-Support* sind keine eingebauten Features, sondern nur als Community-Addon verfügbar.

PyCharm

PyCharm ist (wie es der Name vermuten lässt) eine IDE explizit für Python. Es ist neben der Hauptdatei keine Erweiterung nötig, ein neues Projekt kann sofort gestartet und mit dem Programmieren sofort begonnen werden. PyCharm ist Plattformunabhängig und sowohl in einer freien Open-Source-Version nutzbar, als auch in einer kostenpflichtigen Pro-Version erhältlich.

Vorteile: PyCharm bietet einen vielfältigen Support an und eine sehr aktive Community. Egal an welcher Stelle man ein Problem hat, es wird einem mit großer Wahrscheinlichkeit geholfen.

Nachteile: Die Ladezeiten sind vergleichbar lang und an manchen Stellen finden sich kleinere Usability-Schwachpunkte.

1.7 Elementare Datentypen

Ähnlich wie bei Java und C oder C++ gibt es auch in Python Variablen. Allerdings gibt es dabei immense Unterschiede zu den anderen Programmiersprachen, weshalb sich ein genauerer Blick auf die einzelnen Datentypen in jedem Fall lohnt. Bei vielen bekannten Sprachen wird einer Variablen ein bestimmter Datentyp zugeordnet (deklariert). Der Datentyp kann darauf folgend zur Laufzeit nicht wieder geändert werden, der Wert innerhalb des Datentyps allerdings schon. So lassen sich in eine Variable des Typ Integer beispielsweise keine String-Werte speichern. In Python hingegen ist dies ohne weiteres möglich. Hier wird gänzlich auf eine explizite Typdeklaration verzichtet. Zeigt eine Variable beispielsweise auf eine ganze Zahl, so wird diese als ein Objekt vom Typ Integer interpretiert. Allerdings ist es möglich, diese im nächsten Schritt einfach auf ein String-Objekt zeigen zu lassen. Dies ist in Python möglich, weil eine Variable ein Objekt lediglich referenziert und dadurch keinem Typ zugewiesen wird.

Betrachten wir nun die Datentypen etwas genauer.

1.7.1 Zahlenoperatoren

Da in Python auf Typdeklaration verzichtet wird, muss dies beim Anlegen von Variablen nicht berücksichtigt werden. Wird eine ganze Zahl (Integer) benötigt, kann diese, falls nötig, auch in eine Gleitkommazahl (float) umge-

wandelt werden, ohne viel am Code zu ändern. Python deklariert im Hintergrund selbst und spart so unnötige Komplexitäten und Fehlerquellen. (Beispiel 1.7.1)

```
# Zahlenoperatoren
i = 42
type(i)
// Ausgabe: <class 'int'>
i = 42.22
type(i)
// Ausgabe: <class 'float'>
```

Boolean

Boolean gibt an, ob ein Statement *true* oder *false* ist. Dadurch lassen sich Fallunterscheidungen oder Abfragen ermöglichen. (Beispiel in Listing 1.7.1)

```
# Boolean
i = True
i
// Ausgabe: True
```

String

Der String ist eine Zeichenkette, also eine Aneinanderreihung von verschiedenen Zeichen. Dazu zählen Wörter, aber auch beispielsweise Hexadezimal-Codes oder E-Mail Adressen.

Wie in den meisten objektorientierten Programmiersprachen lassen sich auch in Python die einzelnen Zeichen eines Strings abrufen, indem der dazugehörige Index abgefragt wird.

Wie in Listing 1.7.1 kann die Länge des gesamten Strings durch einfache Abfrage angezeigt werden.

```
# Strings
i = "Python"
print (i)
// Ausgabe: Python

print(i[0])
// Ausgabe: P

print(len(i))
```

```
// Ausgabe: 6
```

1.7.2 ENUMs

Enums dienen in den objektorientierten Programmiersprachen zur Aufzählung von Ausdrücken einer endlichen Menge. So werden zum Beispiel Jahreszeiten, Monate oder Farben oft als Enums umgesetzt (vgl. Listing 1.7.2).

```
# Enums
from enum import Enum
class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3
```

1.7.3 NULL oder NONE

Das Schlüsselwort *NULL* wird in vielen Programmiersprachen genutzt. Die Idee dahinter ist einer Variable ein neutrales Verhalten zu geben. Das Äquivalent zu *NULL* in Python ist *NONE*. Der Vorteil ist, dass *NONE* exakt der Aufgabe des Schlüsselworts entspricht. Ein Anwendungsfall für *NONE* wäre beispielsweise um zu Überprüfen, ob die Verbindung zu einer Datenbank aufgebaut werden konnte oder nicht (Siehe Beispiel 1.7.3).

```
# NULL oder NONE
database_connection = None

try:
    database = MyDatabase(host, user, password, database)
    database_connection = database.connect()
except DatabaseException:
    pass

if database_connection is None:
    // Solange die Variable "NONE", keine Verbindung aufgebaut
    print('The database could not connect')
else:
    print('The database could connect')
```

1.7.4 Referenz, Identität und Kopie

Wie bereits erwähnt wurde, wird in Python eine Variable keinem Typ zugewiesen. Zeigt eine Variable jedoch ständig auf ein neues Objekt, sind Verwechslungen innerhalb des Codes möglich. Um dies zu vermeiden bietet sich die Identitätsfunktion `id()` an. Diese hilft uns dabei, die verschiedenen Instanzen voneinander zu unterscheiden. Jede Instanz hat dabei unabhängig von ihrem Wert und ihrem Typ eine eindeutige Identität.

Dies ist in Python möglich, weil eine Variable ein Objekt lediglich referenziert und dadurch keinem Typ zugewiesen wird.



Übungsaufgaben

Aufgabe 1.7.1

- a) Erstellen Sie zwei Variablen `a` und `aa` und übergeben Sie ihnen die Werte 5 und 2.
- b) Erstellen Sie eine Variable `b`, die den Wert von der Summe von `a` und `aa` erhalten soll.
- c) Erstellen Sie eine String-Variablen `c` 'Das Ergebnis ist'.
- d) Ändern Sie den Wert von `a` zu 0.5
- e) Überprüfen sie den Typ von `a`, `b` und `c`.
- f) Was kommt heraus, wenn Sie sich `c+b+a` ausgeben lassen. Überlegen Sie vorher eine Lösung und überprüfen Sie diese anschließend im Programm.
- g) Lassen Sie in einem Programm für zwei Variablen `a` und `b` die Summe, die Differenz, das Produkt und den Quotient inklusive Rest ausgeben.
- h) Erstellen Sie eine Aufzählung für die Jahreszeiten.

1.8 Kontrollstrukturen

Die Kontrollstrukturen in Python haben einen formalen Unterschied zu Java oder C++, funktional allerdings sind sie identisch. In Python werden keine

geschweiften Klammern genutzt, um die Blöcke der einzelnen Abfragen abzugrenzen. Dazu genügt das Einrücken der Anweisung. Dies gilt sowohl für Bedingungen und Conditional Expressions, als auch für Schleifen. Im Folgenden schauen wir uns die einzelnen Strukturen im Detail und mit Beispielen an.

1.8.1 If-then-else

Die if-then-else-Struktur ermöglicht es, wie wir es bereits kennen, simple wenn-dann Abfragen zu tätigen.

Mehrere Abfragemöglichkeiten werden mit elif markiert. Vergleich hierzu Listing 1.8.1.

```
# If-then-else
if statement1:
    print("Fall 1")
elif statement2:
    print("Fall 2")
else:
    print("Fall 3")
```

Conditional Expressions

Die Conditional Expressions (engl. bedingte Ausdrücke) stellen eine kompaktere Schreibweise als if-then-else-Bedingungen dar. Ein Beispiel ist in Listing 1.8.1 zu finden.

```
# Conditional Expressions
# Klassisches If-Else
if wort == "start":
    x = "los"
else:
    x = "halt"

# If-Else als Conditional Expression
x = ("los" if wort == "start" else "halt")
```

1.8.2 Schleifen

Python hat sowohl bedingte, als auch Zähler-Schleifen, welche wir uns beide im Folgenden genauer ansehen werden (vgl. Listing ?? und ??). Schleifen

fen bestehen aus einer Anweisung und einem Kontrollblock, welcher solange durchlaufen wird, bis die Anweisung oder ein Abbruchkriterium erfüllt wurde. Schleifen, die niemals ein Abbruchkriterium erfüllen und so endlos durchlaufen werden, heißen Endlosschleifen. Diese führen dazu, dass der Interpreter irgendwann aufgibt und abbricht.

```
# While-Schleife
while Bedingung:
    Anweisungsblock
    if Bedingung:
        Anweisungsblock
        continue
    if Bedingung:
        Anweisungsblock
        break
    Anweisungsblock
```

```
# For-Schleife
for Variable in Objekt (von, bis, Variablenveränderung):
    Anweisungsblock
    if Bedingung:
        Anweisungsblock
        continue
    Anweisungsblock
    if Bedingung:
        Anweisungsblock
        break
    Anweisungsblock
```

Die while-Schleife erinnert stark an die Verwendung in anderen Programmiersprachen. Wird jedoch die for-Schleife betrachtet, fallen einige Unterschiede bei der Beschreibung der Anweisung auf. Die Variable wird einmalig am Anfang der Anweisung definiert. Anschließend wird die range, also die Grenzen von wo bis wo die Variable in der Schleife durchgegangen werden soll. Die beiden Grenzen werden mit einem Komma getrennt. Fügt man eine dritte Zahl dahinter ein, dient diese dazu die Variable bei jedem Durchgang der Schleife zu verändern. Lässt man diese Zahl weg, so wird die Variable bei jedem Schleifendurchlauf um eins erhöht. Schreibt man beispielsweise eine 3, wird die Variable jedes Mal, wenn die Schleife erneut durchgegangen wird um 3 erhöht.

Zur Veranschaulichung finden Sie im Folgenden zwei Beispiele zur Verwendung einer while- und einer for-Schleife. Beide Schleifen bilden die Summe der Zahlen von 1 bis 10.

```
# Beispiel als While-Schleife

n = 10
s = 0
i = 1

while i <= n:
    s = s + i
    i = i + 1

print ("Summe:", s)

# Ausgabe: "Summe: 55"
```

```
# Beispiel als For-Schleife

s = 0

for i in range (0,11):
    s = s + i

print ("Summe: ", s)

# Ausgabe: "Summe: 55"
```

1.8.3 Ausdrücke und Operatoren

Die meisten Operatoren für Zahlenwerte sind in Python ähnlich wie bei anderen Programmiersprachen. In Tabelle 1.1 wird eine Übersicht gegeben.

1.9 Fehler- und Ausnahmebehandlung

In diesem Kapitel beschäftigen wir uns mit der Fehler- und Ausnahmebehandlung in Python. Dabei sollte darauf geachtet werden, dass überall da, wo ein potenzieller Fehler auftreten kann, die Maßnahmen zur Fehler- und

Tabelle 1.1: Ausdrücke und Operatoren

| Operator | Bezeichnung | Beispiel |
|---|------------------------------|--|
| <code>+, -</code> | Addition, Subtraktion | <code>4 - 3</code> |
| <code>*, %</code> | Multiplikation, Rest | <code>24 % 5</code> Ergebnis: 4 |
| <code>/</code> | Division | <code>10 / 3</code> Ergebnis: <code>3.3333333333335</code> |
| <code>//</code> | Ganzzahldivision | <code>10 // 3</code> Ergebnis: 3 |
| <code>+x, -x</code> | Vorzeichen | <code>-5</code> |
| <code>**</code> | Exponentiation | <code>2 ** 4</code> Ergebnis: 16 |
| <code>or, and, not</code> | Boolsches Oder / Und / Nicht | <code>(a or b) and c</code> |
| <code>in</code> | Element von | <code>1 in [1,2,3]</code> |
| <code><, <=, >, >=, !=, ==</code> | Vergleichsoperatoren | <code>4 <= 5</code> |

Ausnahmebehandlung angewendet werden. Somit können wir entsprechende Syntaxfehler zur Laufzeit abfangen und geeignet behandeln, ohne das unser Programm vorzeitig durch einen Absturz beendet wird.

1.9.1 Mögliche Fehlerquellen

Die Möglichkeiten der Fehlerquellen sind vielseitig. Eine der Bekanntesten davon ist wohl ein Eingabefehler durch den Benutzer. Dieser soll beispielsweise eine Zahl mithilfe der Tastatur eingeben, damit diese durch das Programm weiterverarbeitet werden kann. Durch ein Vertippen des Anwenders wird ein Text in Form eines einzelnen Buchstabens anstatt einer Zahl übergeben. Dies führt zur Laufzeit zu einem Fehler, das Programm hat eine Zahl an Stelle eines Textes erwartet. Eine andere Fehlerquelle wäre die Division durch die Zahl Null. Ebenso könnte der Zugriffsversuch auf eine Datei zum Bearbeiten fehlschlagen, da diese zu dem aktuellen Zeitpunkt noch nicht existiert.

Aus diesen genannten Ursachen ist eine Fehler- und Ausnahmebehandlung

sinnvoll und sollte von einem Programmierer für einen möglichen Einsatz stets bedacht werden.

1.9.2 try, except, else und finally

Der `try`-Block wird mit dem Schlüsselwort `try` gefolgt von einem Doppelpunkt eingeleitet. Anschließend wird der auszuführende Code, der einen Fehler beinhalten könnte, darin angegeben. Mit dem Schlüsselwort `except` und dem Namen der zu behandelnden Fehlerklasse wie beispielsweise `ZeroDivisionError` kann ein entsprechender Fehler geeignet behandelt werden. Dabei sind auch mehrere `except`-Anweisungen mit unterschiedlichen Fehlerklassen möglich, um eine entsprechende Behandlung zu ermöglichen. Somit kann jeder Fehler nach seiner eigenen Art und Weise nach dem Auftreten konsequent und individuell behandelt werden. Auch die Erstellung von eigenen Fehlerklassen ist in Python möglich, dazu später mehr.

Einige wichtige und gängige Fehlerklassen sind hierbei:

"`ZeroDivisionError`": Tritt auf bei einer Division durch die Zahl null.

"`FileNotFoundException`": Tritt auf, wenn die zu öffnende Datei nicht gefunden werden kann.

"`IOError`": Tritt auf, wenn man auf eine Ressource zugreifen möchte, die momentan nicht verfügbar ist. Beispielsweise der Zugriff auf einen Drucker, der zu dem aktuellen Zeitpunkt sich in dem Status „offline“ befindet.

"`ValueError`": Tritt auf, wenn ein anderer Datentyp als der erwartete Verarbeiteten werden soll. Beispiel: Es wird eine Zahl erwartet, aber ein Text übergeben.

Darüber hinaus gibt es noch weitere wie z. B. `ImportError`, `KeyError`, `MemoryError`, `NameError`, `TypeError` und viele mehr, die hier im Kontext nicht weiter erläutert werden.

Im folgenden Listing wird ein Fehler provoziert, indem wir eine Division mit der Zahl Null herbeiführen. Hierbei wird die Fehlerbehandlung mit der Klasse `ZeroDivisionError` abgefangen und anschließend das Programm durch die Fehlerbehandlung ordnungsgemäß beendet. Dabei wird die nachfolgen-

de `else`-Klausel durch das Auftreten und Auffangen des Fehlers nicht ausgeführt.

```
# chapters/basics/src/ExceptionHandling.py
# Fehler- und Ausnahmebehandlung

# Auffangen des ZeroDivisionError
try:
    a, b = 5, 0
    res = a/b
except ZeroDivisionError as e:
    print("error message: ", e)
    print("Das Programm wird beendet.")
else:
    print("Ergebnis: " + str(res))

# Ausgabe:
# error message: division by zero
# Das Programm wird beendet.
```

else

Die `else`-Anweisung kann im Code optional mit angegeben werden und wird nur ausgeführt, falls es zu keiner Ausnahme in dem `try`-Block kommt. Somit wird nach der Ausführung des `try`-Blocks auch der in `else` stehende Code ausgeführt, wie es das nachfolgende Listing demonstriert:

```
# chapters/basics/src/ExceptionHandling.py
# Fehler- und Ausnahmebehandlung
# Auffangen des ZeroDivisionError
# else
try:
    a, b = 5, 2
    res = a / b
except ZeroDivisionError as e:
    print("error message: ", e)
    print("Das Programm wird beendet.")
else:
    print("Ergebnis: " + str(res))

# Ausgabe:
# Ergebnis: 2.5
```

finally

Mit der ebenfalls optionalen Angabe von `finally` lässt sich ein Codestück unter allen Umständen ausführen. Somit wird gewährleistet das dieser Teil

des Codes ungeachtet, ob eine Ausnahme eintrifft oder nicht garantiert durchlaufen wird. Dies kann vor allem bei dem Schließen einer Datei oder einer Datenbankverbindung sehr sinnvoll eingesetzt werden.

Bei dem folgenden Listing wird eine Datei zunächst angelegt und geöffnet. Anschließend wird das Ergebnis in die Datei geschrieben. Ungeachtet dessen ob im `try`-Block eine Fehlerbehandlung eintritt oder nicht, wird zum Schluss garantiert die `finally`-Anweisung durchlaufen. Damit ist die Schließung der Datei garantiert.

```
# chapters/basics/src/ExceptionHandling.py
# Fehler- und Ausnahmebehandlung
# finally Nutzung
try:
    file = open("datei.txt", "w")
    a, b = 5, 2
    res = a / b
    file.write("Ergebnis: " + str(res))
except ZeroDivisionError as e:
    print("error message: ", e)
    print("Das Programm wird beendet.")
except FileNotFoundError as e:
    print("error message: ", e)
    print("Das Programm wird beendet.")
else:
    print("Es ist kein Fehler aufgetreten.")
    print("Ergebnis: " + str(res))
finally:
    file.close()
    print("Die Datei wurde geschlossen.")

# Ausgabe auf der Konsole:
# Es ist kein Fehler aufgetreten.
# Ergebnis: 2.5
# Die Datei wurde geschlossen.
```

Anstatt die beiden Fehlerklassen `ZeroDivisionError` und `FileNotFoundError` mit demselben Code individuell zu behandeln, bietet es sich hier an, mehrere Ausnahmen innerhalb eines `except` zusammenzufassen. Tritt nun während der Abarbeitung im `try`-Block einer der genannten Fehler auf, wird die Fehlerbehandlung ausgeführt.

```
except (ZeroDivisionError, FileNotFoundError) as e:
```

```
print("error message: ", e)
print("Das Programm wird beendet.")
```

1.9.3 raise

Die `raise`-Anweisung ermöglicht uns das Generieren bzw. Auslösen einer Ausnahme. Dabei wird die Ausnahme angegeben, die ausgelöst werden soll.

```
# chapters/basics/src/ExceptionHandling.py
# Fehler- und Ausnahmebehandlung
# raise
raise NameError('ups')

# Ausgabe:
# Traceback (most recent call last):
#   File "C:/Users/t/Python-Tutorial/chapters/filehandling
#     /src/fehler_und_ausnahmebehandlung
#       /ExceptionHandling.py", line 2, in <module>
#         raise NameError('ups')
# NameError: ups
```

Somit ist es uns beispielsweise möglich eine `ZeroDivisionError` zu erzwingen, wie das folgende Beispiel demonstriert:

```
# chapters/basics/src/ExceptionHandling.py
# Fehler- und Ausnahmebehandlung
# Erzwingen der ZeroDivisionError
# raise
try:
    raise ZeroDivisionError
except ZeroDivisionError:
    print("Das Programm wird beendet.")

# Ausgabe:
# Das Programm wird beendet.
```

1.9.4 Selbst definierte Ausnahmen

Nachdem wir uns mit der `raise`-Anweisung beschäftigt haben, widmen wir uns den selbst definierten Ausnahmen. Diese werden in der Regel als eige-

nen Klassen definiert und müssen dabei von der Oberklasse `Exception` erben. Das nachfolgende Listing zeigt ein entsprechendes Szenario. Dabei wird zunächst eine eigene Klasse `MyException` definiert, die von `Exception` erbt. Durch die `raise`-Anweisung können wir unsere selbst definierte Ausnahme auslösen.

```
# chapters/basics/src/ExceptionHandling.py
# Fehler- und Ausnahmebehandlung
# Eigene Ausnahme Klasse verwenden
class MyException(Exception):
    def __init__(self):
        self.data = "Eigene Ausnahme aufgetreten!"

try:
    raise MyException
except MyException as e:
    print(e.data)

# Ausgabe:
# Eigene Ausnahme aufgetreten!
```

1.9.5 Zusammenfassung

In diesem Kapitel haben wir uns ausgiebig mit der Fehler- und Ausnahmebehandlung befasst. Zu Beginn haben wir uns einige mögliche Fehlerquellenszenarios, die durch fehlerhafte Benutzereingaben entstehen können näher angeschaut. Neben der Betrachtung der einzelnen `try`, `except`, `else`, `finally`, `raise` Anweisungen, haben wir uns am Ende auch mit der Implementierung von eigens definierten Ausnahmen beschäftigt. Dabei geht aus diesem Kapitel vor allem hervor, dass ein Programmierer immer darüber nachdenken sollte, ob es in seinem Code zu einem Fehler kommen kann. Denn dies gilt es mit einer entsprechenden Fehler- und Ausnahmebehandlung abzufangen.



Übungsaufgaben

Aufgabe 1.9.1

- a) Ein Programm soll eine Variable überprüfen. Hat die Variable den Wert '1' soll sie 'eins' ausgeben. Bei dem Wert '2' soll sie 'zwei' ausgeben, andernfalls soll sie 'weder eins, noch zwei' ausgeben.
- b) Von zwei int-Variablen soll die größere von beiden bestimmt werden. Lösen Sie dies mithilfe eines Programms.

Beispiel: Bei den Werten a=5 und b=2 soll ausgegeben werden: 'a ist größer als b'.

- c) Lösen Sie Aufgabe b) als Conditional Expression. (Der Fall, dass a und b gleich sind muss hier nicht beachtet werden)
- d) Ändern Sie Ihr Programm aus Aufgabe b) so ab, dass das Maximum von drei Zahlen statt von zwei bestimmt werden kann.
- e) Der Benutzer einer Applikation will herausfinden, für welche Fahrzeuge er bereits einen Führerschein machen darf. Dabei gilt: Ab 15 Jahren Mofa, ab 16 Jahren Motorroller und kleinere Motorräder, ab 17 Jahren begleitetes Fahren, ab 18 Jahren PKW (alleine), ab 21 Jahren LKW und ab 25 Jahren jegliche Motorräder.

Entwerfen Sie ein Programm, dass zu dem Alter des Benutzers alle möglichen Fahrzeuge ausgibt.

Aufgabe 1.9.2

- a) Bilden Sie die Summe aller Zahlen von 1 bis 100 als while-Schleife.
- b) Lösen Sie Aufgabe a) als for-Schleife.
- c) Schreiben Sie ein Programm, das alle Teiler einer ganzen Zahl ausgibt. (Beispiel: Bei der Zahl 10 sollen die Teiler 1,2,5 und 10 ausgegeben werden)
- d) Schreiben Sie ein Programm, das eine Zahl in ihre Primfaktoren zerlegt. (Beispiel: Die Zahl 112 wird in ihre Primfaktoren 2,2,2,2,7 zerlegt)

Die Lösungen zu den Aufgaben finden Sie im Anhang A.2.

Aufgabe 1.9.3

Schreiben Sie ein Programm das eine `ZeroDivisionError` Ausnahme auslöst. Gehen Sie dabei wie folgt vor:

Initialisieren Sie zwei List-Datenstrukturen, a mit den Werten 0-10 und b mit 11-21. Achten Sie hierbei unbedingt auf eine einheitliche Länge der beiden Strukturen. Legen Sie anschließend eine neue Textdatei an und öffnen Sie diese zum Beschreiben. Iterieren Sie über beide Datenstrukturen von hinten nach vorne und dividieren Sie dabei alle Elemente nach der Formel b/a . Anschließend schreiben Sie das Ergebnis in die Textdatei. Mögliche Fehler sollen dabei abgefangen werden.

Aufgabe 1.9.4

Schreiben Sie ein Programm, das eine Zahl über die Konsole entgegennimmt, anschließend quadriert und danach wieder auf der Konsole ausgibt. Bei einer falschen Eingabe vom Benutzer soll das Programm geeignet reagieren.

Hinweis: Verwenden Sie für die Fehler- und Ausnahmebehandlung `ValueError`.

Aufgabe 1.9.5

In dieser Aufgabe sollen Sie eine eigene Ausnahme Klasse definieren und anwenden. Diese soll dieselbe Funktionalität aus der vorherigen Aufgabe aufweisen. Verwenden Sie dazu die `raise`-Anweisung um die Ausnahme auszulösen.

1.10 Collections

In Python 3 existieren nativ die vier Datenstrukturen `List`, `Tuple`, `Set` und `Dictionary`, welche im Folgenden vorgestellt werden.

1.10.1 List

[List](#)

Die Datenstruktur `List` bietet einen geordneten und veränderbaren Behälter für Python-Objekte, der Duplikate von Elementen erlaubt. Da eine `List` immer sortiert ist, können einzelne Elemente aus der Datenstruktur über den entsprechenden Index ausgewählt und verändert werden. Python unterstützt intern keine Arrays, alternativ hierzu kann eine `List` verwendet werden.

Eine `List` kann wie folgt initialisiert werden:

```
# chapters/basics/src/list/ListInit.py
# Initialisierung einer List

liste = [1, 2, 3]

# oder
liste = list((1, 2, 3))
```

Dabei kann sie jegliche Art von Objekten beinhalten; der Datentyp spielt hierbei keine Rolle.

Beispiel:

```
# chapters/basics/src/list/ListDataType.py
# Objekte mit unterschiedlichen Datentypen in einer List

liste = [1, "hallo", 2.3, (5, 6), [22, 23, 24], 'a']
```

Im Gegensatz zu Java und C++ muss der Programmierer darauf achten und sicherstellen, dass die Datenstruktur mit Werten des entsprechenden Datentyps befüllt wird, um Fehler aufgrund unterschiedlicher Datentypen zu vermeiden.

Der Inhalt einer List kann über die `print()`-Methode ausgegeben werden. Im folgenden Beispiel werden verschiedene Elemente der List auf der Konsole ausgegeben. Wird die List als Parameter gewählt, wird der Inhalt ausgegeben.

```
# chapters/basics/src/list>ListPrint.py
# Ausgabe des Inhalts einer List auf der Konsole

liste = [1, 2, 3]
print(liste)
```

Wie zuvor erwähnt, ähnelt die Verwaltung einer List der eines Arrays aus Java oder C++. Durch die Verwendung eines Index können einzelne Elemente ausgewählt oder verändert werden.

```
# chapters/basics/src/list/ListIndex.py
# Beispiel fuer das Ueberschreiben und Ausgeben eines
# einzelnen Elements einer List

liste = [1, 2, 3]
```

```
print(liste[1])
liste[1] = 4
print(liste)
```

Python erlaubt die Nutzung von negativen Indizes. Mit diesen kann der Inhalt der List in umgekehrter Reihenfolge ausgegeben werden. Ein Index von -1 wird dem letzten Element der List zugeordnet, -2 dem vorletzten.

```
# chapters/basics/src/list/ListNegativeIndex.py
# Beispiel fuer die Verwendung eines negativen Index

liste = [1, 2, 3]
print(liste[-1])
print(liste[-2])
```

In Python existiert für die Datenstruktur List keine Methode, die mit `contains()` in Java oder der `find()` aus C++ vergleichbar ist. Stattdessen stehen die Membership Operatoren `in` oder `not in` zur Verfügung, die auf eine beliebige Sequenz oder die hier beschriebenen Collections angewendet, Auskunft darüber gibt, ob das spezifizierte Element darin enthalten ist.

```
# chapters/basics/src/list/ListInOperator.py
# Verwendung des in-Operators

liste = [1, 2, 3]

print(2 in liste)

if 2 in liste:
    print("Gefunden!")
else:
    print("Nicht gefunden!")
```

Der Python Interpreter stellt nativ einige Funktionen zur Verfügung. Eine davon ist die `len()`-Methode, die die Anzahl an Elementen in einem Objekt liefert.

```
# chapters/basics/src/list/ListLen
# Ausgaben der Anzahl der Elemente in einer List

liste = [1, 2, 3]

print(len(liste))
```

Das `del`-Statement erlaubt das Löschen einzelner Elemente oder der gesamten List.

```
# chapters/basics/src/list/ListDelete.py
# Beispiel zur Verwendung des del-Operators

liste = [1, 2, 3]
print(liste)

del liste[1]
print(liste)

del liste

print(liste) # ERROR, da die Liste nicht mehr existiert!
```

List-Methoden Methoden einer List

append(): Fügt am Ende der List ein Objekt hinzu.

```
# chapters/basics/src/list/ListAppend.py
# Verwendung der append-Methode

liste = [1, 2, 3]
print(liste)
liste.append(4)
print(liste)
```

clear(): Entfernt sämtliche Objekte aus der List.

```
# chapters/basics/src/list/ListClear.py
# Verwendung der clear-Methode

liste = [1, 2, 3]
liste.clear()
print(liste)
```

copy(): Liefert eine Kopie der List.

```
# chapters/basics/src/list/ListCopy.py
# Verwendung der copy-Methode

liste = [1, 2, 3]
```

```
duplicate = liste.copy()
print(duplicate)
```

count(): Liefert die Anzahl des spezifizierten Objekts in der List.

```
# chapters/basics/src/list/ListCount.py
# Verwendung der count-Methode

liste = [1, 2, 3, 2, 2]
print(liste.count(2))
```

extend(): Fügt der liste1 den Inhalt der liste2 am Ende hinzu.

```
# chapters/basics/src/list/ListExtend.py
# Verwendung der extend-Methode

liste1 = [1, 2, 3]
liste2 = [4, 5, 6]
liste1.extend(liste2)
print(liste1)
```

index(): Liefert den Index der Position, an der sich das erste spezifizierte Objekt in der List befindet.

```
# chapters/basics/src/list>ListIndexMethode.py
# Verwendung der index-Methode

liste = [1, 2, 3, 2, 2]
print(liste.index(2))
```

insert(): Fügt ein Objekt an der gewählten Position der List hinzu.

```
# chapters/basics/src/list/ListInsert.py
# Verwendung der insert-Methode

liste = ["1", "2", "3"]
liste.insert(1, "4")
print(liste)
```

pop(): Entfernt das Objekt, das sich an der durch den Index spezifizierten Position befindet.

```
# chapters/basics/src/list/ListPop.py
# Verwendung der pop-Methode
```

```
liste = [1, 2, 3]
liste.pop(1)
print(liste)
```

remove(): Entfernt das erste Objekt der List, das der Spezifikation entspricht.

```
# chapters/basics/src/list/ListRemove.py
# Verwendung der remove-Methode

liste = [1, 2, 3, 2]
liste.remove(2)
print(liste)
```

reverse(): Invertiert die Folge der Objekte in der List.

```
# chapters/basics/src/list/ListReverse.py
# Verwendung der reverse-Methode

liste = [1, 2, 3]
liste.reverse()
print(liste)
```

sort(): Sortiert die List.

```
# chapters/basics/src/list/ListSort.py
# Verwendung der sort-Methode

# Lexikographisches Sortieren
liste = ["b", "c", "a"]
print(liste)
liste.sort()
print(liste)

# Sortieren nach Zahlenwert
liste = [6, 1, 2, 3, 4, 5]
print(liste)
liste.sort()
print(liste)

# Umkehren der Sortierreihenfolge
liste.sort(reverse=True)
print(liste)
```

```
# Sortieren nach der Laenge einzelner Objekte
liste = ["aa", "aaa", "a"]

def sortFunc(x):
    return len(x)

liste.sort(key=sortFunc)
print(liste)

# Umkehren der Sortierreihenfolge
liste.sort(reverse=True, key=sortFunc)
print(liste)
```

1.10.2 Tuple

Tuple

Ein Tuple stellt einen geordneten und unveränderbaren Behälter für Python-Objekte dar. Dieser erlaubt, wie eine List, Duplikate und den Zugriff auf einzelne Elemente über einen Index. Tuple sind Datenstrukturen, die ausschließlich gelesen werden können.

Ein Tuple wird mit folgender Syntax erzeugt:

```
# chapters/basics/src/tuple/TupleInit.py
# Die Initialisierung eines Tuples

tupel = (1, 2, 3)

# oder
tupel = tuple((1, 2, 3))
```

Es ist möglich, leere Tuple zu erzeugen. Wie zuvor erwähnt, ist deren Inhalt unveränderlich.

Arbeiten mit einem Tuple

Der Inhalt eines Tuple kann, analog zur List, auf der Konsole ausgegeben werden. Das Zuweisen eines neuen Objekts mittels Index führt im Gegensatz zur List zu einem Fehler.

```
# chapters/basics/src/tuple/TupleIndex.py
# Zuweisung eines Objekts ueber den Index

tupel = (1, 2, 3)
tupel[0] = 4 # ERROR
```

Die Verwendung der Operatoren `in` und `not in` ist, wie die `len()`-Methode, analog zur List-Datenstruktur.

```
# chapters/basics/src/tuple/TupleInLen.py
# Verwendung des "in"-Operators

tupel = (1, 2, 3)

print(2 in tupel)
print(len(tupel))
```

Das `del`-Statement erlaubt das Löschen des Tuple. Aufgrund der Unveränderbarkeit der Datenstruktur können keine einzelnen Elemente entfernt werden.

```
# chapters/basics/src/tuple/TupleDelete.py
# Verwendung des del-Statements

tupel = (1, 2, 3)

del tupel
```

Methoden eines Tuple

count(): Liefert die Anzahl des gewählten Werts in einem Tuple.

```
# chapters/basics/src/tuple/TupleCount.py
# Verwendung der count-Methode

tupel = (1, 2, 4, 3, 2, 2)
print(tupel.count(2))
```

index(): Liefert die Position des ersten Werts, der mit dem spezifizierten Wert übereinstimmt.

```
# chapters/basics/src/tuple/TupleIndexMethode.py
# Verwendung der index-Methode

tupel = (1, 2, 3, 2)
print(tupel.index(2))
```

1.10.3 Set

Set

Ein Set ist durch das Hinzufügen oder Entfernen von Objekten veränderbar und erlaubt keine Duplikate. Das Initialisieren mit mehrfach identischen Werten führt nicht zu einem Fehler, jedoch werden die überzähligen Werte aus dem Set entfernt. Die enthaltenen Elemente sind unveränderlich. Zudem ist die Datenstruktur ungeordnet, weshalb nicht auf einzelne Objekte mittels Index zugegriffen werden kann.

Ein Datenbehälter vom Typ Set kann mit folgender Syntax erzeugt werden:

```
# chapters/basics/src/set/SetInit.py
# Die Initialisierung eines Sets

set1 = {1, 2, 3}

# oder
set1 = set((1, 2, 3))
```

Arbeiten mit Sets

Bei der Ausgabe eines Set auf der Konsole ist die Reihenfolge der Elemente nicht garantiert.

Die Syntax für die Ausgabe auf der Konsole ist analog zur List. Die Verwendung eines Index ist nicht erlaubt und führt zu einem Fehler.

```
# chapters/basics/src/set/SetPrint.py
# Ausgabe des Inhalts eines Set auf der Konsole

set1 = {1, 2, 3}
print(set1)
for x in set1:
```

```

    print(x)
print(set1[0])  # ERROR
set1[1] = 4   # ERROR

```

Set-Methoden Methoden eines Sets

add(): Fügt dem Set ein Objekt hinzu.

```

# chapters/basics/src/set/SetAdd.py
# Verwendung der add-Methode

set1 = {1, 2, 3}
print(set1)
set1.add(4)
print(set1)

```

clear(): Entfernt alle Elemente aus dem Set.

```

# chapters/basics/src/set/SetClear.py
# Verwendung der clear-Methode

set1 = {1, 2, 3}
print(set1)
set1.clear()
print(set1)

```

copy(): Liefert eine Kopie des Sets.

```

# chapters/basics/src/set/SetCopy.py
# Verwendung der copy-Methode

set1 = {1, 2, 3}
x = set1.copy()
print(x)

```

difference(): Liefert ein Set, das diejenigen Elemente enthält, die ausschließlich in `setX` vorkommen. Alle Element, die mit denen von `setY` übereinstimmen, werden aus dem ersten entfernt. Alternativ ist dies auch über den Operator – möglich.

```

# chapters/basics/src/set/SetDifference.py
# Verwendung der difference-Methode

```

```
set1 = {1, 2, 3}
set2 = {3, 8, 4}
x = set1.difference(set2)
print(x)

# oder
y = set1 - set2
print(y)
```

difference_update(): Entfernt diejenigen Elemente aus dem ersten Set, die mit denen aus dem zweiten übereinstimmen.

```
# chapters/basics/src/set/SetDifferenceUpdate.py
# Verwendung der difference_update-Methode

setX = {1, 2, 3}
setY = {3, 8, 4}
setX.difference_update(setY)
print(setX)
```

discard(): Entfernt das gewählte Element aus dem Set. Duplikate werden ebenfalls entfernt.

```
# chapters/basics/src/set/SetDiscard.py
# Verwendung der discard-Methode

set1 = {1, 2, 4, 3, 4}
set1.discard(4)
print(set1)
```

intersection(): Liefert ein Set mit der Schnittmenge zweier Sets. Alternativ ist dies auch mit der Angabe des &-Operators möglich.

```
# chapters/basics/src/set/SetIntersection.py
# Verwendung der intersection-Methode

setX = {1, 2, 3, 4, 5}
setY = {3, 4, 9, 5, 8, 7}
print(setX & setY)
print(setX.intersection(setY))
```

intersection_update(): Entfernt alle Elemente, die sich nicht in der Schnittmenge beider Sets befinden.

```
# chapters/basics/src/set/SetIntersectionUpdate.py
# Verwendung der intersection_update-Methode

setX = {1, 2, 3, 4, 5}
setY = {3, 4, 9, 5, 8, 7}
setX.intersection_update(setY)
print(setX)
```

isdisjoint(): Gibt Auskunft darüber, ob zwei Sets eine Schnittmenge besitzen. Liefert `True`, wenn kein Element des ersten Sets im zweiten enthalten ist.

```
# chapters/basics/src/set/SetIsDisJoint.py
# Verwendung der isdisjoint-Methode

setX = {1, 2, 3, 4, 5}
setY = {6, 7, 8, 9, 10}
print(setX.isdisjoint(setY)) # Liefert True

setX = {1, 2, 3, 4, 5}
setY = {3, 4, 9, 5, 8, 7}
print(setX.isdisjoint(setY)) # Liefert False
```

issubset(): Gibt an, ob das gewählte Set eine Teilmenge enthält, die exakt dem ersten Set entspricht. Alternativ kann das Zeichen `<` verwendet werden.

```
# chapters/basics/src/set/SetIsSubSet.py
# Verwendung der issubset-Methode

setX = {3, 4, 5}
setY = {3, 4, 9, 5, 8, 7}
print(setX.issubset(setY))
print(setX < setY)
```

pop(): Entfernt ein beliebiges Element aus dem Set. Sollte das Set leer sein, wird ein Fehler generiert.

```
# chapters/basics/src/set/SetPop.py
# Verwendung der pop-Methode

set1 = {1, 2, 3}
set1.pop()
```

```
print(set1)
```

remove(): Entfernt das gewählte Element aus dem Set. Sollte das gewählte Element nicht in dem Set enthalten sein, wird ein Fehler angezeigt.

```
# chapters/basics/src/set/SetRemove.py
# Verwendung der remove-Methode

set1 = {1, 2, 3}
print(set1)
set1.remove(3)
print(set1)
```

symmetric_difference(): Liefert ein Set, das die Vereinigung zweier Sets ohne deren Schnittmenge enthält.

```
# chapters/basics/src/set/SetSymDiff.py
# Verwendung der symmetric_difference-Methode

set1 = {1, 2, 3, 4, 5}
set2 = {6, 7, 8, 9, 10}
print(set1.symmetric_difference(set2))
```

symmetric_difference_update(): Vereinigt zwei Sets und entfernt deren Schnittmenge.

```
# chapters/basics/src/set/SetSymDiffUpdate.py
# Verwendung der symmetric_difference_update-Methode

set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8, 9, 10}
set1.symmetric_difference_update(set2)
print(set1)
```

union(): Liefert ein Set, das die Vereinigung zweier Sets darstellt. Duplikate werden entfernt.

```
# chapters/basics/src/set/SetUnion.py
# Verwendung der union-Methode

set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8, 9, 10}
print(set1.union(set2))
```

update(): Fügt einem Set die Items eines anderen hinzu. Duplikate werden entfernt.

```
# chapters/basics/src/set/SetUpdate.py
# Verwendung der update-Methode

set1 = {1, 2, 3, 4, 5}
set2 = {6, 7, 8, 9, 10}
set1.update(set2)
print(set1)
```

Frozenset

Im Gegensatz zu einem „normalen“ Set kann ein Frozenset nicht mehr verändert werden. Das Hinzufügen eines neuen Elements ist nicht erlaubt und führt zu einem Fehler.

```
# chapters/basics/src/set/FrozenSet.py
# Die Initialisierung eines Frozenset

set1 = frozenset([1, 2, 3, 4])
set1.add(5) # ERROR
```

1.10.4 Dictionary

Ein Dictionary ist eine ungeordnete, veränderbare Datenstruktur, die keine Duplikate erlaubt und Schlüssel-Objekt-Paare beinhaltet. Auch beim Dictionary ist die Reihenfolge der Ausgabe nicht garantiert, denn ein Dictionary besitzt keine Ordnung.

Ein Datenbehälter vom Typ Dictionary kann mit folgender Syntax erzeugt werden:

```
# chapters/basics/src/dictionary/DictInit.py
# Initialisierung eines Dictionary

dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
```

```
dictionary = dict(k1="v1", k2="v2", k3="v3")
```

Demnach befindet sich hinter dem Schlüssel `k1` das Objekt `v1` und analog dazu die weiteren Schlüssel-Objekt-Paare. Über den Schlüssel `k1` lässt sich auf das Objekt `v1` direkt zugreifen. Ebenso kann ein neues Objekt unter dem Schlüssel `k1` zugewiesen werden.

```
# chapters/basics/src/dictionary/DictPrint.py
# Ausgabe des, dem Schluessel k1 zugeordneten, Objekts

dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}

print(dictionary["k1"])
```

Eine alternative Möglichkeit, ein Dictionary zu erstellen, ist die Methode `zip()`. Mit deren Hilfe kann aus zwei separaten List-Behältern ein Dictionary generiert werden.

```
# chapters/basics/src/dictionary/DictZip.py
# Verwendung der zip-Methode

sprache = ["englisch", "deutsch", "franzoesisch"]
laender = ["England", "Deutschland", "Frankreich"]

laendersprache = dict(zip(laender, sprache))

print(laendersprache)
```

Methoden eines Dictionary

clear(): Entfernt alle Einträge aus dem Dictionary.

Dictionary-Methoden

```
# chapters/basics/src/dictionary/DictClear.py
# Verwendung der clear-Methode

dictionary = {
    "k1": "v1",
```

```

    "k2": "v2",
    "k3": "v3"
}
print(dictionary)
dictionary.clear()
print(dictionary)

```

copy(): Liefert eine Kopie des Dictionary.

```

# chapters/basics/src/dictionary/DictCopy.py
# Verwendung der copy-Methode

dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
x = dictionary.copy()
print(x)

```

fromkeys(): Liefert ein Dictionary mit den angegebenen Schlüsseln und Objekten.

```

# chapters/basics/src/dictionary/DictFromKeys.py
# Verwendung der fromkeys-Methode

x = ("k1", "k2", "k3")
y = "v"
dictionary = dict.fromkeys(x, y)
print(dictionary)

```

get(): Liefert das Objekt, das dem angegebenen Schlüssel zugeordnet ist.

```

# chapters/basics/src/dictionary/DictGet.py
# Verwendung der get-Methode

dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
print(dictionary.get("k1"))

```

items(): Liefert eine List mit einem Tuple für jedes Schlüssel-Objekt-Paar.

```
# chapters/basics/src/dictionary/DictItems.py
# Verwendung der items-Methode

dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
print(dictionary.items())
```

keys(): Liefert eine List von allen im Dictionary verwendeten Schlüsseln.

```
# chapters/basics/src/dictionary/DictKeys.py
# Verwendung der keys-Methode

dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
print(dictionary.keys())
```

pop(): Entfernt das Element mit dem entsprechenden Schlüssel aus dem Dictionary und liefert das Objekt zurück.

```
# chapters/basics/src/dictionary/DictPop.py
# Verwendung der pop-Methode

dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
print(dictionary.pop("k1"))
```

popitem(): Liefert das zuletzt hinzugefügte Schlüssel-Objekt-Paar als Tuple und entfernt es aus dem Dictionary.

```
# chapters/basics/src/dictionary/DictPopItem.py
# Verwendung der popitem-Methode

dictionary = {
    "k1": "v1",
```

```

    "k2": "v2",
    "k3": "v3"
}
print(dictionary.popitem())

```

setdefault(): Liefert das dem Schlüssel zugeordneten Objekt. Existiert dieser Schlüssel nicht, wird ein neues Schlüssel-Objekt-Paar mit dem angegebenen Schlüssel und Objekt angelegt.

```

# chapters/basics/src/dictionary/DictSetDefault.py
# Verwendung der setdefault-Methode

dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
x = dictionary.setdefault("k2", "v4")
print(x)
print(dictionary)
x = dictionary.setdefault("k4", "v5")
print(x)
print(dictionary)

```

update(): Fügt dem Dictionary ein Schlüssel-Objekt-Paar hinzu.

```

# chapters/basics/src/dictionary/DictUpdate.py
# Verwendung der update-Methode

dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
print(dictionary)
dictionary.update({"k5": "v5"})
print(dictionary)

```

values(): Liefert eine Liste mit allen im Dictionary enthaltenen Werten.

```

# chapters/basics/src/dictionary/DictValues.py
# Verwendung der values-Methode

```

```
dictionary = {  
    "k1": "v1",  
    "k2": "v2",  
    "k3": "v3"  
}  
print(dictionary.values())
```



Übungsaufgaben

Abschließend soll das in diesem Kapitel erlangte Wissen über Collections in Übungen angewandt und vertieft werden. Hierbei liegt der Fokus bei den Datentypen List, Tuple, Set und Dictionary.

Aufgabe 1.10.1

- a) Erzeugen Sie zunächst eine List und befüllen Sie diese mit den Elementen 1 bis 10. Um es möglichst effizient zu gestalten, verwenden Sie hierbei eine Schleife. Zur Validierung Ihres Ergebnisses können Sie die `print()`-Methode verwenden.
- b) Geben Sie die Länge der List auf der Konsole aus.
- c) Löschen Sie anschließend den Wert 4 aus der List.
- d) Fügen Sie zwischen den Werten 3 und 5 die Zahl 22 ein.
- e) Erweitern Sie die List am Ende um den Wert 1.
Was fällt Ihnen auf?
- f) Sortieren Sie anschließend die List.

Aufgabe 1.10.2

- a) Erzeugen Sie eine List und befüllen Sie diese mit den Elementen 1 bis 10 in beliebiger Reihenfolge.
- b) Geben Sie die Summe aller Elemente der List auf der Konsole aus.
- c) Geben Sie den größten Zahlwert der List auf der Konsole aus.
- d) Erzeugen Sie eine weitere List mit den Elementen 0 und 1. Schreiben Sie anschließend ein Programm, das die List mit den ersten zehn Fibonacci-Zahlen befüllt.

Tipp:

In der Fibonacci-Zahlenfolge ergibt die Summe zweier aufeinander folgenden, natürlichen Zahlen den nächsten Zahlwert.

Aufgabe 1.10.3

- a) Erzeugen Sie ein Tupel mit den Werten 2 und 3.
- b) Überprüfen Sie mit dem `in`-Operator, ob die Zahl 2 in dem Tuple vor kommt. Falls dies der Fall ist, soll das Wort `gefunden` auf der Konsole ausgegeben werden.
- c) Verändern Sie den Wert an der Position mit dem Index 0.
Was stellen Sie fest?

Aufgabe 1.10.4

- a) Erweitern Sie das Tuple aus der vorangegangenen Aufgabe um zwei weitere Elemente. Verwenden Sie hierfür die `list`- und `tuple`-Konstruktoren.

```
# Hinweis  
  
tuple1 = (1, 2, 3)  
tuple2 = tuple((1, 2, 3))  
tuple3 = tuple(tuple1)
```

- b) Verwenden Sie die zuvor genutzten Konstruktoren und die `str()`-Methode, um den Datentyp der Elemente des folgenden Tuples zu String abzuändern und geben Sie das Tuple zur Validierung auf der Konsole aus:
`stringTuple = ("hallo", 1, 2.1, False, "string")`

Aufgabe 1.10.5

- a) Initialisieren Sie ein Set mit den Werten 4, 5 und 6.
Anschließend ein weiteres Set mit dem Namen `Set2` und dem Inhalt 3, 4 und 5.
- b) Geben Sie nun die Schnittmenge der beiden Sets auf der Konsole aus
- c) Führen Sie anschließend eine Vereinigung beider Sets durch und speichern Sie diese in einer neuen Variable mit dem Namen `Set3`.
- d) Fügen Sie dem neuen Set die Werte 7 und 4 hinzu. Was stellen Sie fest?

- e) Erzeugen Sie nun ein Frozenset und versuchen Sie hierbei, ein weiteres Element hinzuzufügen.
Was stellen Sie fest?

Aufgabe 1.10.6

- a) Erzeugen Sie ein Dictionary. Der `key` soll der Anfangsbuchstabe Ihres Vornamens sein, während der `value` den komplett ausgeschriebenen Vornamen beinhaltet.

Anschließend soll Ihr Vorname als AKRONYM dienen. Erweitern Sie das Dictionary um weitere Einträge, entsprechend der Länge Ihres Vornamens.

Benutzen Sie hierzu das NATO-Alphabet.
Beispiel: ALEX = Alpha, Lima, Echo, Xray.

- b) Geben Sie anschließend nur die `values` aus.

- c) Schreiben Sie ein Programm, das die Vereinigung von zwei Dictionaries in einem weiteren speichert. Das resultierende Dictionary darf keine Duplikate beinhalten.

1.11 Klassen und Objekte

Python ist wie Java eine objektorientierte Programmiersprache. Das bedeutet, dass in Python fast alles aus Objekten und Klassen besteht. Klassen sind Vorlagen, aus denen Objekte generiert werden können. Dabei enthält die Klasse je nach Verwendungszweck Variablen und Methoden. Ein Beispiel hierfür wäre die Klasse `Kreis`. Jeder Kreis besitzt einen Radius, jedoch besitzt nicht jeder Kreis den selben.

Wie in Abschnitt 1.7 beschrieben wird der Datentyp einer Variable anhand der Daten, auf die sie referenziert, bestimmt. Auch bei elementaren Datentypen entspricht dies einer Zuordnung zu einer Klasse. Dies lässt sich durch folgendes Listing nachvollziehen:

```
#Die Variable i referenziert die 5
i = 5
#Da 5 einem Integer entspricht wird i dieser Klasse zugeordnet
print(type(i))
#Addition mittels Operand
print(i + 3)
```

```
#Variable j referenziert die 4
j = 4
#Variable j wird der Klasse Integer zugeordnet
print(type(j))
#Addition mittels Methode
print(j.__add__(7))
```

Ausgabe des Programmcodes:

```
<class 'int'>
8
<class 'int'>
11
```

Dadurch dass die Variablen i und j der Klasse Integer zugeordnet werden ist es möglich mittels Operand oder Methode eine Addition durchzuführen.

1.11.1 Klassen und Objekte erstellen

Um dies Anhand eines Python Programms zu verdeutlichen, wird eine neue Klasse erstellt.

```
# Erstellen einer Klasse
class Kreis:
    radius = 1
```

Mithilfe der erstellten Klasse, können nun verschiedene Objekte erstellt werden, welche die Variablen und Methoden der Klasse beinhalten.

```
class Kreis:
    radius = 1

kreis1 = Kreis()
print(kreis1.radius)
```

Ausgabe des Programmcodes:

```
1
```

Die Variablen der Objekte sind zunächst gleich mit denen der Klasse, aus der diese erstellt wurden. Allerdings sind die Variablen des erstellten Objekts unabhängig von denen der Klasse. Das bedeutet, dass diese auch unabhängig für jedes einzelne Objekt geändert werden können. Die im obigen

Beispiel verwendete Klasse ist in realen Anwendungen nicht verwendbar, da die Attribute des Objekts von Anfang an festgelegt wurden. Um einen dynamischen Ansatz nutzen zu können, sollte man wie im folgenden Beispiel vorgehen.

```
class Kreis:
    def __init__(self, radius):
        self.radius = radius

kreis1 = Kreis(3)
kreis2 = Kreis(5)
print(kreis1.radius)
print(kreis2.radius)
```

Ausgabe des Programmcodes:

```
3
5
```

1.11.2 Die `__init__()` Methode

Jede Klasse hat eine `__init__` Methode, die immer ausgeführt wird, wenn die Klasse initialisiert und ausgeführt wird. Diese Methode wird verwendet um den Variablen des Objektes einen Wert zu geben. Hierbei ist der `self` Parameter notwendig um die Klasseninstanz selbst, somit das generierte Objekt, zu referenzieren und auf dessen Variablen zugreifen zu können. Dieser Parameter ist also notwendig, muss aber nicht `self` genannt werden, sondern kann einen beliebigen Namen haben. Er wird als erster Parameter bei jeder Methode angegeben.

```
class Kreis:
    def __init__(self, radius):
        self.radius = radius

    def getRadius(self):
        print(self.radius)

kreis1 = Kreis(3)
kreis2 = Kreis(5)
print(kreis1.radius)
print(kreis2.radius)
```

Ausgabe des Programmcodes:

```
3  
5
```

Objekte enthalten die aus den Klassen übernommenen Methoden. Diese Methoden gehören jetzt zu dem Objekt. Um Parameter zu modifizieren oder zu löschen, können folgende Befehle verwendet werden.

```
■ objektname.parameter = neuer Wert (modifizieren)  
■ del objektname.parameter (löschen)  
  
class Kreis:  
    def __init__(self, radius):  
        self.radius = radius  
  
    def getRadius(self):  
        print(self.radius)  
  
kreis1 = Kreis(3)  
kreis2 = Kreis(5)  
kreis1.radius = 3  
kreis1.getRadius()  
kreis2.getRadius()  
del kreis2.radius
```

Ausgabe des Programmcodes:

```
3  
5
```

1.11.3 Vererbung

Ein weiterer wichtiger Aspekt in Python, ist die Vererbung und Ergänzung einer Klasse.

```
class Kreis:  
    def __init__(self, radius):  
        self.radius = radius  
  
    def getRadius(self):  
        return self.radius
```

```
class Farbe(Kreis):
    def __init__(self, radius, farbe):
        Kreis.__init__(self, radius)
        self.farbe = farbe;

    def getRadius(self):
        print(str(Kreis.getRadius(self)) + ", " + self.farbe)

kreis1 = Farbe(3, "rot")
kreis2 = Farbe(5, "gelb")
kreis1.getRadius()
kreis2.getRadius()
```

Ausgabe des Programmcodes:

```
3, rot
5, gelb
```

Im Beispiel 1.11.4 wurde die Klasse Kreis durch die Klasse Farbe erweitert. Dies wird durch das Hinzufügen eines weiteren Attributes (bspw. farbe) realisiert. In dieser Klasse wurde die Methode `getRadius()` abgeändert. Die Klasse, von der geerbt wird, kann entweder mit dem Klassennamen `Kreis` oder mit `super` referenziert werden. Trotz der Änderung der Methode `getRadius()` in der Klasse `Farbe`, behält die Klasse `Kreis` ihre Methoden.

Innerhalb der Klasse `Farbe` ist es somit möglich über `Kreis.getRadius()` oder mittels `super().getRadius()` auf die überschriebene Methode `getRadius()` der Klasse `Kreis` zuzugreifen. Da die Abgeleitete Klasse `Farbe` eigene Attribute besitzt werden diese über die `__init__` Methode initialisiert. Hierbei wird die `__init__` Methode der Klasse `Kreis` überschrieben und somit, beim erzeugen eines Objektes des Typs `Farbe`, nicht mehr automatisch aufgerufen. Damit das geerbte Attribut `radius` initialisiert und verfügbar ist, muss die `__init__` Methode der Klasse `Kreis` explizit aufgerufen werden.

1.11.4 Die `__del__()` Methode

Daraufhin bleibt zu erwähnen, dass auch komplette Objekte löschenbar sind. Dies ist mit dem Befehl `del Objektname` möglich. Im nachfolgenden Listing wird eine Fehlermeldung ausgegeben, da versucht wird, auf das gelöschte Objekt zuzugreifen.

```

class Kreis:
    def __init__(self, radius):
        self.radius = radius

    def getRadius(self):
        print(self.radius)

kreis1 = Kreis(3)
kreis2 = Kreis(5)
kreis1.radius = 6
kreis1.getRadius()
kreis2.getRadius()
del kreis2.radius
del kreis2
kreis2.getRadius()

```

Fehlermeldung beim Ausführen des Programmcodes:

```

Traceback (most recent call last):
  File "C:/users/Patrick/Desktop/python/objekte_undklassen.py", line 15, in <module>
    5
    kreis2.getRadius()
NameError: name 'kreis2' is not defined

```

Es ist möglich dass mehrere Referenzen auf ein Objekt zeigen. Erst sobald alle Referenzen auf ein Objekt mittels `del` gelöscht wurden wird das Objekt selbst gelöscht. Die Methode `__del__` wird, sofern sie implementiert ist, aufgerufen sobald ein Objekt gelöscht wird. Zu diesem Zeitpunkt sind die Attribute nicht mehr verfügbar, weshalb in der `__del__` Methode nicht mehr auf sie zugegriffen werden kann. Sehen wir uns hierzu folgendes Listing an:

```

class A():
    def __init__(self, name):
        self.name = name
        print(name + " wurde erzeugt!")

    def __del__(self):
        print("Objekt der Klasse A wurde gelöscht")

class B():
    def __init__(self, name):

```

```
self.name = name
print(name + " wurde erzeugt!")

def __del__(self):
    print("Objekt der Klasse " + name + " wurde gelöscht")

x = A("A")
y = B("B")

#Z erhält die gleiche Referenz wie x
#Beide zeigen somit auf ein Objekt der Klasse A
z = x

print("Löschen von x der Klasse A")
del x
print("Löschen von z der Klasse A")
del z
print("Löschen von y der Klasse B")
del y
```

Fehlermeldung beim Ausführen des Programmcodes:

```
A wurde erzeugt!
B wurde erzeugt!
Löschen von x der Klasse A
Löschen von z der Klasse A
Objekt wurde gelöscht
Löschen von y der Klasse B
Exception ignored in:
<bound method B.__del__ of <__main__.B object at 0x7f6813e3a898>>
...
NameError: name 'name' is not defined
```

Anfänglich wird ein Objekt der Klasse A erzeugt und durch x referenziert. Ebenso eines der Klasse B und durch y referenziert. Durch z = x gibt es Zwei Referenzen auf das Objekt der Klasse A. Erst sobald beide gelöscht sind wird der entsprechende Text der __del__ Methode ausgegeben. In der Letzten Zeile des Listings wird y gelöscht, was jedoch zu einem Fehler führt, da in der __del__ Methode auf ein Attribut zugegriffen wird welches nicht mehr verfügbar ist.

Bezüglich der Vererbung verhält es sich bei der `__del__` Methode wie bei der `__init__` Methode, sie überschreibt die Methode der Klasse von der geerbt wird.

1.11.5 Klassenattribute

Im Gegensatz zu den bis jetzt betrachteten Instanzattributen die jedes erzeugte Objekt für sich hat, greifen die Objekte einer Klasse gemeinsam auf die Klassenattribute zu. Hierdurch betrifft eine Änderung der Klassenattribute jedes Objekt der Klasse. Es ist auch möglich Klassenattribute zu verwenden ohne ein Objekt der Klasse erzeugt zu haben. Folgendes Listing soll zeigen wie ein Klassenattribut verwendet werden kann:

```
class A:
    counter = 0

    def __init__(self):
        A.counter += 1

    def __del__(self):
        A.counter -= 1

x = A();
print("Anzahl Instanzen von A:" + str(A.counter))
y = A();

#Klassenattribute sind auch über die Klasseninstanz erreichbar
print("Anzahl Instanzen von A:" + str(y.counter))

del y
print("Anzahl Instanzen von A:" + str(A.counter))
del x
print("Anzahl Instanzen von A:" + str(A.counter))
```

Ausgabe des Programmcodes:

```
Anzahl Instanzen von A:1
Anzahl Instanzen von A:2
Anzahl Instanzen von A:1
Anzahl Instanzen von A:0
```

Klassenattribute werden in der Klasse ohne `self` initialisiert. In diesem Beispiel ist dies `counter`. Dieser speichert die Anzahl der von der Klasse A erzeugten Objekte. Da jedes Objekt beim Erzeugen die `__init__` bzw. beim Löschen die `__del__` Methode aufruft, kann hierdurch das objektübergreifende Attribut `counter` entsprechend angepasst werden.

1.11.6 Statische Methoden

Statische Methoden werden nicht über Objekte einer Klasse aufgerufen, sondern direkt über die entsprechende Klasse. Sie können dazu benutzt werden Operationen bereitzustellen die unabhängig von den einzelnen Objekten sind. Sie werden als Funktion definiert und dann mittels `staticmethod` an eine Klasse gebunden. In Python gibt es keine Funktionsüberladung. Dies bedeutet sobald Funktionen mit dem gleichen Namen im selben Namensraum definiert werden würde dies zu Fehlern führen. Dies gilt auch für Methoden innerhalb einer Klasse. Nun ist es mittels statischen Methoden möglich Funktionen zur Objekterzeugung an eine Klasse zu binden. Nachfolgendes Listing soll dies verdeutlichen:

```
def standardType():
    return A("default")

class A:
    def __init__(self, type):
        self.type = type

    def getType(self):
        print(self.type)

#Bindet die Funktion als statische Methode an die Klasse
standardType = staticmethod(standardType)

x = A("extra")
y = A.standardType()

x.getType()
y.getType()
```

Ausgabe des Programmcodes:

```
extra
```

default

Mit Hilfe der Funktion `standardType()`, welche an die Klasse A gebunden ist, kann zur Erzeugung des Objektes ein Parameter Wert für die `__init__` Methode festgelegt werden.



Übungsaufgaben

Aufgabe 1.11.1

Schreiben Sie eine Klasse „Kreis“ die zwei Klassenvariablen (`radius`, `farbe`) enthält. Fügen Sie dieser Klasse Getter und Setter Methoden bei. Anschließend schreiben Sie eine Methode die beide Variablen auf der Konsole ausgibt.

Aufgabe 1.11.2

Die Klasse „Erweiterung“ soll nun von der Klasse Kreis aus Übung 1 abgeleitet und um das Attribut `xposition` ergänzt werden.

Aufgabe 1.11.3

Schreiben Sie eine Klasse „Fahrzeug“ die um die Klasse „Auto“ erweitert wird, welche somit von „Fahrzeug“ erbt. Überlegen Sie sich zwei Attribute die beim Anlegen eines Objektes vom Typ Auto mitgegeben und durch die `__init__` Methode der Klasse Fahrzeug initialisiert werden. Ebenso ein Attribut, welches in der `__init__` Methode der Klasse Auto initialisiert wird. Fügen Sie ebenso das Attribut „Kilometerstand“ hinzu, welches beim Erzeugen des Objektes initial auf 0 steht. Fügen Sie die Methode „fahren“, mit dem Parameter „kilometer“, hinzu. Wird die Methode aufgerufen soll entsprechend der Kilometerstand aktualisiert werden.

Aufgabe 1.11.4

Erweitern Sie Ihr Ergebnis aus Übung 3 um zwei weitere Klassen, mit jeweils einem Attribut, die von „Fahrzeug“ erben. Prüfen Sie ob die Attribute die Sie für die Klasse „Fahrzeug“ gewählt haben noch passend sind für die zwei neuen Klassen.

Aufgabe 1.11.5

Passen Sie Ihr Ergebnis aus Übung 4 so an, dass es möglich ist die Anzahl der erzeugten Objekte einer jeden Klasse zu ermitteln. Hierbei ist zu beachten, dass im Falle der Klasse „Fahrzeug“ auch die Objekte der erbenden Klassen dazu gehören sollen.

1.12 Iteratoren

In Kapitel 1.10 wurden Collections vorgestellt. Häufig ist es nötig, die gesamten Elemente einer Collection zu durchlaufen, beispielsweise um ein Element zu suchen oder jedes Element einer Liste auszugeben. Dies kann natürlich durch eine Schleife erreicht werden. Dazu muss sich jedoch der Programmierer um den Durchlauf der Collection kümmern. Eine weitere Möglichkeit ist die Nutzung eines Iterators, um den Durchlauf zu ermöglichen. Iteratoren sind dabei nicht auf Collections beschränkt und können vielfältig eingesetzt werden. Es ist auch möglich, für selbst erzeugte Klassen Iteratoren bereitzustellen. Deshalb sollen im Folgenden Iteratoren und deren Funktionsweise in Python erläutert werden.

1.12.1 Iterator und Iterable

Zunächst muss zwischen einem Iterator und einem Objekt mit der Eigenschaft `Iterable` unterschieden werden. Bei einem Iterator handelt es sich um ein Objekt, welches eine beliebige Anzahl an Werten enthält, die nacheinander durchlaufen werden können. Dazu muss die Methode `next()` implementiert sein. Sie liefert bei jedem Aufruf den nächsten verfügbaren Wert zurück. Sollten keine Werte mehr verfügbar sein, wird hingegen eine `StopIteration` Exception geworfen.

`Iterable` bedeutet, dass ein Objekt einen Iterator mithilfe der `iter()` Methode erzeugen kann. Welche Elemente ein Iterator zurückliefert ist implementierungsabhängig. Meist gibt es jedoch einen direkten Zusammenhang mit dem erzeugenden Objekt. Zu den wichtigsten Objekten, welche `Iterable` sind, gehören alle Collections und der Datentyp String. Ein durch eine Collection erzeugter Iterator liefert beispielsweise die Elemente, welche die Collection hält, zurück. Bei einem String hingegen werden als Elemente nacheinander die einzelnen Zeichen durch die `next()`-Methode geliefert.

1.12.2 Benutzung von Iteratoren

Die Nutzung eines Iterators ist simpel. Folgende Schritte müssen absolviert werden:

- 1) Erzeugen/Erhalten eines `Iterable`-Objektes

- 2) Erzeugen des zugehörigen Iterators durch Übergabe des `Iterable`-Objektes an `iter()`
- 3) Erhalt des nächsten Elementes durch Übergabe des Iterators an die `next()`-Methode
- 4) (Optional) Durchführung von Operationen, Ausgaben usw. mit dem erhaltenen Element
- 5) Wiederholung des letzten zwei Schrittes, bis Exception `StopIteration` geworfen wird (Fehlerbehandlung! siehe Kapitel 1.9)

Tipp:

Es ist nicht notwendig, die durch `next()` erhaltenen Elemente zu benutzen, jedoch macht dies wenig Sinn, da der Zugriff auf die Elemente der Hauptgrund zur Nutzung eines Iterators sind.

Im Folgenden wir ein einfaches Beispiel vorgestellt:

```
# Beispiel 01 Iteratoren

#Erstellung einer Liste
list =["triangle", "square", "circle", "rectangle"]
#Erzeugung des passenden Iterators zur Liste
iterator = iter(list)

#Erhalt und Ausgabe des naechsten Objekt des Iterators

print(next(iterator))
print(next(iterator))
print(next(iterator))
print(next(iterator))
```

Zunächst wird ein Objekt mit der Eigenschaft `Iterable` benötigt. Im Falle des Beispieles handelt es sich dabei um eine neu erzeugte Liste mit einigen Elementen. Mithilfe der Anweisung `iter()` wird der zugehörige Iterator `iterator` zu dem übergebenen `Iterable` Objekt erzeugt. Durch Aufruf der `next()`-Methode mit `iterator` als Parameter, wird das nächste Element des Iterators zurückgeliefert und auf der Konsole ausgegeben.

Verwendung einer for-Schleife

Bisher wurde die `next()`-Methode passend zur Anzahl der Objekte aufgerufen. Dies ist jedoch unpraktisch und wird in der Praxis bei einer unbekannten Anzahl an Objekten nicht einsetzbar sein. Die einfachste Möglichkeit, auf jedes Element eines `Iterable`-Objektes zuzugreifen ist die Verwendung einer `for`-Schleife. Dabei kann direkt das `Iterable`-Objekt an die Schleife übergeben werden. Dieser erzeugt selbstständig den Iterator und liefert pro Durchlauf ein weiteres Element zurück. Auch die Fehlerbehandlung, nachdem der Iterator keine Elemente mehr besitzt, wird übernommen. Dabei wird folgende Syntax verwendet: `for e in i` mit:

e Variable, welche bei jedem Durchlauf der Schleife mit dem Ergebnis des Aufrufes von `next()` belegt ist

i `Iterable`-Objekt, welches durchlaufen werden soll

for : Folgendes Beispiel zeigt die Anwendung einer `for`-Schleife:

```
# Beispiel 02 Iteratoren - for-Schleife

#Erstellung eines Strings
string = "ILoveTriangles"
#Erzeugung des passenden Iterators zum String
iterator = iter(string)

for element in iterator:
    print(element)
```

1.12.3 Erzeugung eigener Iteratoren

Es ist möglich, für selbst erzeugte Klassen eigene Iteratoren zu definieren. Dazu muss die `iter()` implementiert sein und ein Objekt zurück liefern, welches die `next()`-Methode implementiert. Ein Objekt kann auch sich selbst als Iterator zurückliefern. Wie `next()` implementiert ist, ist abhängig davon, welche Elemente und in welcher Reihenfolge der Iterator diese zurückliefern soll. Im Folgenden wird ein Beispiel für eine Klasse mit einfachem Iterator gezeigt, welche eine Collection hält und durch den Iterator deren Elemente rückwärts nacheinander zurückgibt. Es handelt sich dabei um ein Beispiel, welches aus [?] übernommen und leicht angepasst wurde:

```
# Beispiel 03 Iteratoren - Erstellung eines eigenen Iterators
```

```
class Reverse:
    # Klasse, welche Daten hält und selbst als Iterator fungiert
    # und Elemente in umgekehrter Reihenfolge zurück liefert

    def __init__(self, data):
        self.data = data
        self.index = len(data)

        # Objekt gibt sich selbst als Iterator zurück
    def __iter__(self):
        return self

    def __next__(self):
        # Abbruch falls Ende der Daten erreicht
        if self.index == 0:
            raise StopIteration
        # Vermindere Index bei jedem Aufruf von next()
        self.index = self.index - 1

        # Rückgabe des nächsten Elements
        return self.data[self.index]
```

1.13 Generatoren

Die vorher gezeigte Möglichkeit zur Erzeugung von Iteratoren sollte vermieden werden. Zur einfachen Realisierung von Iteratoren stehen die sogenannten Generatoren zur Verfügung. Diese werden wie normale Funktionen definiert und entsprechen von der Funktionalität her etwa der `next()`-Methode eines Iterators. Anstatt `return` wird jedoch `yield` verwendet, um Elemente zurückzuliefern. Generatoren haben die Eigenschaft, dass sie ein Gedächtnis haben, um alle Variablen und ausgeführten Anweisungen bis zum nächsten Aufruf des Generators zu speichern. Beim nächsten Aufruf von `next()` setzt der Generator seine Berechnung fort. Ein weiterer Vorteil ist der automatische Auslösen einer `StopIteration` Exception sobald keine weiteren Elemente mehr übrig sind. Es gilt zu beachten, dass sowohl Generatoren als auch selbst erzeugte Iteratoren gleich mächtig sind, also die selben Probleme gelöst werden können [?].

Hier wird noch einmal das Beispiel aus Kapitel 1.12.3 gezeigt, jedoch diesmal als Generator implementiert (übernommen aus [?]):

```
# Beispiel 04 Iteratoren - Erstellung eines eigenen Iterators

def reverse(data):
    #Hinweis: Die range(start, stop, step)-Anweisung
    #generiert eine Liste mit Nummern
    #von start bis stop mit einer Schrittweite von step
    for index in range(len(data)-1, -1, -1):
        yield data[index]

#Aufruf des Generators und Ausgabe der gelieferten Elemente
string = 'ILoveTriangles'
for char in reverse(string):
    print(char)
```



Übungsaufgaben

Abschließend soll das in diesem Abschnitt erlangte Wissen durch Übungen noch einmal wiederholt und vertieft werden.

Aufgabe 1.13.1

- Erzeugen sie eine Liste mit mindestens 3 Elementen und lassen sie diese mithilfe eines Iterators auf der Konsole ausgeben. Achtung: Es soll dabei möglich sein, dass die Liste eine beliebige Anzahl an Elementen hat.
- Erweiterung: Zählen sie die Anzahl der Elemente der Liste und lassen sie diese auch auf der Konsole ausgeben.

Aufgabe 1.13.2

Schreiben sie einen Generator, welcher ein `Iterable`-Objekt als Parameter annimmt und die Elemente des Objekts in umgekehrter Reihenfolge ausgibt.

1.14 Zusammenfassung

Abschließend soll das Kapitel Grundlagen und dessen Inhalt noch einmal für den Leser zusammengefasst werden. Zu Beginn des Kapitels hat der Le-

ser seine ersten Schritte mit der Sprache Python gemacht. Es wurde grundlegend vorgestellt, wobei es sich bei Python überhaupt handelt und wie die Programmiersprache installiert wird. Der Python Interpreter wurde vorgestellt, eine einfache Konsolenanwendung zur Ausführung von Python Code. Weiterhin wurden Kommentare und die spezielle Blockstruktur von Python eingeführt. Um das erste funktionierende Programm mit dem bis dahin erlangten Wissen erstellen, wurde ein klassisches Hello-World! Programm geschrieben. Hierbei wurde auch die Einfachheit von Python im Zusammenhang mit fehlenden Klassenkonstrukten wie bei Java oder C++ erläutert. Auch wenn sich einfache Konzepte und Funktionen noch leicht in einem Texteditor oder der Standard IDE umsetzen lassen, lohnt es sich, eine IDE mit zusätzlichem Funktionsumfang zu nutzen. Dazu wurden zuerst wünschenswerte Funktionen, deren Nutzen erläutert und anschließend passende IDE vorgestellt. Die Autoren empfehlen ausdrücklich die Verwendung einer IDE bei der Bearbeitung der Aufgaben in den folgenden Kapiteln. Dabei sollte sich der Leser eine IDE aussuchen, welche ihm zusagt und ihn optimal beim Entwickeln von Python Code unterstützt. Als Grundlage für die weitere Arbeit mit Python wurden die verschiedenen einfachen Datentypen betrachtet. Dabei wurden neben den Zahlenwerten, Boolean- und Stringvariablen auch Aufzählungen mit der Hilfe von ENUMs gezeigt. Dem Leser wurde ebenfalls das neutrale Element vorgestellt, welches an verschiedenen Stellen zum Einsatz gebracht werden kann. Im Zusammenhang mit einfachen Datentypen wurde auch eine Besonderheit von Python erläutert, die Möglichkeit der Referenzierung eines Objektes durch eine Variable, ohne dabei einen festen Typ zuzuordnen. Ein wichtiger Bestandteil einer Programmiersprache sind die zugehörigen Kontrollstrukturen. In diesem Zusammenhang wurde der Umgang mit Abfragen gezeigt. Zum einen wurde dabei betrachtet wie man if-else Anweisungen einsetzt und wie alternativ dazu Conditional Expressions verwendet werden können. Im weiteren wurde die Nutzung von Schleifen in der Python-Programmierung gezeigt. Als wichtiger Bestandteil des Themas Kontrollstrukturen wurde eine kurze Übersicht über die logischen Operatoren und deren Nutzen geliefert. Um Fehler abfangen und behandeln zu können wurden zuerst mögliche Fehlerquellen vorgestellt, Ausnahmen (Exceptions) zur Fehlerbehandlung eingeführt und zugehörige Methoden erläutert. Zu den gezeigten Methoden gehören `try`, `except`, `else`, `finally`, `raise`. Weiterhin wurde dem Leser die Möglichkeit eigene Exceptions zu erstellen und zu behandeln vorgestellt. Es gilt stets zu beachten, dass Fehler abgefangen und behandelt werden müssen, damit die Funktionalität eines Programmes gewährleistet ist. Im abschließenden Kapitel der Grundlagen wurden Collections zur Aufbewahrung von Daten und

deren Verwendung in Python beschrieben. Collections können je nach Datenstruktur unterschiedliche Eigenschaften aufweisen. Dem Leser wurden die verschiedenen Eigenschaften, zugehörige Funktionen und die Anwendungsmöglichkeiten vorgestellt. Zu den gezeigten Datentypen gehören List, Tuple, Set und Dictionary. Dem Leser wurde das Klassenkonzept von Python vorgestellt. Dazu gehört zum einen die Erstellung und Nutzung einer Klasse, die Benutzung der `init()`-Methode, als auch das Arbeiten mit Objekten. Auch das Prinzip der Vererbung zur Ergänzung bereits bestehender Klassen wurde gezeigt. Das Konzept der Iteratoren und der Unterschied zwischen einem Iterator und Iterable wurden eingeführt. Iteratoren lassen sich durch `Iterable`-Objekte wie beispielsweise Container oder Strings erzeugen und können deren Elemente schrittweise durchlaufen. Weiterhin wurde das Erstellen eigener Iteratoren bzw. die Verwendung sogenannter Generatoren vorgestellt. Durch das Wissen, welches der Leser im Kapitel Grundlagen erlangt hat, ist dieser nun bereit einfache Konzepte von Python anzuwenden und zu verstehen. Es wurden eine Vielzahl an Übungen bereitgestellt, um den Lernprozess zu unterstützen. Da es sich hierbei um essentielle Grundlagen von Python handelt, werden diese für alle weiteren Kapitel vorausgesetzt. Es wird deshalb empfohlen, erst mit dem Tutorial fortzufahren, wenn dieses Kapitel vollständig abgeschlossen ist.

Kapitel 2

Ein- und Ausgabe

In Python gibt es verschiedene Methoden, um Daten vom Benutzer entgegenzunehmen und Daten dem Benutzer zur Verfügung zustellen.

2.1 Konsolenausgabe mit `print()`

Die `print()`-Funktion hat vielerlei Nutzen und wird entsprechend oft verwendet. Daher ist es definitiv sinnvoll, sich mit ihr vertraut zu machen.

```
# die Print-Methode

print(value1, value2, ..., sep=" ",
      end="\n", file=sys.stdout, flush= False)
```

In Listing (2.1) sind die Parameter von `print()` zu sehen. Die auszugebenden Werte stehen an erster Stelle (`value1, value2, ...`), hierbei handelt es sich um eine beliebige Anzahl an Werten. Der Parameter `sep` bildet den Seperator zwischen den Werten und hat als Standardwert das Leerzeichen. In Listing 2.1 können wir sehen, dass wir durch Angabe eines Separators eine bessere Lesbarkeit erreichen können.

Seperator

```
# die Print-Methode mit Seperator

print(1,2,3)
# Ausgabe: 1 2 3

print(1,2,3, sep=" | ")
# Ausgabe: 1 | 2 | 3
```

Nach dem Seperator folgt der *end*-Parameter, dieser fügt standardgemäß einen Zeilenumbruch (\n) an das Ende der Ausgabe.

```
# die Print-Methode mit End-Angabe

print("Satz mit Zeilenumbruch")
print("Nächster Satz")
# Ausgabe:
# Satz mit Zeilenumbruch
# Nächster Satz

print("Satz mit Punkt und Leerzeichen." , end=". ")
print("Nächster Satz")
# Ausgabe:
# Satz mit Punkt und Leerzeichen. Nächster Satz
```

Der *file*-Parameter bestimmt den Datenstrom (*Stream*) für den Output. In `sys.stdout` steht für die Konsole. Möchten wir den Output bspw. in eine Textdatei schreiben, dann können wir diese als Ziel des Datenstroms festlegen.

```
# die Print-Methode mit Ausgabe in Textdatei

# open() -> festlegen einer Textdatei als Stream
# "w" steht für 'write' und gibt an,
# dass wir etwas in die Datei schreiben möchten

txtFile = open(beispielText.txt, "w")
print("Hello, World.", file="txtFile")
txtFile.close()
# close() schließt den Stream
```



- 2.1 Was sind die Parameter der `print()`-Funktion?
- 2.2 Welche Funktionalität bietet uns die `print()`-Funktion?
- 2.3 Was ist der Unterschied zwischen dem Seperator und dem *end*-Parameter?

2.2 Formatierung von Strings

Es wäre sicherlich hilfreich, wenn wir die String-Ausgaben nach belieben formatieren könnten. Bisher haben wir den Separator von `print()` kennengelernt - dieser ist von seiner Funktionsweise jedoch stark beschränkt. Python bietet uns hierfür die `format()`-Methode an, vorher betrachten wir aber die Modulo-Arithmetik und machen uns mit den Formatierungszeichen vertraut.

Mittels Modulo-Arithmetik leiten wir ein Formatierungszeichen ein. Dieses gilt als Platzhalter für einen Wert.

Modulo-Arithmetik

```
# Modulo-Arithmetik

print("Körper: %s , Fläche: %f" %
      ('Dreieck', 42.6))

# Ausgabe:
# Körper: Dreieck , Fläche: 42.6
```

Bei "Körper" setzen wir den ersten Platzhalter mit "%s". Die Reihenfolge der Platzhalter setzt fest, welcher Wert anschließend eingebunden wird. Der erste Platzhalter trägt also den ersten Wert, der nach dem Ausgabetext folgt, der zweite den zweiten und so weiter.

Achtung:

Die einzusetzenden Werte werden nach dem String mittels Modulo als Tupel festgelegt!

Das Formatierungszeichen nach dem Modulo bestimmt den Datentyp des Wertes. Bei "s" handelt es sich um einen String, bei "f" um ein *float*. In Tabelle 2.1 sind die möglichen Formatierungszeichen aufgelistet.



- 2.1 Wir errichten einen Platzhalter `%d` und geben den einzusetzenden Wert '2.3' an. Ausgegeben wird der Wert '2', wieso?
- 2.2 Wie bestimmen wir, welcher Wert zu welchem Platzhalter gehört?

Was ist jedoch, falls die Ausgabe eine bestimmte Länge haben soll? Mit Hilfe der Syntax der Modulo-Arithmetik können wir dieses Problem lösen.

```
# Modulo-Arithmetik Syntax
```

Tabelle 2.1: Formatierungszeichen und ihre Bedeutung ([?])

| Platzhaltersymbol | Bedeutung |
|-------------------|--|
| d | Vorzeichenbehaftete Ganzzahl (Integer, dezimal) |
| i | Vorzeichenbehaftete Ganzzahl (Integer, dezimal) |
| o | vorzeichenlose Ganzzahlen (oktal) |
| u | vorzeichenlose Ganzzahlen (dezimal) |
| x | vorzeichenlose Ganzzahlen (hexadezimal) |
| X | vorzeichenlose Ganzzahlen (hexadezimal) |
| e | Fließkommazahl im Exponentialformat |
| E | Fließkommazahl im Exponentialformat |
| f | wie e |
| F | wie E |
| g | Wie e, wenn der Exponent größer ist als -4 oder kleiner als die Präzision. Ansonsten wie f |
| G | wie E und analog zu g |
| c | ein Zeichen |
| s | Eine Zeichenkette (String), beliebige Python-Objekte werden in String mittel der Methode <code>str()</code> gewandelt. |
| r | wie s |
| % | Es findet keine Konvertierung des Arguments statt, es wird ein %-Zeichen ausgegeben |

```
%[Flag] [Minimum der Gesamtlänge].[Präzision][Typ]
```

Das Minimum der Gesamtlänge bringt große Vorteile mit sich, wenn wir z.B. einen linksbündigen Text ausgeben wollen. Alle Ausgaben, die kürzer als das vorgegebene Minimum sind, werden mit Leerzeichen aufgefüllt.

Achtung:

Es handelt sich hierbei um das Minimum der Gesamtlänge. Alle Ausgaben die größer sind, werden nicht beschränkt und in voller Länge ausgegeben!

Mittels Punkt können wir folgend die Präzision einstellen, was bei einem `float`-Datentyp die Nachkommastellen bestimmt. Alle Zahlen werden zu der angegebenen Nachkommastelle aufgerundet!

```
# Modulo-Arithmetik Präzision
```

```
print("Eine Zahl %f" % (1.234))
print("Eine gerundete Zahl %.2f")

# Ausgabe:
# Eine Zahl 1.234
# Eine gerundete Zahl 1.23
```

2.2.1 Formatierung mit format()

Python bietet uns für die Formatierung von String-Elementen die Methode `format()`.

```
# Syntax der format()-Methode

string.format(par0, par1, ..., key0=val0, key1=val1, ...)
```

`format()` ersetzt markierte Stellen im gegebenen String durch angegebene Werte (Parameter in `format()`) und liefert diesen zurück. Die Stellen werden durch geschweifte Klammern markiert und mittels Modulo-Arithmetik präzisiert. In der geschweiften Klammer geben wir als erstes den Index (oder das Schlüsselwort) des Parameters an.

```
# format() mit gegebenem String

str = "Hallo, {0:s} und {1:s}"
print(str.format("Rainer", "Denis"))
# Ausgabe:
# Hallo, Rainer und Denis

print(str)
# Ausgabe:
# Hallo, {0:s} und {1:s}

# format() verändert den String nicht,
# sondern liefert den veränderten Wert zurück

str = str.format("Rainer", "Denis")
print(str)
# Ausgabe:
# Hallo, Rainer und Denis

# der Wert von 'str' wurde durch den Rückgabewert
```

```
# von format() überschrieben!

str = "Hallo, {1:s} und {0:s}"
str.format("Rainer", "Denis")
print(str)
# Ausgabe:
# Hallo, Denis und Rainer

# in 'str' wurde der angegebene Index vertauscht

str = "Hallo, {r:s} und {d:s}"
print(str.format(r = "Rainer", d ="Denis"))
# Ausgabe:
# Hallo, Rainer und Denis

# Angabe von Schlüsselwort-Parametern
```

Achtung:

Möchte man geschweifte Klammern ausgeben, dann werden diese doppelt geschrieben ("{{" und "}}")

Die `format()`-Methode bietet uns außerdem Ausrichtungsoptionen, was zu besserer Lesbarkeit beitragen kann. Somit können wir bspw. Werte links- oder rechtsbündig ausgeben. Hierfür gibt man die Formatierungsanweisung wie in Tabelle 2.2 an und den Wert des Abstandes bzw. der Größe des Platzhalters. Ist das Wort zu kurz, wird der restliche Platz mit Leerzeichen aufgefüllt.

Tabelle 2.2: Links- und rechtsbündiger Text

| Formatierungsanweisung | Bedeutung |
|------------------------|----------------------------------|
| < | Text wird linksbündig ausgelegt |
| > | Text wird rechtsbündig ausgelegt |

```
# Ausrichtung mit Formatierungsanweisung

# linksbündig
str = "{0:<10s} {1:d}"
print(str.format("Viereck", 4))
print(str.format("Fünfeck", 5))
print(str.format("Sechseck", 6))
```

```
# Ausgabe:  
# Viereck      4  
# Fünfeck      5  
# Sechseck     6  
  
# rechtsbündig  
str = "{0:>10s} {1:d}"  
print(str.format("Viereck", 4))  
print(str.format("Fünfeck", 5))  
print(str.format("Sechseck", 6))  
  
# Ausgabe:  
#      Viereck 4  
#      Fünfeck 5  
#      Sechseck 6
```

Python bietet uns für Dictionaries einen einfachen Weg, diese mittels `format()` und der Nutzung von Schlüsselwort-Parametern, auszugeben.

```
# Formatierung eines Dictionarys  
  
dictMath = {"Dreieck" : "3",  
            "Viereck" : "4",  
            "Fünfeck" : "5",  
  
str = "{body}: {corners}"  
  
for geoBody in dictMath:  
    print(str.format(body=geoBody,  
                    corners=dictMath[geoBody]))  
  
# Ausgabe:  
# Dreieck: 3  
# Viereck: 4  
# Fünfeck: 5
```

2.3 Konsoleneingabe mit `input()`

Mit Hilfe von `input()` erlauben wir dem Nutzer Eingaben über die Konsole. Somit erhalten wir den ersten Grad an Interaktion zwischen Nutzer und Programm.

```
# die input-Methode

input(prompt)
```

Sobald `input()` aufgerufen wird, wartet das Programm mit dem weiteren Ablauf, bis der Nutzer seine Eingabe mit der Eingabetaste bestätigt. Die `input()`-Methode liefert den eingegebenen Wert als String zurück. Damit der Nutzer weiß, was er denn eingeben muss, bietet `input()` den optionalen Standard-Parameter `prompt` an - hierbei handelt es sich um einen leeren String. Geben wir `prompt` nun einen Wert, wird dieser dem Nutzer für die Eingabe angezeigt.

```
# die input-Methode mit prompt-Angabe

userName = input("Geben Sie Ihren Namen ein.")
print("Hallo, " + userName)
```

Achtung:

Der Eingabewert des Nutzer liefert immer einen String zurück. Bei gewünschtem Datentyp muss *gecasted* werden!



- 2.1 Wie verhält sich das Programm bei Aufruf der `input()`-Methode?
- 2.2 Welchen Wert liefert `input()` zurück? Um was für einen Datentyp handelt es sich?
- 2.3 Welchen Effekt hat die Angabe des `prompt`-Parameters?

Bei primitiven Datentypen ist das Umwandeln recht einfach. Bei nicht-primitiven kann es jedoch zu Überraschungen kommen.

```
# die input-Methode mit Typ-Umwandlung

summe = int(input("2 + 3 = "))
print(summe, type(summe), sep=" - ")
```

```
# Eingabe: 5
# Ausgabe: 5 - <class 'int'>

geoKoerper = list(input("Geben Sie einige" +
                      "geometrische Körper an"))
print(geoKoerper)
# Eingabe: ["Dreieck", "Viereck"]
# Ausgabe: [' ', '[', "'", 'D', 'r', 'e',
#           'i', 'e', 'c', 'k', "'", ' ', '',
#           ' ', "'", 'V', 'i', 'e', 'r',
#           'e', 'c', 'k', "'", ']']
```

print(type(geoKoerper[0]))
Ausgabe: <class 'str'>

Python wandelt den String in eine Liste um, jedoch nimmt es jedes einzelne Zeichen der Eingabe als Listenelement. Dies kann durchaus nützlich sein, verfehlt hierbei aber das Ziel. Um die geometrischen Körper als Elemente zu erhalten, nutzen wir die `eval()`-Funktion. Hierbei wird die Eingabe interpretiert und der entsprechende Datentyp zurückgeliefert (Evaluierung).

eval()

Tipp:

`eval()` funktioniert auch bei anderen Collections!

```
# die input-Methode mit eval()

geoKoerper = eval(input("Geometrische Körper: "))
print(geoKoerper)
# Eingabe: ["Dreieck", "Viereck"]
# Ausgabe: ['Dreieck', 'Viereck']
```



- 2.4 Wie verhält sich der Rückgabewert von `input()`, wenn man ihn zu einer Liste umwandelt?
- 2.5 Welche Methode bietet uns Python an, um den Rückgabewert wie gewünscht zu erhalten?

2.4 Dateien lesen und schreiben

Python bietet nativ Möglichkeiten für das Bearbeiten von Dateien. Hierfür werden Objekte erstellt und verwendet, die eine Datei im Quellcode repräsentieren. Mithilfe dieser, im folgenden `fileObjects` genannt, lässt sich der Inhalt einer Datei ändern und wird gespeichert, sobald es im Code mit der `close()`-Methode geschlossen wird.

2.4.1 Dateitypen

In Python werden Dateien in zwei Kategorien eingeteilt. Entweder in Text- oder Binärdateien.

Textdateien bestehen aus Zeilen, die aus einer Zeichensequenz bestehen und mit einem „End of Line“-Zeichen beendet werden. Als solches kann beispielsweise ein Zeilenumbruch oder Komma dienen.

Als Binärdateien werden sämtliche Dateien interpretiert, die keine Textdateien sind. Um diese nutzen zu können, muss der Programmierer eine Möglichkeit zur Verarbeitung bereitstellen.

In diesem Kapitel werden Beispiele anhand einer Textdatei durchgeführt.

2.4.2 Open-Methode

Die tragende Rolle für das Bearbeiten von Dateien in Python ist die `open()`-Methode. Diese erlaubt das Erstellen, Öffnen, Aktualisieren, Lesen und Schreiben einer Datei.

Mithilfe des folgenden Codes wird eine neue Datei erstellt und als `fileObject` geöffnet.

```
# chapters/inputOutput/src/dateienLesenUndSchreiben
# /FileHandlingReadWrite.py

file = open("datei.txt", "x")
```

Zum Erstellen einer neuen Datei wird als erster Parameter ein Dateiname und als zweiter der `"x"`-Modus gewählt. Die Datei wird am gleichen Speicherort, wie die .py-Datei erzeugt, sofern vor dem Dateinamen kein Pfad

angegeben wird. Sollte an der angegebenen Stelle bereits eine Datei mit dem gewählten Namen existieren, bleibt diese unverändert und es wird ein Fehler erzeugt.

Wird das `fileObject` im Code nicht mehr benötigt, wird es mit folgender Zeilen-Anweisung geschlossen:

```
# chapters/inputOutput/src/dateienLesenUndSchreiben  
# /FileHandlingReadWrite.py  
  
file.close()
```

Nach einem Schließen der Datei wird der verwendete Speicherplatz freigegeben. Das Arbeiten ist dann über das entsprechende `fileObject` nicht mehr möglich.

Tipp:

Direkt nach Ausführen der gewünschten Operationen, empfiehlt es sich, die Datei zu schließen. Auf diese Weise wird sichergestellt, dass die Datei nicht unabsichtlich bearbeitet wird.

Für die `open()`-Methode stehen folgende Modi zur Verfügung:

- x:** Erzeugen einer neuen Datei. Sollte bereits eine Datei mit dem gewählten Namen existieren, wird ein Fehler ausgegeben.
- r:** Lesen einer Datei.
- r+:** Lese- und Schreibrechte auf einer Datei.
- a:** Hinzufügen von Inhalt am Ende der Datei. Erzeugt eine neue Datei, falls keine mit dem gewählten Namen an der angegebener Pfadangabe existiert.
- a+:** "a" wird um das Leserecht auf der Datei ergänzt.
- w:** Schreiben einer Datei. Überschreibt den Inhalt der Datei. Sollte die Datei mit dem gewählten Namen noch nicht existieren, wird eine neue erzeugt.
- w+:** "w" wird um das Leserecht auf der Datei ergänzt.
- t, b:** Angabe, ob die Datei als Text- "t" oder Binärdatei "b" interpretiert werden soll. Diese Modi können jeweils zu den anderen hinzugefügt wer-

den. Standardmäßig wird die Datei als Text interpretiert, "t" kann hierbei weggelassen werden.

```
# chapters/inputOutput/src/dateienLesenUndSchreiben
# /FileHandling.py
# Typ des Datei-Objekts festlegen
# Beispiel am "r"-Modus

file = open("datei.txt", "rt")    # Text
# oder
file = open("datei.txt", "r")     # Text

file = open("datei.txt", "rb")    # Binaer
```

2.4.3 Methoden

Die zuvor erstellte Datei hat noch keinen Inhalt. Um dies zu ändern, wird die datei.txt im "w"-Modus geöffnet. Danach kann der Datei über die write()-Methode wie folgt eine Textzeile hinzugefügt werden.

```
# chapters/inputOutput/src/dateienLesenUndSchreiben
# /FileHandlingReadWrite.py
# datei.txt Text hinzufuegen

file = open("datei.txt", "w")
file.write("Hallo Welt.")  # eine Zeile hinzufuegen
file.close()
```

Mithilfe der writelines()-Methode kann die Datei mit einer List von String-Werten beschrieben werden.

```
# chapters/inputOutput/src/dateienLesenUndSchreiben
# /FileHandlingReadWrite.py
# datei.txt Text hinzufuegen

textList = ["Hallo Welt.\n",
            "Das ist ein...\\n", "Beispieltext"]
# Ohne \\n wuerde der gesamte Inhalt in eine Zeile geschrieben

file = open("datei.txt", "w")
file.writelines(textList)
file.close()
```

Die Datei in dem Beispiel wurde erstellt, geöffnet, beschrieben und überschrieben. Als nächstes soll der Inhalt aus der Datei auf der Konsole ausgegeben werden. Hierzu wird der Modus, in dem die `datei.txt` geöffnet wird, auf "r" gestellt. Die `read()`-Methode liefert den Inhalt als String, welcher über `print()` ausgegeben wird.

```
# chapters/inputOutput/src/dateienLesenUndSchreiben
# /FileHandlingReadWrite.py
# Auslesen und Ausgeben des Inhalts der datei.txt

file = open("datei.txt", "r")
print(file.read())
file.close()

# Ausgabe:
# Hallo Welt.
# Das ist ein
# Beispieltext
```

Soll eine einzelne Zeile ausgegeben werden, kann die `readline()`-Methode verwendet werden. Mittels eines int-Werts als Parameter kann eine Grenze festgelegt werden, die bestimmt, bis zu welcher Position die Zeile ausgelesen werden soll. Ohne Angabe eines Parameters wird die gesamte Zeile ausgelesen. Dies gilt sowohl für die `read()`- als auch für die `readline()`-Methode.

Wird der folgende Code ausgeführt, fällt auf, das die `datei.txt` drei Zeilen enthält und für jede Zeile die Anweisung `print(fileObject.readline())` benötigt wird, um den Inhalt vollständig auszugeben. Folglich muss im `fileObject` die aktuelle Leseposition gespeichert sein.

```
# chapters/inputOutput/src/dateienLesenUndSchreiben
# /FileHandlingReadWrite.py
# Code-Beispiel

fileObject = open("datei.txt", "r")
print(fileObject.readline())
print(fileObject.readline())
print(fileObject.readline())
fileObject.close()

# Ausgabe:
# Hallo Welt.
#
```

```
# Das ist ein
#
# Beispieltext
```

Anstelle der Ausgabe über `read()` oder der mehrfachen Verwendung von `readline()`, können wir auch über das `fileObject` iterieren. In diesem Fall verwenden wir eine `for`-Schleife.

```
# chapters/inputOutput/src/dateienLesenUndSchreiben
# /FileHandlingReadWrite.py
# Ausgeben des Inhalts der datei.txt

file = open("datei.txt", "r")
for line in file:
    print(line)
file.close()

# Ausgabe
# Hallo Welt.
#
# Das ist ein
#
# Beispieltext
```

Eine weitere Alternative ist die `readlines()`-Methode, die eine Liste mit den Zeilen der Datei als Inhalt liefert.

```
# chapters/inputOutput/src/dateienLesenUndSchreiben
# /FileHandlingReadWrite.py
# Auslesen und Ausgeben des Inhalts der datei.txt

file = open("datei.txt", "r")
print(file.readlines())
file.close()

# Ausgabe: ['Hallo Welt.\n', 'Das ist ein...\n',
# 'Beispieltext']
```

Wird die `readlines()`-Methode zweimal hintereinander verwendet, erhalten wir folgende Ausgabe:

```
# Ausgabe:
```

```
['Hallo Welt.\n', 'Das ist ein\n', 'Beispieltext']
[]
```

Nach dem ersten Aufruf der Methode befindet sich der Zeiger am Ende des `fileObject`. Somit kann bei dem zweiten Aufruf kein Inhalt mehr ausgelesen werden. Mithilfe der `tell()`-Methode kann die aktuelle Position des Zeigers ausgegeben werden. Fügen wir den folgenden Code vor den `readlines()`-Methoden ein, kann der Zeiger verfolgt werden.

```
# chapters/inputOutput/src/dateienLesenUndSchreiben
# /FileHandlingReadWrite.py
# Ausgeben der Zeigerposition

file = open("datei.txt", "r")
print(file.tell())
print(file.readlines())
print(file.tell())
print(file.readlines())
file.close()

# Ausgabe:
# 0
# ['Hallo Welt.\n', 'Das ist ein... \n', 'Beispieltext']
# 41
# []
```

Soll die Ausgabe beider Lists identisch sein, muss der Zeiger an den Anfang zurückgesetzt werden. In Python existiert für diesen Zweck die `seek()`-Methode. Wird der Zeiger direkt nach der ersten Verwendung der `readlines()`-Methode auf die Position 0 zurückgesetzt, erhalten wir die gewünschte Ausgabe.

```
# chapters/inputOutput/src/dateienLesenUndSchreiben
# /FileHandlingReadWrite.py
# Zeiger zuruecksetzen

file = open("datei.txt", "r")
print(file.tell())
print(file.readlines())
file.seek(0)
print(file.tell())
print(file.readlines())
file.close()
```

```
# Ausgabe:
# 0
# ['Hallo Welt.\n', 'Das ist ein...\n', 'Beispieltext']
# 0
# ['Hallo Welt.\n', 'Das ist ein...\n', 'Beispieltext']
```

2.4.4 With-Statement

Bisher mussten wir in dem Code-Beispiel die `open()`-Methode verwenden und darauf achten, dass das `fileObject` mit `close()` nach Gebrauch wieder geschlossen wird.

Alternativ kann das `with`-Statement genutzt werden. So wird die Datei nach Verwendung automatisch geschlossen, ohne die explizite Angabe von `close()`. Der Code zum Auslesen der `datei.txt` sieht wie folgt aus:

```
# chapters/inputOutput/src/dateienLesenUndSchreiben
# /FileHandlingReadWrite.py
# with-Statement zum Oeffnen der Datei

with open("datei.txt", "r") as file:
    print(file.read())

# Ausgabe:
# Hallo Welt.
# Das ist ein...
# Beispieltext
```

2.4.5 Attribute

Jedes `fileObject` besitzt Attribute, die Auskunft über das jeweilige Objekt angeben.

closed: Gibt Auskunft darüber, ob die Datei geschlossen wurde. Als Rückgabe erhalten wir einen boolean-Wert.

mode: Liefert den Zugriffsmodus auf die Datei als String zurück.

name: Liefert den Namen der geöffneten Datei als String zurück.

```
# chapters/inputOutput/src/dateienLesenUndSchreiben
# /FileHandlingAttributes.py
# Attribute des Datei-Objekts

with open("datei.txt", "r") as file:
    print(file.closed)
    print(file.mode)
    print(file.name)

# Ausgabe:
# False
# r
# datei.txt
```

2.5 JSON

JavaScript Object Notation (JSON) ist ein Format für den Austausch von Daten, das unabhängig von der Programmiersprache ist. Aufgrund von Konventionen, die dieses Format mit Programmiersprachen aus der C-Familie, wie C, C++, Java oder Python teilt, liefert es eine Programmierern bekannte Textstruktur.

In Python 3 ist nativ das json-Package enthalten, das das Arbeiten mit dem JSON-Format ermöglicht. Mithilfe des folgenden Codes binden wir das Package in das Projekt ein.

```
# chapters/inputOutput/src/jsonInPython/JsonInPython.py
# JSON Package

import json
```

Ein gegebener JSON-String wird über die `loads()`-Methode in ein in Python existierendes, entsprechendes Objekt geparsst. In diesem Fall wird ein Dictionary angelegt.

```
# chapters/inputOutput/src/jsonInPython/JsonInPython.py
# JSON zu Python

student = '{"name": "Student", "age": 24, "height": 1.8}'
studentDict = json.loads(student)
```

JSON zu Python

```

print(student)
print(studentDict)
print(studentDict["height"])

# Ausgabe:
# {"name": "Student", "age": "24", "height": "1.8"}
# {'name': 'Student', 'age': '24', 'height': '1.8'}
# 1.8

```

Für das Umwandeln eines Python-Objekts in einen JSON-String verwenden wir die `dumps()`-Methode.

```

# chapters/inputOutput/src/jsonInPython/JsonInPython.py
# Python zu JSON

newStudent = {
    "name": "Neuer Student",
    "age": 26,
    "height": 2.0
}

newStudentJSON = json.dumps(newStudent)
print(newStudent)
print(newStudentJSON)

# Ausgabe:
# {"name": "Neuer Student", "age": 26, "height": 2.0}
# {"name": "Neuer Student", "age": 26, "height": 2.0}

```

Konvertieren wir Python- zu JSON-Objekte, werden diese im JSON-Äquivalent (JavaScript) angelegt.

Wenn wir einen Dictionary mit mehreren Schlüssel-Objekt-Paaren anlegen, werden wir bei der Ausgabe des JSON-Objekts feststellen, dass diese auf eine Zeile beschränkt sind.

```

# chapters/inputOutput/src/jsonInPython/JsonInPython.py
# JSON-string formatieren

demoStudent = {
    "name": "Harry",
    "age": 20,
    "height": 1.78,

```

```

    "assistant": False,
    "pets": ("Katze", "Maus"),
    "cars": None,
    "projects": [
        {"name": "Pythonkurs", "done": True},
        {"name": "Giraffen zaehmen", "done": False}
    ]
}

print(json.dumps(demoStudent))

# Ausgabe:
# {"name": "Harry", "age": 20, "height": 1.78 . . .}

```

Zur Formatierung unserer Ausgabe verwenden wir die `dumps()`-Methode. Mithilfe des `indent`-Parameters können wir festlegen, ob und wie weit die Textstruktur eingerückt werden soll. Der `separators`-Parameter legt die Trennzeichen fest und mit `sort_keys=True` wird die Ausgabe der Schlüssel lexikografisch sortiert.

```

# chapters/inputOutput/src/jsonInPython/JsonInPython.py
# JSON-string formatieren

print(json.dumps(demoStudent, indent=4,
                 sort_keys=True, separators=(" ", " = ")))

# Ausgabe:
# {
#     "age" = 20 &
#     "assistant" = false &
#     "cars" = null &
#     "height" = 1.78 &
#     "name" = "Harry" &
#     "pets" = [
#         "Katze" &
#         "Maus"
#     ] &
#     "projects" = [
#         {
#             "done" = true &
#             "name" = "Pythonkurs"
#         } &

```

```

#           {
#           "done" = false &
#           "name" = "Giraffen zaehmen"
#       }
#   ]
# }
```

2.6 Zusammenfassung

In diesem Kapitel haben wir uns mit dem lesen und beschreiben einer Datei auseinandergesetzt. Dies geschieht in Python mithilfe eines `fileObject` um eine Datei zu erstellen, ändern, löschen und abzuspeichern. Dabei kann eine Datei als Textdatei oder Binärdatei interpretiert werden. Eine der wichtigsten Methoden stellt hierbei die `open()`-Methode dar. Diese ermöglicht uns das Erstellen, Öffnen, Aktualisieren, Lesen und Beschreiben einer Datei. Dabei ist zu beachten, dass die Datei direkt nach der Ausführung der gewünschten Operationen mithilfe der `close()`-Methode geschlossen wird. Um dies nicht zu vergessen, besteht in Python auch die Möglichkeit ein automatisches Schließen mit dem `with`-Statement zu erwirken. Abschließend haben wir noch den Zugriff auf die wichtigsten Attribute, die ein `fileObject` besitzt, kennengelernt und uns mit dem Standardisierten Datenaustausch mittels JSON beschäftigt.



Übungsaufgaben

Aufgabe 2.6.1

Schreiben Sie einen `print()`-Befehl, welcher drei Variablen als Parameter annimmt und diese mittels Seperator und End-Parameter in die entsprechende Form bringt.

```

# Ausgabe mit Seperator und End-Parameter

a = "Drei"
b = "Vier"
c = "Fünf"

print(...)

# Ausgabe:
```

```
# Dreieck, Viereck, Fünfeck
```

Aufgabe 2.6.2

Schreiben Sie ein Programm, das es dem Nutzer ermöglicht, eine beliebige Anzahl an Eingaben zu tätigen. Diese sollen in einer Liste gespeichert und nach der Eingabe des Wortes `exit` in der Konsole ausgegeben werden.

Tipp:

Für die Eingabe wird hier eine `while`-Schleife genutzt, welche immer `True` ist (Endlosschleife). Das Ausbrechen aus einer Schleife gelingt mit dem Schlüsselwort `break`.

```
# Ein- und Ausgabe durch Nutzer-Interaktion
```

```
inputList = []
print("Füllen Sie mittels Eingabe eine"
+ "Liste mit geometrischen Körpern")
print("Bei Eingabe 'exit' wird die " +
    "Eingabeschleife beendet.\n")

while True:
    ...
    print("\nDie geometrischen Körper lauten:")
    ...
```

Aufgabe 2.6.3

Gegeben sind drei Zahlen, welche mit einem `print()`-Befehl ausgegeben werden sollen, sodass sie untereinander stehen und linksbündig geschrieben sind.

Verzichten Sie auf den Seperator-Parameter und Formatierungsanweisungen wie "<" für die Linksbündigkeit. Nutzen Sie hierfür die minimale Länge mit Wert 10.

```
# Ausgabe mittels minimaler Länge

a = 1234567890
b = 12345
c = 5

print(...)
```

```
# Ausgabe:  
# 1234567890  
#      12345  
#          5
```

Aufgabe 2.6.4

Schreiben Sie ein Programm, das es dem Nutzer ermöglicht, eine beliebige Anzahl an Eingaben zu tätigen. Diese sollen in einer Liste gespeichert und nach der Eingabe des Wortes `exit` in der Konsole ausgegeben werden. Bei den Eingaben handelt es sich um geometrische Körper, diesen Teil der Aufgabe sollten Sie bereits aus der zweiten Übung dieses Kapitels kennen.

Anschließend soll der Nutzer zu jedem geometrischen Körper die Anzahl der Ecken angeben. Die Angabe der Ecken sowie der geometrische Körper werden in einem Dictionary gespeichert. Um dem Nutzer zu versichern, dass alles funktioniert hat, geben Sie den Inhalt des Dictionarys in der Konsole aus.

Kapitel 3

Funktionen und Module

In diesem Kapitel gehen wir auf die Nutzung von Funktionen in Python ein. Eine Funktion bildet einen Code-Block ab, also eine Sequenz von Befehlen, die eine bestimmte *Funktion* erfüllt.

Dieser Code-Block wird mit dem Schlüsselwort `def` gestartet, gefolgt vom Namen der Funktion, anschließend von Klammern (welche Input-Parameter beinhalten können). Zum Schluss der Funktionsdefinition folgt noch ein Doppelpunkt, nach diesem kommt die Befehlssequenz. Soll die Funktion Werte zurückliefern, dann steht am Ende der Sequenz das `return`-Schlüsselwort.

```
# Definition einer Funktion in Python

def funktionsname (parameter):
    ...
    return rueckgabewert
```

Achtung:

Eine Funktion liefert immer einen Wert zurück. Wird keiner angegeben, so wird der Wert `None` als Rückgabewert festgelegt.

3.1 Vorteile von Funktionen

Warum benutzen Programmierer Funktionen? Diese bieten eine Vielzahl an Vorteilen, wie z.B.

- das Aufteilen von komplexen Aufgaben in mehrere simple

- das Verhindern von Code-Duplikationen
- bessere Lesbarkeit/Erweiterbarkeit/Veränderbarkeit
- vereinfachtes Debugging

3.1.1 Aufteilen von komplexen Aufgaben

Bei komplexen Aufgaben kommt es schnell zu einer hohen Anzahl an Code-Zeilen. Das kann für den Programmierer und die Kollegen, die an den Aufgaben mitwirken, problematisch sein. Der Code ist schlecht zu lesen und die Fehlersuche schwierig. Schritte, die immer wieder gebraucht werden, wie z.B. das Empfangen, Auslesen, Interpretieren und Aufbereiten von Daten. Anfänger könnten den Fehler machen, diese Aufgaben in einer großen komplexen Funktion zu schreiben.

```
# Beispiel für schlechten Umgang mit komplexen Prozessen

def processData(source):
    ...
    return finalData
```

Es ist mühsam herauszufinden, was genau in dieser Funktion passiert. Ein Beispiel zum Aufteilen des Code-Blocks könnte weiterhelfen.

```
# Aufteilung des komplexen Prozesses
# in mehrere kleine Prozesse

def processData(source):
    rawData = readData(source)
    parsedData = parseData(rawData)
    editedData = editData(parsedData)
    finalData = sortData(editedData)
    return finalData
```



- 3.1 Was sind die Vorteile, die beim Aufteilen von komplexen Aufgaben in mehrere simple auftreten?
- 3.2 Welche Voraussetzungen finden Sie geeignet, um zu beschließen, dass eine Funktion nicht weiter aufgeteilt werden sollte?

3.1.2 Reduktion von Code-Duplikationen

Bestimmte Prozesse werden beim Programmieren immer wieder benötigt. Bei der Arbeit mit Datensätzen ist es üblich, diese nach gewissen Kriterien zu sortieren. In einer Datenbank ist die Sortierung nach der Identifikationsnummer vorteilhaft. Diese *Funktion* soll nicht für jeden Datensatzaufruf dupliziert werden müssen. Daher wird dieser Prozess in einer Funktion gespeichert, auf die wir von verschiedenen Positionen im Programm zugreifen können.

3.1.3 Bessere Lesbarkeit, Erweiterbarkeit, Veränderbarkeit

Wie in Unterabschnitt 3.1.1 zu sehen ist, bringt das Aufteilen des Codes in spezifische Funktionen eine bessere Lesbarkeit mit sich. Der Nutzer muss den Code nicht erst interpretieren. Bei intelligent gewählten Funktionsnamen versteht er, was in der Funktion passiert. Besonders beim Debuggen kann das Vorteile mit sich bringen, da der Programmierer nicht an den falschen Stellen suchen muss.

```
array = calculateArray()  
  
sortedArray = quickSort(array)
```

In diesem Beispiel weiß der Nutzer, dass das Array durch einen QuickSort-Algorithmus sortiert wird. Sollte nun auffallen, dass es sich um falsche Werte handelt, muss der Programmierer nur die calculateArray-Funktion ansehen, sind die Werte falsch sortiert, so wird die quickSort-Funktion näher betrachtet.

Durch das Kapseln von Prozessen in einzelne Funktionen sind diese auch einfach erweiterbar und veränderbar. Wäre der Code nur dupliziert worden, müsste der Nutzer diesen an allen Stellen ändern. Da das Programm aber an diesen Stellen nur die Funktion aufruft, muss nur diese Funktion erweitert oder verändert werden.



- 3.3 Welchen Vorteil bietet das Auslagern von Code-Duplikationen in eine Funktion? Denken Sie dabei daran, dass Code immer unter der Voraussetzung geschrieben werden sollte, dass er in Zukunft nochmal geändert oder erweitert werden muss.

3.2 Gültigkeitsbereich von Variablen und Funktionen

Je nachdem, wie und wo Funktionen definiert sind, können diese zu anderen Ergebnissen führen. Zur Verdeutlichung folgt ein einfaches Beispiel (3.2).

```
# Beispiel zu Gültigkeitsbereichen

def myFunction():
    # lokaler Gültigkeitsbereich der Funktion
    a = 1
    print('myFunction:', a)

# globaler Gültigkeitsbereich
a = 0
myFunction()
print('global:', a)

# Ausgabe
# myFunction: 1
# global: 0
```

In beiden Bereichen benutzen wir die Variable `a`. Die Funktion wird nach dem Initialisieren der Variable aufgerufen. Warum erhalten wir also zwei unterschiedliche Werte?

Der Grund: es handelt sich nicht um die gleiche Variable, da beide Variablen in verschiedenen Gültigkeitsbereichen definiert werden. Ließen wir die lokale Zuweisung aus, würde zweimal der Wert 0 ausgegeben werden. Python sucht nach dem nächstmöglichen Gültigkeitsbereich: `lokal`, `umschließend`, `global` und `built-in`.

Nun ein Beispiel mit verschachtelten Funktionen:

```
# Verschachtelte Funktionen

def enclosing():
    a = 1

    def innerFunction():
        a = 2
        print('innerste:', a)
```

```
innerFunction()  
print('umschließend:', a)  
  
a = 0  
enclosing()  
print('global:', a)  
  
# Ausgabe  
# innerste: 2  
# umschließend: 1  
# global: 0
```



3.1 Welche Gültigkeitsbereiche gibt es in Python?

- 3.2 Bei der Nutzung einer Variable sucht Python nach dem nächstmöglichen Gültigkeitsbereich. Wie ist die Reihenfolge der Gültigkeitsbereiche?

3.2.1 Statements zu Gültigkeitsbereichen - global und nonlocal

Nicht nur durch die Positionen werden Gültigkeitsbereiche definiert, auch durch die Schlüsselwörter `global` und `nonlocal` können wir den Gültigkeitsbereich bestimmen.

Durch `nonlocal` wird eine Variable auf die nächst umschließende Definition festgelegt (3.2.1). nonlocal-Statement

```
# Nonlocal Statement  
  
def enclosing():  
    a = 1  
  
    def innerFunction():  
        nonlocal a  
        a = 2  
        print('innerste:', a)  
  
    innerFunction()  
    print('umschließend:', a)  
  
a = 0
```

```
enclosing()
print('global:', a)

# Ausgabe:
# innerste: 2
# umschließend: 2
# global: 0
```

Achtung:

Würde in der enclosing-Funktion `a` auf `nonlocal` gesetzt, dann käme es zu einer `Exception`, da die nächste Ebene `global` ist.

global-Statement

Das Gleiche können wir mit dem globalen Gültigkeitsbereich machen, wie in 3.2.1 gezeigt.

```
# Global Statement

def enclosing():
    a = 1

    def inner():
        global a
        a = 2
        print('innerste:', a)

    innereFunction()
    print('umschließend:', a)

a = 0
enclosing()
print('global:', a)

# Ausgabe
# innerste: 2
# umschließend: 1
# global: 2
```

Während `nonlocal` nur den nächst umschließenden Gültigkeitsbereich wählt - in welcher die Variable deklariert wurde - greift `global` immer auf den globalen Gültigkeitsbereich zu.



- 3.3 Von welcher Konvention befreit uns die Nutzung von Statements bezüglich des Gültigkeitsbereiches?
- 3.4 Was passiert, wenn eine Variable mit dem Statement `nonlocal` gekennzeichnet wird, es sich beim nächsten Gültigkeitsbereich jedoch um den globalen handelt?

3.3 Input-Parameter

In diesem Abschnitt sehen wir uns die verschiedenen Arten von Übergabeparametern genauer an, sowie die Möglichkeiten, diese an eine Funktion zu übergeben.

```
# Übergabe eines primitiven Datentyps

def myFunction(x):
    x = 1

x = 2
myFunction(x)
print(x)

# Ausgabe:
# 2
```

Mittels Codes (3.3) wird der Wert '2' ausgedruckt. Aus dem vorherigen Abschnitt über Gültigkeitsbereiche wissen wir, dass es sich hierbei - trotz gleichem Namen - um zwei verschiedene Variablen handelt. Eine Integer-Variable im lokalen und eine im globalen Gültigkeitsbereich. Dies gilt jedoch nur für primitive Datentypen, für nicht-primitive Datentypen verhält es sich anders.

```
# Übergabe eines nicht-primitiven Datentyps

def myFunction(x)
    x[0] = 100

x = [0,1,2]
myFunction(x)
print(x)
```

Da es sich hier (3.3) nicht um einen primitiven Datentyp handelt, stellt der Input-Parameter eine Referenz dar, womit der Wert von `x` auch in der Methode geändert wird. Beim Ausgeben erhalten wir: `[100, 1, 2]`.

Es ist wichtig, sich dieses Verhalten zu merken, da beim objekt-orientierten Programmieren sonst Fehler auftreten.



- 3.1 Unter welcher Voraussetzung führen Änderungen eines Übergabeparameters innerhalb einer Funktion zu Änderungen außerhalb?
- 3.2 Welche nicht-primitiven Datentypen kennen wir?

3.3.1 Arten von Input-Parametern

In folgendem Unterabschnitt gehen wir auf verschiedene Arten von Input-Parametern ein.

Positional

Diese Art von Parametern dürfte bereits bekannt sein. Es handelt sich um eine endliche Anzahl von Parametern, die man von links nach rechts liest.

```
# Funktion mit positionalen Parametern

def myFunction(x, y, z):
    ...
myFunction(1, 2, 3)
```

Schlüsselwort

Bei Schlüsselwort-Parametern ist die Reihenfolge der Parameter nicht elementar, da die Variablen über den Namens-Wert übergeben werden.

Achtung:

Bei der Verwendung von positionalen und Schlüsselwort-Parametern, müssen positionale Parameter immer links der Schlüsselwort-Parameter stehen.

```
# Funktion mit Schlüsselwort-Parametern

def myFunction(x, y, z):
    ...
myFunction(x = 1, z = 3, y = 2)
```

Standard

Standard-Parameter folgen einer ähnlichen Syntax wie Schlüsselwort-Para-

meter, sie werden jedoch in der Funktionsdefinition festgelegt und nicht beim Aufruf. Zu beachten ist, dass beim Verwenden von positionalen und Standard-Parametern alle Standard-Parameter *nach* den positionalen stehen müssen.

```
# Funktion mit Standard-Parametern

def myFunction(x, y = 10, z = 20):
    print(x, y, z)

myFunction(0)                      # 0 10 20
myFunction(y = 20, x = 10, z = 30)  # 10 20 30
myFunction(0, z = 1)                # 0 10 1
```

Variable Parameter ermöglichen quasi eine unendliche Anzahl an Übergabe-Parametern. Möchte ein Nutzer beispielsweise den Durchschnitt mehrerer Zahlen wissen, so könnte die Funktion wie folgt aussehen:

Variable

```
# Funktion mit variablen Parametern

def average(*numbers):
    value = 1
    for number in numbers:
        value = value * number

    value = value / len(numbers)
    # durch die Anzahl an Parametern
    return value

result = average(5, 5, 10, 8, 2)
print(result) # druckt das Ergebnis 6
```

Die multiplen Parameter werden als Liste gehandhabt. Bei der Definition von variablen Parametern in einer Funktion ist es wichtig, dass sie als letztes Argument stehen.



- 3.3 Worin liegt der Unterschied zwischen Standard- und Schlüsselwort-Parametern?
- 3.4 Worauf ist bei gleichzeitiger Nutzung von positionalen und Standard-Parametern zu achten?
- 3.5 Welchen Nutzen bieten variable Parameter? Gibt es Kontrollstrukturen, bei denen sich variable Parameter anbieten?

3.4 Lambda-Funktionen

Lambda-Funktionen (oder auch anonyme Funktionen) bieten den Vorteil zur besseren Lesbarkeit und sollten benutzt werden, wenn bspw. die Funktion in einer Zeile geschrieben werden kann.

```
# Syntax einer Lambda-Funktion
# lambda_name = lambda [Parameter] : [Ausdruck]

addition = lambda a, b: a + b
```

Wie in Listing 3.4 zu sehen ist, werden Lambda-Funktionen mit dem Schlüsselwort `lambda` definiert - und nicht mit `def`.

3.5 Modularisierung

Unter Modularisierung versteht man die Aufteilung des Programmcodes in Modul. Ein Modul stellt dabei verschiedene Datentypen, Funktionen und Werte bereit. Durch das Auslagern dieser Elemente in Modul wird das Projekt besser strukturiert. Zudem wird die Lesbarkeit des Quellcodes durch das Auslagern von Funktionen verbessert.

Bei den Modulen wird zwischen lokalen und globalen Modulen unterschieden. Während lokale Module nur in einem Projekt genutzt werden, können globale Module projektübergreifend verwendet werden. Ein Beispiel für ein globales Modul ist das Modul `math` der Standardbibliothek, welches mathematische Funktionen und Konstanten bereitstellt.

Ob ein Modul lokal oder global ist, hängt vom Speicherort ab. Ein lokales Modul ist in der Regel im Programmverzeichnis oder in einem Unterverzeichnis hinterlegt. Ein globales Modul wird in einem bestimmten Verzeichnis der Python-Installation angelegt. Hierzu gehört beispielsweise das Verzeichnis `site-packages`, in dem auch einige Module von Drittanbietern installiert werden.



3.1 Welche Vorteile bietet uns die Modularisierung?

3.2 Was ist der Unterschied zwischen einem lokalen und einem globalen Modul?

3.5.1 Erstellung eines lokalen Moduls

Zur Erstellung eines lokalen Moduls wird eine Datei *myModul.py* im Programmverzeichnis angelegt:

```
def hello(constantName):  
    print("Hello, " + constantName)  
  
MAX = "Maximilian"
```

Der Inhalt dieser Datei führt keinerlei Code aus, sondern stellt lediglich eine Funktion und eine Konstante bereit, welche später im Hauptprogramm genutzt werden können.

3.5.2 Module verwenden

Um lokale und globale Module verwenden zu können, müssen diese zunächst im Hauptprogramm eingebunden werden. Hierzu wird die *import*-Anweisung verwendet. Prinzipiell ist es egal, wo dies im Quellcode geschieht. Es empfiehlt sich jedoch alle Module zu Beginn des Quelltextes einzubinden. Die *import*-Anweisung besteht aus dem Schlüsselwort *import*, gefolgt vom Namen des Moduls.

```
# Einbinden des zuvor erstellten Moduls  
  
import myModule
```

Zu Beachten ist, dass beim Einbinden von Modulen die Dateiendung entfällt. Mit einer *import*-Anweisung können auch mehrere Module eingebunden werden. Hierzu werden die Modulnamen hinter dem Schlüsselwort - durch Kommas getrennt - aufgelistet:

```
import myModule, math  
  
# anstelle von:  
  
import myModule  
import math
```

Durch das Einbinden eines Moduls wird ein neuer Namensraum mit dem Modulnamen erzeugt. Über den Namensraum können alle Elemente des Moduls angesprochen werden:

```
import myModule

myModule.hello("World")
myModule.hello(myModul.MAX)
```

Ausgabe:

```
Hello, World
Hello, Maximilian
```



3.3 Wie wird ein Modul erstellt und wie kann man es verwenden? Welches Schlüsselwort wird zum Einbinden benötigt?

3.4 Was passiert beim Einbinden eines Moduls?

Mit Hilfe der *import/as*-Anweisung ist es möglich den Namen des Namensraums zu verändern:

```
import myModule as myBib
myBib.hello(World)
```

Tipp:

Es gilt zu beachten, dass nach dieser Anweisung das Modul *myModule* ausschließlich über den Namensraum *myBib* zu erreichen ist.

from x import a Python bietet zudem die Möglichkeit *einzelne* Elemente aus einem Modul zu importieren. Hierzu wird die *from/import*-Anweisung verwendet:

```
from math import sin, cos, tan, sinh, cosh, tanh
```

Falls die Liste der zu importierenden Elemente etwas länger ausfallen sollte, kann die Aufzählung in mehreren Zeilen erfolgen, um so die Lesbarkeit des Codes zu verbessern. Zur Realisierung müssen zunächst die einzubindenden Elemente in runden Klammern gesetzt werden. Anschließend können beliebig viele Zeilenumbrüche innerhalb der runden Klammern erfolgen. Folgendes Beispiel zeigt eine *from/import*-Anweisung mit einem Zeilenumbruch:

```
from math import (sin, cos, tan,
                  sinh, cosh, tanh)
```

Mit einem Stern können *alle* Elemente des Moduls importiert werden:

```
from myModule import *
```

Es gilt zu beachten, dass bei der *from/import*-Anweisung kein eigener Namensraum erzeugt wird. Die Elemente werden stattdessen in den *globalen* Namensraum eingebunden. Dies hat zur Folge, dass der Namensraum bei einem Zugriff auf ein Element nicht mehr angegeben werden muss:

```
from myModul import hello
hello("World")           # statt: myModule.hello("World")
```



- 3.5 Mit welchem Schlüsselwort können wir den Namensraum eines Moduls ändern?
- 3.6 Ist es möglich nur einzelne Elemente eines Moduls einzubinden?
- 3.7 Was gilt es beim Einbinden von Modulen bezüglich des Namensraums zu beachten?

Da die Elemente bei der *from/import*-Anweisung in den globalen Namensraum eingebunden werden, können bereits vorhandene Referenzen kommentarlos überschrieben werden.

```
pi = 42
print("pi hat den Wert:")
print(pi)

from math import pi           # pi wird überschrieben
print("pi hat den Wert:")
print(pi)
```

Ausgabe:

```
pi hat den Wert:
42
pi hat den Wert:
3.141592653589793
```

Um derartige Fehler zu minimieren, sollte die *from/import*-Anweisung sparsam verwendet werden - also nur dann, wenn einzelne Elemente aus einem Modul benötigt werden. Wenn alle oder fast alle Elemente aus einem Modul benötigt werden, empfiehlt es sich, die *import*-Anweisung zu verwenden.



- 3.8 Es wird ein Modul importiert, welches ein Element besitzt, das es schon im derzeitigen Namensraum gibt. Was passiert mit dem Element?
- 3.9 Welche Wege gibt es, um dieses Verhalten zu vermeiden?



Übungsaufgaben

Aufgabe 3.5.1

Hier ist ein *Hello World* Beispiel:

```
# Hallo Welt
print("Hello, World")
```

Schreiben Sie eine Funktion *greet()*, welche eine Grußnachricht in der Konsole ausgibt. Ohne Parameter wird immer *Hello, World* ausgegeben, ansonsten wird der Inhalt begrüßt.

```
# Gruß-Benachrichtigung
greet ("Max Mustermann")
# Ausgabe: Hello, Max Mustermann
```

Aufgabe 3.5.2

Schreiben Sie eine Funktion, welche eine unbestimmte Anzahl von Personen grüßt - zugegeben, ein lästiger Zeitgenosse.

```
# Multiple Gruß-Benachrichtigung
def greetMultiple(...):
    ...
greetMultiple("Max", "Rainer", "Denis", "Frauke",
              "Marc", "Melissa", "Nadine")
# Ausgabe:
Hello, Max
Hello, Rainer
Hello, Denis
Hello, Frauke
Hello, Marc
```

```
Hello, Melissa  
Hello, Nadine
```

Aufgabe 3.5.3

Eine unbestimmte Anzahl von geometrischen Körpern nimmt an einem Wettbewerb teil. Die ganze Welt soll wissen, wer denn der Körper mit den meisten Ecken ist. Schreiben Sie eine Funktion, welche diesen Körper findet und *global* festlegt.

Alle Körper sollen als jeweilige Tupel mit der Anzahl an Ecken und ihrem Namen übergeben werden. Die Variable *geometryWithMostCorners*, welche den Gewinner trägt, darf nicht außerhalb der Methode deklariert werden.

```
# Erster Platz  
  
def getGeometryWithMostCorners(...):  
    ...  
  
getGeometryWithMostCorners(  
    (3, "Dreieck"),  
    (8, "Achteck"),  
    (5, "Fünfeck"))  
print(geometryWithMostCorners[1])  
# Ausgabe: Achteck
```

Aufgabe 3.5.4

Lambda-Funktionen haben ihre Vorteile, machen wir uns mit ihnen vertraut! Schreiben Sie Lambda-Funktionen für das Quadrieren und Wurzelziehen einer Zahl. Nutzen Sie diese in einer eigenen Funktion namens *pythagoras()*, um den Satz des Pythagoras für zwei Zahlen zu berechnen.

```
# Satz des Pythagoras  
  
pow = lambda ...  
root = lambda ...  
  
def pythagoras(a, b):  
    ...  
  
print(pythagoras(3, 2))  
# Ausgabe: ~3.6055
```

Tipp:

Importieren Sie das `math`-Modul und nutzen Sie die Funktion `sqrt`, um die Wurzel zu ziehen.

Aufgabe 3.5.5

Erstellen Sie ein eigenes Modul mit den bereits erstellten Methoden aus dieser Übung. Erweitern Sie das Modul um die Funktionen `Summe()` und `Produkt()`. Beide akzeptieren eine unbestimmte Anzahl von Parametern.

```
# Neue mathematische Funktionen

def sum(...):
    ... # addiert alle Zahlen zu einer Summe

def product(...):
    ... # multipliziert alle Zahlen zu einem Produkt
```

Speichern Sie das Modul ab und nutzen Sie es in einem anderen Programm. Kopieren Sie den `print`-Befehl und vergleichen Sie das Ergebnis.

```
# Nutzen des eigenen Moduls

import myModule

print(myModule.sum(
    myModule.product(2, 3), myModule.product(3, 5)))
# Ausgabe: 21
print(myModule.product(
    myModule.sum(2, 3, 4, 5), myModule.product(0, 9)))
# Ausgabe: 0
```

Aufgabe 3.5.6

Wir möchten nicht alle Funktionen unseres Moduls nutzen. Ändern Sie die `import`-Anweisung so ab, dass nur die Funktionen `sum()` und `product()` eingebunden werden.

Kapitel 4

Testen

4.1 Grundlegende Testmöglichkeiten

Zum Testen unter Python gibt es mehrere Module, die das Testen unterstützen. Zum einen das Modul `doctest`, welches als interaktive Dokumentation Testmuster bereitstellt. Zum anderen das Modul `unittest`, welches den Unitests unter Java mit JUnit sehr ähnelt. Beide Module ermöglichen Regressionstests.

4.1.1 doctest

Das Modul `doctest` ermöglicht es, Tests parallel mit dem Programmcode in die selbe Datei zu schreiben. Dies ist sowohl innerhalb von Funktionen als auch außerhalb möglich. Hierbei werden die Tests in Kommentarblöcken ("""") durch die Zeichenfolge `>>>` eingeleitet. Danach wird die Funktion, die getestet werden soll, mit den gewünschten Testparametern aufgerufen. Diesem folgt dann entweder das Ergebnis oder eine Fehlerbehandlung im Falle einer zu prüfenden Exception.

Das folgende Beispiel zeigt, wie `doctests` in der Praxis Anwendung finden. Die Tests werden, sobald der Programmcode ausgeführt wird, ebenfalls ausgeführt. Dadurch, dass die Tests bei dem zu testenden Code stehen, wird ein direkter Bezug zwischen beiden hergestellt. Sollte beim Ausführen kein Test fehlschlagen, kommt es zu keiner Ausgabe durch den Interpreter. Ist trotzdem eine Ausgabe erwünscht, ist diese mit dem Parameter `-v` (verbose) aktivierbar.

```
# simpleDocTest.py
```

Beispiel

```
# Einfaches Beispiel eines DocTests in Python mit Ausgabe

"""
Outside of a function
>>> area(5, 5)
25
"""

def area(x, y):
    """Inside of a function
    Return the area of an rectangle.

    >>> area(1, 2)
    2

    >>> area(0, 2)
    Traceback (most recent call last):
    ...
    ValueError: x must be > 0

    >>> area(2, 0)
    Traceback (most recent call last):
    ...
    ValueError: y must be > 0
"""

    if not x > 0:
        raise ValueError("x must be > 0")
    if not y > 0:
        raise ValueError("y must be > 0")
    return x * y

if __name__ == "__main__":
    import doctest
```

Im nachfolgenden Beispiel ist die Ausgabe der Tests aus dem oben gezeigten Codebeispiel zu sehen. Der Ablauf der einzelnen Test ist jedes Mal gleich. Zuerst wird der Test mit den Versuchsparametern ausgegeben und im Anschluss der Erwartungswert gezeigt. Sollte der Rückgabewert der Funktion dem Erwartungswert entsprechen, wird der Test mit OK beendet. Gleicher

gilt beim Testen auf Exceptions. Zum Schluss werden die Ergebnisse in einer Auflistung zusammengefasst und nach Zugehörigkeit gruppiert. Hierbei handelt es sich um einen freien Testfall sowie drei Testfälle innerhalb der Funktion. Danach gibt es noch eine weitere Zusammenfassung, die die Tests nach Erfolg und Misserfolg gruppiert. Dies soll dem Anwender ermöglichen, alle Tests mit einem Blick zu erfassen.

```
# simpleDocTest.py
# Einfaches Beispiel eines DocTests in Python mit Ausgabe
"""
Trying:
    area(5, 5)
Expecting:
    25
ok
Trying:
    area(1, 2)
Expecting:
    2
ok
Trying:
    area(0, 2)
Expecting:
    Traceback (most recent call last):
    ...
    ValueError: x must be > 0
ok
Trying:
    area(2, 0)
Expecting:
    Traceback (most recent call last):
    ...
    ValueError: y must be > 0
ok
2 items passed all tests:
  1 tests in __main__
  3 tests in __main__.area
4 tests in 2 items.
4 passed and 0 failed.
```

Um den Fehlerfall zu betrachten, wurde die Rückgabe der Funktion von $x * y$ auf $x + y$ geändert. Dies sorgte sofort für zwei Fehler beim Start

des Programms. Die einzelnen Fehler werden getrennt dargestellt. Im Unterschied zum Erfolgsfall wird hier die Stelle angegeben, an der der Test steht. Zusätzlich dazu wird der zurückgegebene Wert angezeigt. Am Ende erfolgt wieder eine Zusammenfassung.

```
# simpleDocTest.py
# Einfaches Beispiel eines DocTests in Python mit Ausgabe
"""

*****
File "simpleDocTest.py", line 6, in __main__
Failed example:
    area(5, 5)
Expected:
    25
Got:
    10
*****
File "simpleDocTest.py", line 14, in __main__.area
Failed example:
    area(1, 2)
Expected:
    2
Got:
    3
*****
2 items had failures:
  1 of  1 in __main__
  1 of  3 in __main__.area
```

Strukturierung Über das Modul `doctest` ist es einerseits möglich, Tests direkt an den Programmcode zu hängen und über den Docstring der Funktion auszuführen. Diese Tests werden am Ende eines Moduls mit `doctest.testmod()` aufgerufen. Andererseits ist es möglich, über `doctest.testfile()` separate Testdateien als Tests auszuführen. Es ist also möglich, Tests zu einer Modul oder einer Klasse außerhalb der Klasse zu definieren. Damit der Python Interpreter weiß, welches Modul hierbei getestet werden soll, muss dies vorher über eine Importanweisung bekannt gemacht werden. Da sich der Befehl in einem Docstring befindet, muss dieser wie die Funktionsaufrufe über `>>>` eingeleitet werden. Bei dieser Art der Strukturierung ist es darüber hinaus auch möglich, die Tests unabhängig vom Programmcode ablaufen zu lassen. Hierfür muss, wie im folgenden Listing zu sehen, das Modul `doctest` angegeben

und die Testdatei aufgerufen werden. Das Modul erkennt automatisch, dass es in diesem Fall `doctest.testfile()` nutzen muss.

```
python -m doctest exampleTest.txt
```

Die Textdatei für ausgelagerte Testfälle kann die wie folgt aussehen:

```
# doctest.txt
# Example for DocTest as external testdocument

# Importanweisung
>>> from simpleDocTest import area

>>> area(1, 2)
2

>>> area(0, 2)
Traceback (most recent call last):
...
ValueError: x must be > 0

>>> area(2, 0)
Traceback (most recent call last):
...
ValueError: y must be > 0
```

Tipp:

Grundsätzlich ist die Menge des Texts, der zusätzlich zum eigentlichen Test in einer ausgelagerten `doctest`-Datei steht, beliebig. Zu beachten ist lediglich die Importanweisung sowie die Notation mit `>>>` zum Aufrufen der Funktion und deren Ausgabe im Anschluss.

4.1.2 unittest

Das Modul `unittest` ermöglicht das Testen in separaten Klassen. Es wurde in Anlehnung an JUnit erstellt. Ziel ist es, dem Programmierer zu ermöglichen, kleine wiederholbare Tests zu schreiben. Mit diesen Testfällen lässt sich Programmcode auf Integrations- und Operationsebene testen.

Beispieltest

Zum Einstieg in das Thema Unittest, zunächst ein kleines Beispiel einer Testklasse.

```
# simpleTest.py
# einfacher Test in Python

import unittest

class SimpleTest(unittest.TestCase):

    def test_add(self):
        self.assertEqual(5 + 5, 10)

    def test_sub(self):
        self.assertTrue(15 - 5 == 10)

    def test_neg(self):
        self.assertFalse(5 - 5 == 9)

    def test_exception(self):
        with self.assertRaises(TypeError):
            3 - "hello"

if __name__ == '__main__':
    unittest.main()
```

Um einen Test zu erstellen, muss die Klasse, in der die Testfälle definiert werden, von der Klasse `unittest.TestCase` ableiten. Hierfür muss zuvor das Modul `unittest` importiert werden. Anschließend können die einzelnen Testmethoden einleitend mit der Bezeichnung *test* definiert werden. Die einzelnen Testfälle werden parallel und ohne bestimmte Reihenfolge abgearbeitet. Zusätzlich zu den einzelnen Testfällen existieren noch zwei weitere Methoden zu Ablaufsteuerung. Die Methode `setUp()` sowie die Methode `tearDown()`. Diese beide Methoden erlauben das Ausführen von Code vor und nach jeder Testmethode. Dies ermöglicht es, bestimmte Testvoraussetzungen vor jedem Test zu schaffen.

Ausführen von Testzusammenstellungen

Die erstellten Tests können nicht ohne Weiteres auf der Konsole ausgeführt werden. Um das Testen zu starten, ist es notwendig, Python über den Parameter `-m unittest` aufzurufen. Dies ist erforderlich, um das Modul `unittest` im Skriptmodus zu starten. Im Anschluss folgt das zu testende Objekt. Dieses kann entweder mehrere Module, eine Klasse oder eine einzelne Methode umfassen. Die Pythondatei selbst ist ebenfalls ein mögliches Testobjekt. Hierbei werden nur die Testfälle in der Datei ausgeführt.

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestCase.test_method
python -m unittest tests/test_something.py
```

Grundsätzlich werden nach Ausführen des Befehls nur fehlerhafte Testfälle angezeigt. Um eine ausführliche Ausgabe zu forcieren, wird der Parameter `-v` benötigt. Dieser Parameter steht für `verbose` und sorgt dafür, dass erfolgreiche Testfälle ebenfalls eine Ausgabe produzieren. Weitere Parameter hierzu können der Python Dokumentation [?] entnommen werden. Alternativ kann durch den Parameter `-h` eine Liste der möglichen Parameter auf der Konsole ausgegeben werden.



Übungsaufgaben

Aufgabe 4.1.1

Erstellen sie eine Testklasse mit `doctest`, die die folgende Funktionen der Methode `testMod(int, int)` testet.

- Rückgabe des richtigen Restwertes beim Teilen.
- Beim Teilen durch 0 muss eine `Invalid Argument Exception` geworfen werden.

Aufgabe 4.1.2

Erstellen sie eine Testklasse mit `unittest`, die die folgende Funktionen der Methode `testDiv(int, int)` testet.

- Rückgabe des richtigen Wertes beim Teilen.
- Beim Teilen durch 0 muss eine `Invalid Argument Exception` geworfen werden.

Zusammenfassung

In diesem Kapitel wurde das Thema Testen grundlegend mit dem Modulen `doctest` sowie `unittest` gezeigt. Nach Abschluss der zugehörigen Übungen sollte es dem Leser möglich sein, rudimentäre Tests über beide Module zu schreiben. Dies sollte die testgetriebene Entwicklung weiteren Codes ermöglichen. Neben den hier behandelten grundlegenden Themen gibt es noch weiterführendes Material, welches der Python Dokumentation, entnommen werden kann.

Kapitel 5

Benutzeroberflächen

In diesem Kapitel gehen wir auf das Erstellen von grafischen Benutzeroberflächen mithilfe von Tkinter in Python ein, welches unter macOS und Windows zum Lieferumfang von Python gehört. Neben dem hier behandelten GUI-Toolkit existieren dennoch weitere, unter anderem WxPython, PyQt und PySide, PyGTK, Kivy sowie PyFLTK.

5.1 Tkinter

Tk ist ein freies, plattformübergreifendes GUI-Toolkit von Scriptics (früher von Sun Labs entwickelt) zur Programmierung von grafischen Benutzeroberflächen (GUIs). Ursprünglich für die Programmiersprache Tcl (Tool command language) entwickelt, existieren heute Anbindungen an diverse Programmiersprachen. Unter vielen dynamischen Sprachen ist Tk das Standard-GUI und kann umfangreiche, ab dem Release 8.0 mit nativem Look-and-Feel versehene Anwendungen erstellen, die unverändert unter Windows, macOS und Linux laufen.

Tkinter ist eine Sprachanbindung für das am häufigsten verwendete GUI-Toolkit Tk für die Programmiersprache Python. Der Name steht als Abkürzung für Tk interface. Tkinter war das erste GUI-Toolkit für Python, weshalb es inzwischen auf macOS und Windows zum Lieferumfang von Python gehört.

Tkinter besteht aus einer Reihe von Modulen. Das Tk interface wird von einem binären Erweiterungsmodul namens `_tkinter` bereitgestellt. Dieses Modul enthält die Low-Level-Schnittstelle zu Tk und sollte niemals direkt

von Anwendungsprogrammierern verwendet werden. Es handelt sich in der Regel um eine Shared Library (oder DLL), kann aber in einigen Fällen statisch mit dem Python-Interpreter verknüpft sein [?].

5.2 Einbindung

Wir beginnen mit dem Import des Tkinter-Moduls¹, welches alle Klassen und Funktionen enthält, die für die Arbeit mit dem Tk-Toolkit erforderlich sind.

```
# gui/GUI_Imports.py
# Generelles Einbinden von Tkinter

from tkinter import *           # Standard Tkinter Klassen
```

5.3 Hello World mit grafischer Benutzeroberfläche

Um Tkinter zu initialisieren, muss ein Tk Wurzel-Element erstellt werden. Dabei handelt es sich um ein gewöhnliches Fenster mit einer Titelleiste und anderer Dekoration, die vom Betriebssystem bereitgestellt werden. Generell sollte nur ein Wurzel-Element für jedes Programm erstellt werden, das auf höchster Ebene vor allen anderen Widget-Elementen steht.

Im folgenden Beispiel wird dazu ein Wurzel-Element mit dem Namen `root` initialisiert.

```
# gui/GUI_HelloWorld.py
# "Hello World" Ausgabe in einem Fenster

from tkinter import *

root = Tk()
root.title('tkinter Example - Hello World')
root.minsize(300,300)

label_1 = Label(root, text='Hello World')
label_1.grid()
```

¹Ab Python 3.X.X wird Tkinter nach der Import-Anweisung klein geschrieben.

```
root.mainloop()
```

Damit das Fenster mit Inhalt gefüllt werden kann, müssen dem Wurzel-Element weitere Widgets hinzugefügt werden. Dazu wurde ein Label mit dem Namen `label_1` angelegt. Die Parameterliste des `Label`-Widgets nimmt als erstes Argument den Namen des Eltern- bzw. Wurzel-Elements und als zweites Argument den Content bzw. Inhalt des Widgets entgegen.

Als nächstes benötigt Tkinter die Anweisung, wie der Inhalt das Fenster angezeigt werden soll. Dazu muss eine der Layout-Methoden, in diesem Fall `label_1.grid()`, aufgerufen werden. Das Fenster wird erst sichtbar, sobald die Ereignisschleife `mainloop()` betreten wird. Anschließend kann das Programm wie bereits bekannt ausgeführt werden.

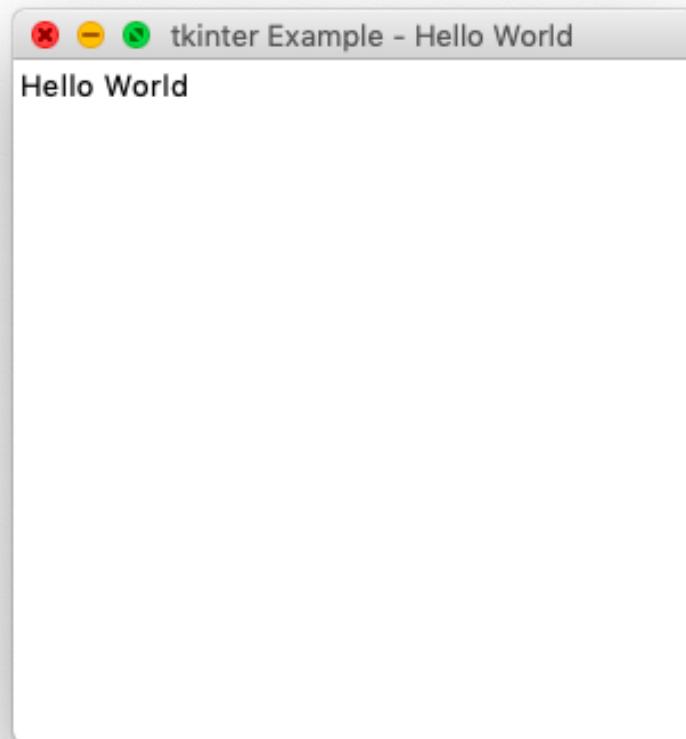


Abbildung 5.1: Hello World Beispiel mit grafischer Benutzeroberfläche

Das Programm wird nun solange ausgeführt und bleibt in der Ereignisschleife, bis das Fenster geschlossen wird. In dieser Schleife werden nicht nur Events wie Nutzereingaben (z.B. Mausklicks oder Tastatureingaben) verarbeitet, sondern auch die des Fenstersystems (z.B. Redraw-Ereignisse und Fenster-Konfigurationsmeldungen) und Ereignisse von Tkinter selbst.



Übungsaufgaben

Aufgabe 5.3.1

Erweitern Sie ihr Programm aus 1.5.2 durch Verwendung grafischer Benutzeroberfläche.

Importieren Sie dazu Tkinter und geben `Hello Python-World!` in einem Fenster aus.

5.4 Die Layout-Manager

In diesem Kapitel werden die Layout- oder Gemeometrie-Manager von Tkinter behandelt. Grundsätzlich stehen drei verschiedene Layout-Manager zur Verfügung:

- Pack
- Grid
- Place

Layout-Manager dienen in erster Linie dazu, Widgets in einem Wurzel-Element zu registrieren, anzuordnen und darzustellen. Besonders die Anordnung durch Angabe von Position und Größe eines Widgets wird durch die Layout-Manager stark vereinfacht.

Einem Wurzel-Element sollte immer nur ein Layout-Manager angehängt werden.

5.4.1 Pack

Der Pack-Manager ist der am einfachsten zu verwendende Layout-Manager. Hier werden Widgets in Zeilen oder Spalten (vertikal oder horizontal) 'gepackt' und durch Optionen wie `fill` oder `expand` gesteuert.

Im Vergleich zu dem sehr ähnlichen Grid-Manager ist der Pack-Manager etwas eingeschränkt, aber in einigen wenigen Situationen sinnvoller zu nutzen. Speziell wenn einfache Widgets übereinander oder nebeneinander angeordnet werden oder Inhalte eines Widgets das gesamte übergeordnete Widget ausfüllen sollen, wird der Pack-Manager bevorzugt.

Das folgende Beispiel soll die Effekte des Pack-Managers verdeutlichen:

```
# gui/GUI_PackExample.py
# Listbox Beispiel im Pack-Manager

from tkinter import *

root = Tk()

listbox = Listbox(root)
listbox.pack()

for i in range(20):
    listbox.insert(END, str(i))

mainloop()
```

Standardmäßig wird die Größe der Listbox so gewählt, dass zehn Elemente angezeigt werden können. Im obigen Quelltext werden der Listbox allerdings doppelt so viele Elemente übergeben. Versucht der User nun das Wurzel-Element, sprich das Fenster, zu vergößern, um alle Elemente anzuzeigen, erzeugt Tkinter rund um das Listbox-Widget einen Abstand zum Fenster. Um das Widget der gesamten zur Verfügung stehenden Fläche anzupassen, müssen die Optionen `fill` und `expand` des Pack-Managers angesprochen werden.

```
# gui/GUI_PackExampleII.py
# Listbox Beispiel im Pack-Manager

from tkinter import *
```

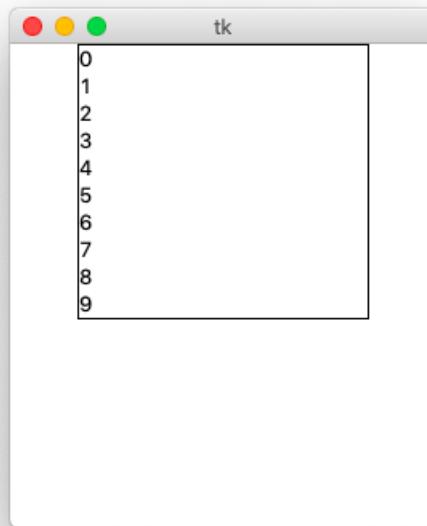


Abbildung 5.2: Listbox Widget mit einem Pack-Manager angeordnet

```
root = Tk()

listbox = Listbox(root)
listbox.pack(fill=BOTH, expand=1)

for i in range(20):
    listbox.insert(END, str(i))

mainloop()
```

Die `fill`-Option teilt dem Pack-Manager mit, dass der gesamte zur Verfügung stehende Raum durch das Widget ausgefüllt werden soll. Der dahinter stehende Wert regelt, wie der Raum gefüllt wird. `BOTH` bedeutet, dass das Widget sowohl horizontal als auch vertikal expandieren soll. Alternativ kann der Raum mit der Angabe `x` nur horizontal und mit der Angabe `y` nur vertikal ausgefüllt werden.

Darüber hinaus können Widgets mithilfe der Layout-Manager angeordnet werden. Der Pack-Manager richtet Widgets, ohne weitere Angaben von Optionen, vertikal, sprich in Spalten aus.

```
# gui/GUI_PackExampleIII.py
```

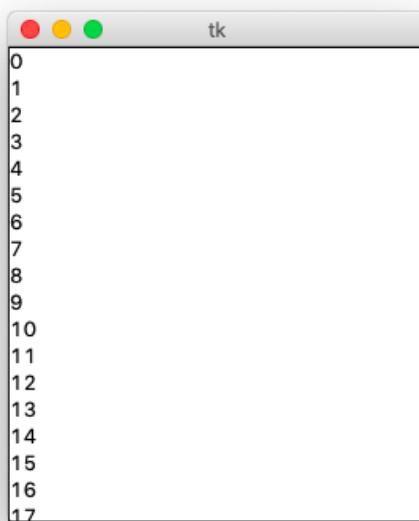


Abbildung 5.3: Listbox Widget mit einem Pack-Manager angeordnet unter Angabe von `fill` und `expand`

```
# Labels anordnen - Beispiel im Pack-Manager

from tkinter import *

root = Tk()

labelRed = Label(root, text="Red", bg="red", fg="white")
labelRed.pack()
labelGreen = Label(root, text="Green", bg="green", fg="black")
labelGreen.pack()
labelBlue = Label(root, text="Blue", bg="blue", fg="white")
labelBlue.pack()

mainloop()
```

Über die Option `fill=x` (horizontal) kann die Breite der Labels dem Eltern-Element angepasst werden.

```
# gui/GUI_PackExampleIV.py
# Labels anordnen - Beispiel im Pack-Manager

from tkinter import *
```

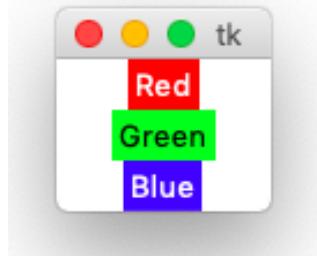


Abbildung 5.4: Mit Pack-Manager vertikal angeordnete Label-Widgets

```
root = Tk()

labelRed = Label(root, text="Red", bg="red", fg="white")
labelRed.pack(fill=X)
labelGreen = Label(root, text="Green", bg="green", fg="black")
labelGreen.pack(fill=X)
labelBlue = Label(root, text="Blue", bg="blue", fg="white")
labelBlue.pack(fill=X)

mainloop()
```

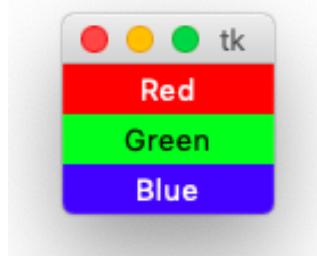


Abbildung 5.5: Mit Pack-Manager vertikal angeordnete Label-Widgets unter Angabe von fill

Neben der vertikalen Ausrichtung der Labels ist auch eine horizontale möglich, dazu steht die `side`-Option zur Verfügung. Mit `side=LEFT` werden die Widgets von links beginnend angeordnet. Weitere Parameter sind `side=TOP` (default), `side=BOTTOM` und `side=RIGHT`.

5.4.2 Grid

Der Grid-Manager ist der am flexibelsten einzusetzende Layout-Manager von Tkinter. Besonders komfortabel ist der Einsatz bei dem Gestalten von Dialogfeldern. Die Handhabung ist denkbar einfach. Das folgende Schaubild erläutert die Aufteilung eines Gitterasters:

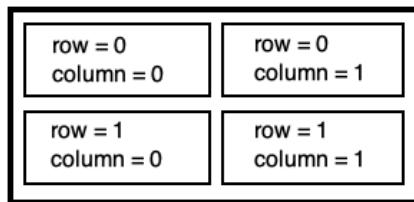


Abbildung 5.6: Gitterraster mit zwei Spalten und Reihen

Nach dem Erzeugen eines Widget-Elements kann es über den Methoden-Aufruf des Grid-Managers, unter Angabe von Zeilen und Spalten ausgerichtet werden. Dabei muss weder Höhe noch Breite der entsprechenden Zelle angegeben werden, der Grid-Manager passt diese dem Content des Widgets an.

```
# gui/GUI_PackExampleIV.py
# Labels und Inputs anordnen - Beispiel im Grid-Manager

from tkinter import *

root = Tk()

labelRed = Label(root, text="Label One", bg="red",
                 fg="white").grid(row=0)
labelGreen = Label(root, text="Label Two", bg="green",
                   fg="black").grid(row=1)

e1 = Entry(root)
e2 = Entry(root)

e1.grid(row=0, column=1)
e2.grid(row=1, column=1)

mainloop()
```

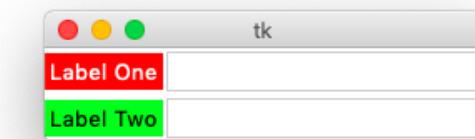


Abbildung 5.7: Mit Grid-Manager angeordnete Label- und Input-Widgts

Leere Spalten und Zeilen werden von dem Layout-Manager ignoriert, das Ergebnis wäre das selbe, wenn die Angabe anstelle von `grid(row=0)` und `grid(row=1), grid(row=5)` und `grid(row=10)` lauten würde.

Der Inhalt der Zellen wird ohne weitere Angaben mittig angeordnet, was über die `sticky`-Option geändert werden kann. Im Gegensatz zu der `side`-Option des Pack-Managers, nimmt `sticky` Himmelsrichtungen, also N, E, S und W als Angabe entgegen. Mit `Label(...).grid(row=0, sticky=W)` beginnt der Text des Label-Widgts am linken Zellenrand (Westen). Auch Kombinationen wie beispielsweise 'unten links' (`sticky=W+S`) sind möglich.

5.4.3 Place

Der dritte Layout-Manager von Tkinter ist der Place-Manager. Hier kann die Position und Größe eines Fensters explizit festgelegt werden, entweder absolut oder relativ zu einem anderen Fenster. In der Regel werden Pack- oder Grid-Manager für die Gestaltung herkömmlicher Dialog- und Fensteroberflächen verwendet, in einigen wenigen Fällen ist jedoch der Place-Manager die bessere Wahl. Beispielsweise bei der Anordnung von Steuerelementen in Popup-Dialogen kommt der Place-Manager zum Einsatz.

Der Zugriff erfolgt wie bei den vorherigen Layout-Managern über den Methodenaufruf unter Angabe von X und Y - Koordinaten. Alternativ können mit der `anchor`-Option ähnlich der `side`-Option des Pack-Managers, die Widgets über Kompass-Richtungen ausgerichtet werden. Default ist NW (obere linke Ecke des Eltern-Elements).

5.4.4 Zusammenfassung

Zusammenfassend kann gesagt werden:

- Der Pack-Manger organisiert Widget-Elemente in Blocks. Anschließend werden diese im Eltern-Element platziert.
- Der Grid-Manger platziert Widget-Elemente in einer Tabellenartigen Struktur im Eltern-Element.
- Der Place-Manager platziert Widget-Elemente in spezifischer Position im Eltern-Element.



Übungsaufgaben

Aufgabe 5.4.1

Erweitern Sie ihr Programm aus Aufgabe 5.3.1. Legen Sie dazu drei weitere Label-Widgets an und platzieren Sie diese mithilfe des Grid-Managers. Dabei soll das Label mit `Hello Python-World!` in der ersten Spalte, erste Reihe angezeigt werden. Die übrigen Labels sollen so verteilt werden, dass eine 2X2 Matrix aus Labels entsteht.

5.5 Widgets

Unter den Widgets werden in Tkinter alle grafischen Interaktionselemente wie Buttons, Labels, Listboxes sowie Canvas etc. verstanden. Tkinter bietet hierfür bereits vorgefertigte Klassen, aus denen diese Widgets erzeugt werden können. Im Rahmen dieses Kapitels wird eine Anwendung erstellt, die den Umgang mit einigen Widgets verdeutlichen soll und fortlaufend dahingehend erweitert wird.

5.5.1 Frame

Frames fungieren, ähnlich wie die sogenannten `divs` in HTML, als Container um verschiedene Widgets oder weitere Frames in einem bestimmten Bereich des Fensters zu gruppieren. Dabei wird ein Rechteck erstellt und dem Eltern-Element über den Layout-Manager hinzugefügt. In folgendem Beispiel werden drei Frames erzeugt, die sich untereinander befinden und unterschiedliche Hintergrundfarben besitzen. Um zu verdeutlichen, dass weitere Widgets jetzt den jeweiligen Frame-Bereichen zugeordnet werden können, wird in je-

dem Frame ein Label hinzugefügt. Somit kann die Platzierung von Widgets besser kontrolliert werden.

```
# gui/GUI_FrameExample.py
# Beispiel eines Fensters mit Frames

from tkinter import *

root = Tk()
root.minsize(400, 400)

headFrame = Frame(root, bg="red")
headFrame.pack(side=TOP, fill="both", expand=True)
centerFrame = Frame(root, bg="blue")
centerFrame.pack(fill="both", expand=True)
bottomFrame = Frame(root, bg="green")
bottomFrame.pack(side=BOTTOM, fill="both", expand=True)

label_head = Label(headFrame, text="HEAD")
label_head.pack()
label_center = Label(centerFrame, text="CENTER")
label_center.pack()
label_bottom = Label(bottomFrame, text="BOTTOM")
label_bottom.pack()

root.mainloop()
```

5.5.2 Label

Über Labels können im Userinterface einfache Textzeilen ausgegeben werden. Diese lassen sich weiterhin auch dynamisch verändern (z.B. bei einem Klick auf einen Button).

5.5.3 Button

Mit dem Button hat der Nutzer die Möglichkeit, mit dem Userinterface über einen Maus-Klick zu interagieren. Buttons werden aus der Klasse `Button` erzeugt, dem Fenster oder Frame bekannt gemacht und mit einem Text versehen. Zusätzlich kann dem Button über `command` noch eine Funktion zugewiesen werden, die ausgeführt wird, sobald der Button gedrückt wurde. In

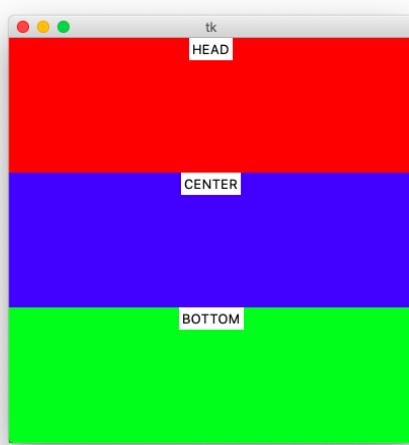


Abbildung 5.8: Frame-Widget Beispiel

folgendem Beispiel wird bei Klick auf einen Button in einem Label der Satz Hallo Python-World ausgegeben.

```
# gui/GUI_ButtonLabelExample.py
# Button der den Text eines Labels veraendert

from tkinter import *

def printHelloPythonWorld():
    label_1["text"] = "Hallo Python-World!"

root = Tk()

button_1 = Button(root, text="Click me!",
                  command=printHelloPythonWorld)
button_1.pack()

label_1 = Label(root, text="Hello World")
label_1.pack()

root.mainloop()
```

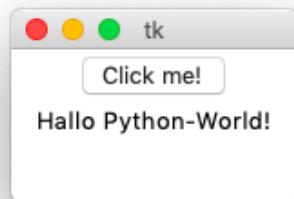


Abbildung 5.9: Button- und Label-Widget Beispiel

5.5.4 Entry

Über die sogenannten Entries lassen sich Eingaben vom Nutzer erfassen. Diese können dann im Python-Skript weiterverarbeitet werden. Das vorherige Beispiel wird nun dementsprechend erweitert, dass der Text, der bei Klick auf den Button ausgegeben wird, über `get()` vom Entry entgegen genommen wird.

```
# gui/GUI_EntryExample.py
# Einlesen und Ausgeben eines String ueber Entry

from tkinter import *

def printStringFromEntry():
    label_1["text"] = entry_1.get()

root = Tk()

entry_1 = Entry(root)
entry_1.pack()

button_1 = Button(root, text="Click me!",
                  command=printStringFromEntry)
button_1.pack()

label_1 = Label(root, text="")
label_1.pack()

root.mainloop()
```



Abbildung 5.10: Entry-Widget Beispiel

5.5.5 Listbox

Um verschiedene Elemente in einer Liste anzeigen zu können, bietet sich die Listbox an. Der Text, der vom Entry entgegengenommen wird, soll nun einer solchen Listbox hinzugefügt werden. Dazu dient die Funktion `insert()`. Der erste Parameter, der übergeben wird, bestimmt die Position in der Liste und der zweite das eigentliche Element. Zusätzlich wird der Anwendung ein zweiter Button hinzugefügt um das jeweilig angewählte Element der Listbox zu entfernen. Die Funktion `curselection()` gibt eine Liste der angewählten Elemente zurück. Mithilfe von `delete()` wird dann das erste Element der zurückgegebenen Liste aus der Listbox entfernt.

```
# gui/GUI_ListboxExample.py
# Einlesen und Hinzufügen in Listbox

from tkinter import *

def addStringToList():
    if entry_1.get() != "":
        myList.insert(END, entry_1.get())

def removeItemFromList():
    selectedItem = myList.curselection()
    if len(selectedItem):
        myList.delete(selectedItem[0])

root = Tk()

entry_1 = Entry(root)
entry_1.pack()
```

```
button_frame = Frame(root)
button_frame.pack()

button_add = Button(button_frame, text="ADD",
                     command=addStringToList)
button_add.pack(side=LEFT)

button_remove = Button(button_frame, text="REMOVE",
                      command=removeItemFromList)
button_remove.pack(side=RIGHT)

myList = Listbox(root)
myList.pack()

root.mainloop()
```

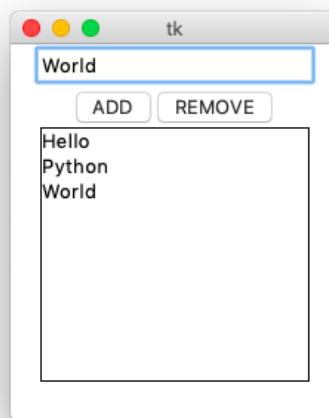


Abbildung 5.11: Listbox-Widget Beispiel

5.5.6 Colorchoosers

Das Colorpicker oder Colorchooser-Modul bietet die Möglichkeit Farben auszuwählen. Diese kann anschließend über die Methode `askcolor()` ausgelesen und Beispielsweise einem weiteren Widget übergeben werden. Es handelt sich um ein Tupel der Form `((99, 222, 170), '#63deaa')`, wobei

der erste Teil die RGB-Werte und der zweite Teil den hexadezimal Farbc ode verkörpert. Das Colorchooser-Modul von Tkinter bietet eine Schnittstelle zum nativen Farbauswahl-Widget des zugrundeliegenden Systems, weshalb sich das Widget in Optik und Funktion je nach Betriebssystem unterscheiden kann. Der Colorchooser wird über `tkinter.colorchooser` importiert.

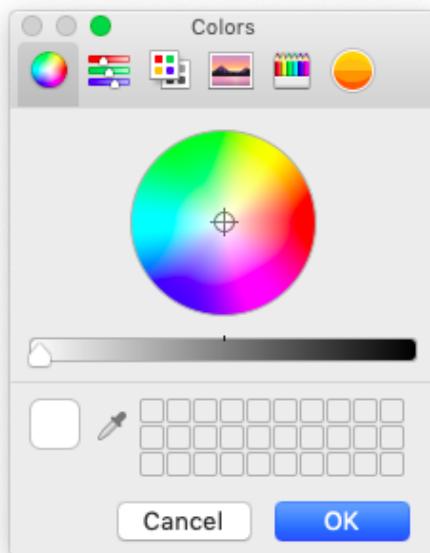


Abbildung 5.12: Colorchooser-Widget unter macOS

5.5.7 Canvas

Analog zu dem in HTML 5 eingeführten Canvas-Widget oder Canvas-Element bietet auch Tkinter ein Element zur Darstellung verschiedener grafischer Objekte. Mithilfe von Canvas können Linien, Kreise oder aufwendige Formen in einem dafür vorgesehenen Rahmen angezeigt werden. Dabei stehen mehrere Methoden zum gestalten grafischer Objekte zur Verfügung. Mit `create_line()` können Linien, mit `create_oval()` oder `create_rectangle()` fertige geometrische Formen erstellt und anschließend dem Canvas zum Zeichnen übergeben werden. `Create_polygon()` bietet sogar die Möglichkeit ganze Polygonzüge und somit uneingeschränkte, zweidimensionale Figuren zu erstellen. Dabei muss als erstes Argument eine Reihe von kartesischen Koordinaten übergeben werden. Im folgenden Beispiel wird ein blaues Dreieck erstellt und anschließend gezeichnet:

```
canvas.create_polygon([120, 120, 220, 220, 220, 120], blue)
```

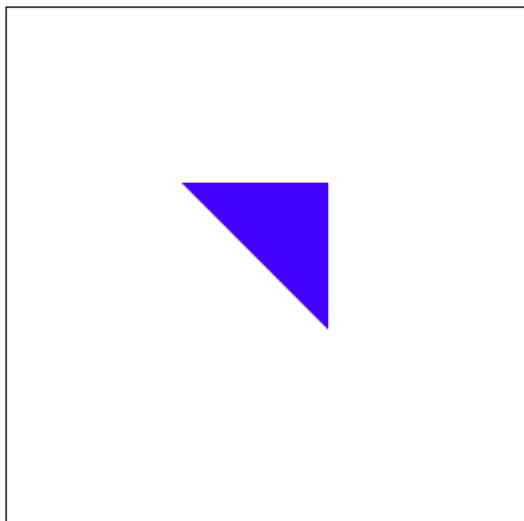


Abbildung 5.13: Ein Mithilfe der Create Polygon Canvas-Methode erstelltes Dreieck

Um ein gezeichnetes Objekt zu löschen bzw. die Zeichenfläche zu reinigen steht die Methode `canvas.delete(ALL)` zur Verfügung. Der Übergabe Parameter besagt dabei, alle Objekte die dem Widget übergeben wurden sollen gelöscht werden.



Übungsaufgaben

Aufgabe 5.5.1

Um den Umgang mit Tkinter zu üben, werden wir eine einfache Anwendung zum Anzeigen von geometrischen Figuren programmieren. Diese Anwendung wird unter anderem aus einem Listen-Widget, mehrerer Buttons, einem Canvas-Widget, sowie einem Popup-Fenster mit Labels und Eingabe-Widgets bestehen. Es ist sinnvoll, die vorherigen Aufgaben bearbeitet und ein grundlegendes Verständnis für Klassen und Funktionen in Python zu haben. Das Listen-Widget wird die Namen der geometrischen Figuren enthalten. Über Buttons können neue Figuren angelegt oder bestehende gelöscht werden. Ist eine Figur aus der Liste ausgewählt, kann diese über einen Button auf das Canvas-Widget gezeichnet werden. Ein weiterer Button löscht die Anzeigefläche des Canvas-Widget. Das folgende Bild beschreibt einen möglichen Aufbau der Anwendung:

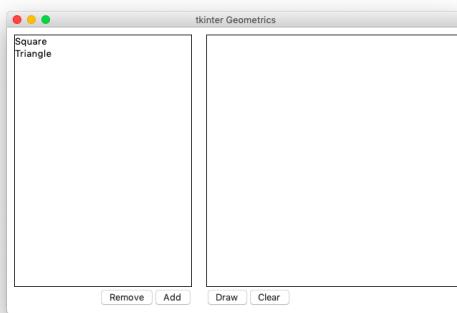


Abbildung 5.14: Auf der linken Seite befindet sich das Listen-Widget, sowie zwei Buttons 'Add' und 'Remove'. Auf der rechten Seite das Canvas-Widget und zwei Buttons 'Draw' und 'Clear'.

Legen Sie als Erstes zwei Frame-Widgets mit dem Namen LeftFrame und RightFrame an. Übergeben Sie diese anschließend dem Fenster, so dass es in zwei gleichgroße Sektionen unterteilt wird.

Aufgabe 5.5.2

Übergeben Sie dem linken Frame nun mithilfe des Pack-Managers eine Listbox und zwei Buttons mit dem Namen 'Add' und 'Remove'.

Aufgabe 5.5.3

Wiederholen Sie diesen Vorgang nun für das rechte Frame. Übergeben Sie diesem ein Canvas-Widget sowie zwei Button mit den Namen 'Draw' und 'Clear'.

Aufgabe 5.5.4

Schreiben Sie eine kleine Klasse, deren instanziertes Objekt eine geometrische Figur verkörpert und legen Sie anschließend statisch zwei geometrische Figuren an.

In unserem Fall genügt es, der Klasse die Attribute Namen, Points und Color zu geben, um eine geometrische Figur zu verkörpern. Points besteht dabei aus einer Liste mit X- und Y-Koordinaten. Um ein Quadrat anzulegen, wird also eine Liste mit Vier punkten benötigt, wobei jeder Punkt aus zwei Werten besteht. Beispiel: [0, 0, 100, 100, 100, 100, 100, 0] erzeugt ein Quadrat mit den Punkten (0/0), (0/100), (100/100) und (100,0).

Aufgabe 5.5.5

Übergeben Sie nun dem Listen-Widget mithilfe der `insert()`-Methode die Namen der vorher erzeugten Figuren. `Insert()` nimmt als erstes Argument die Position in der Liste und als zweites Argument den Wert entgegen.

Aufgabe 5.5.6

Schreiben Sie eine Funktion, die die Punkte der ausgewählten Form an das Canvas-Widget übergibt und diese zeichnet. Nutzen Sie hierzu die Methode `create_polygon()` des Canvas-Widget. `create_polygon()` erwartet als erstes Argument eine Liste mit Koordinaten. Übergeben Sie als zweites Argument eine Farbe mit `fill=`. Beispiel: `fill="yellow"`. Übergeben Sie die Funktion anschließend dem Zeichnen-Button mithilfe des `command`-Attributs von Button.

Aufgabe 5.5.7

Erweitern Sie die Anwendung durch Funktionalität der restlichen Buttons 'Add', 'Remove' und 'Clear'.

Wird der 'Add'-Button gedrückt, soll sich ein Popup-Fenster öffnen, in welchem eine neue geometrische Figur unter Angabe der Attribute angelegt und der Liste hinzugefügt werden kann. Ein Popup-Fenster wird gleich dem jetzigen Fenster angelegt und mit einer `mainloop()`-Eventschleife sichtbar gemacht. Wählen Sie anschließend geeignete Eingabefelder aus, um eine neue geometrische Figur zu erstellen.

Tipp:

Damit nicht jeder Punkt der Figur einzeln angegeben werden muss, kann die Klasse `random` eingebunden werden. Mit `random.randint(0, 100)` wird eine zufällige ganze Zahl zwischen 0 und 100 ausgegeben.

Kapitel 6

Python Bibliotheken

6.1 NumPy

NumPy ist eine Python-Bibliothek für wissenschaftliches Rechnen. Sie beinhaltet laut der Dokumentation unter anderem Folgendes [?]:

- mächtige n -dimensionale Array-Objekte
- Werkzeuge zur Integration von C und Fortran
- Funktionen zur linearen Algebra, Fouriertransformation, Erzeugung von Zufallszahlen

Um NumPy zu installieren, kann der Befehl `pip install numpy` verwendet werden.

6.1.1 Arrays

Der Array-Datentyp von NumPy heißt `numpy.ndarray`. Anders als der Standarddatentyp für Listen (`list`) unterstützt der Datentyp `numpy.ndarray` numerische Operatoren. Der NumPy-eigene Datentyp ermöglicht es, Arrays direkt über den `+`-Operator elementweise zu addieren. Eine Addition mit einer einzelnen Zahl vom Typ `int` oder `float` betrifft alle Elemente im Array.

So kann etwa jeder Wert in einem Array mit den folgenden Anweisungen um drei erhöht werden:

```
import numpy as np  
a = np.array([1, 2, 3])
```

```
a + 3 # [4 5 6]
```

NumPy wird hierbei mit dem Namen `np` importiert. Damit folgt dieses Tutorial der Konvention aus der Dokumentation von NumPy.

Subtraktion, Multiplikation, Division, Ganzzahldivision und Potenzieren funktionieren analog:¹

```
a = np.array([1,2,3])
a - 3 # [-2 -1  0]
a * 3 # [3 6 9]
a / 3 # [0.33333333 0.66666667 1.]
a // 3 # [0 0 1]
a ** 3 # [ 1   8 27]
```

Zwei Arrays gleicher Länge können elementweise miteinander verrechnet werden:

```
a = np.array([1,2,3])
b = np.array([4,5,6])
a + b # [5 7 9]
a - b # [-3 -3 -3]
a * b # [ 4 10 18]
a / b # [0.25 0.4 0.5 ]
a ** b # [ 1 32 729]
a // b # [0 0 0]
```

Um ein NumPy-Array zu erzeugen, wird ein `list`-Objekt an die Funktion `np.array()` übergeben. Dabei werden alle Elemente im `list`-Objekt in einem Datentyp von NumPy konvertiert. Um den Datentyp eines Arrays herauszufinden, wird `.dtype.name` genutzt. Anders als bei `list` müssen sämtliche Elemente eines Arrays den gleichen Typ haben.

```
a = np.array([1,2,3])
a.dtype.name # 'int64'
b = np.array([1.4,2.5,3.6])
b.dtype.name # 'float64'
```

Wenn die übergebene `list` Elemente vom Typ `int` und von Typ `float` gemischt enthält, konvertiert NumPy in einen Fließkommatyp. Wie viele Bit für einen Integer beziehungsweise ein Float zur Verfügung stehen, ist von

¹Der Import von `numpy` wird der Übersichtlichkeit halber nachfolgend ausgelassen.

der Prozessorarchitektur abhängig. Diese beträgt aber bei aktuellen Architekturen in der Regel 64 Bit.

6.1.2 Konstanten und Funktionen

Es stehen für die mathematische Anwendungen auch Konstanten zur Verfügung, darunter die Folgenden mit den entsprechenden Werten und Präzisionen mit float64:

```
>>> np.pi  
3.141592653589793  
>>> np.e  
2.718281828459045  
>>> np.euler_gamma  
0.5772156649015329  
>>> np.PINF  
inf  
>>> np.NINF  
-inf  
>>> np.NAN  
nan  
>>> np.PZERO  
0.0  
>>> np.NZERO  
-0.0  
>>> np.NAN  
nan
```

np.NZERO steht für die negative Darstellung der Null bei Fließkommazahlen, np.PZERO für die positive Darstellung.

NumPy unterstützt eine Vielzahl an mathematischen Funktionen, darunter unter anderem Folgende:

- trigonometrische Funktionen
- Rundungsfunktionen
- Summationsfunktionen
- Multiplikationsfunktionen

■ Funktionen zur Behandlung komplexer Zahlen

Die grundlegenden trigonometrischen Funktionen werden elementweise auf das Array angewendet:

```
>>> a = np.array([0, np.pi/6, np.pi/4, np.pi/3, np.pi/2])
>>> np.sin(a)
[0.           0.5           0.70710678  0.8660254   1.          ]
>>> np.cos(a)
[1.00000000e+00  8.66025404e-01  7.07106781e-01  5.00000000e-01
6.12323400e-17]
>> np.tan(a)
[0.00000000e+00,  5.77350269e-01,  1.00000000e+00,
1.73205081e+00,  1.63312394e+16]
```

Umrechnung von Radians in Grad:

```
>>> a = np.array([0, np.pi/6, np.pi/4, np.pi/3, np.pi/2])
>>> np.degrees(a)
[ 0., 30., 45., 60., 90.]
```

Umrechnung von Grad in Radians:

```
>>> a = np.array([ 0, 30, 45, 60, 90])
>>> np.radians(a)
[0.           0.52359878  0.78539816  1.04719755  1.57079633]
```

Mit der Funktion `np.around()` können sämtliche Werte im Array auf eine bestimmte Anzahl von Stellen gerundet werden. Ohne Angabe eines zweiten Arguments wird kaufmännisch auf die nächste Ganzzahl gerundet.

```
>>> a = np.array([1.49, 1.5, 1.51])
>>> np.around(a)
[1. 2. 2.]
```

Mit dem optionalen zweiten Argument wird die Anzahl an Nachkommastellen, auf die gerundet werden soll, angegeben:

```
>>> a = np.array([1.25, 1.53, 1.99])
>>> np.around(a, 1)
[1.2, 1.5, 2. ]
```

Um alle Elemente eines Arrays aufzusummen, wird die Funktion `np.sum()` verwendet.

```
>>> a = np.array([1, 2, 3])
>>> np.sum(a)
6
```

Mit der Funktion `np.prod()` können die Elemente der Liste miteinander multipliziert werden.

```
>>> a = np.array([2, 3, 4])
>>> np.prod(a)
24
```

Sollten `nan` (not a number) im Array vorkommen können, so kann die Funktion `np.nansum()` beziehungsweise `np.nanprod()` verwendet werden. Bei der Funktion `np.nansum()` werden `nan` als 0 interpretiert.

```
>>> a = np.array([np.NAN, 1, 2, 3])
>>> np.sum(a)
nan
>>> np.nansum(a)
6.0
```

Durch die Funktion `np.nanprod()` werden `nan` als 1 interpretiert:

```
>>> a = np.array([np.NAN, 2, 3, 4])
>>> np.prod(a)
nan
>>> np.nanprod(a)
24.0
```

Addition und Multiplikation geben eine Zahl vom Typ `float64` als Ergebnis zurück. `nan` ist ein valider Fließzahlwert, daher werden die restlichen Werte ebenfalls nach `float64` konvertiert.

6.1.3 Erzeugen und Manipulieren von Arrays

Bislang haben die Beispiele in diesem Kapitel Arrays immer auf die folgende Weise erzeugt:

```
>>> a = np.array([1, 2, 3])
>>> a
[1 2 3]
```

Hierbei wird zuerst eine `list` mit konkreten Werten erzeugt und dann mittels `np.array` in ein NumPy-Array konvertiert.

Neben Listen kann `np.array` auch sämtliche anderen Sequences in Arrays umwandeln, zum Beispiel Tupel und Ranges:

```
>>> t = (1, 2, 3)
>>> np.array(t)
[1, 2, 3]
>>> r = range(1,6)
>>> np.array(r)
[1, 2, 3, 4, 5]
```

Numpy kann auch direkt ein Numpy-Array mit den gleichen Paramtern wie von `range` erzeugen. Die Funktion hierzu heißt `np.arange()`:

```
>>> np.arange(1,6)
[1, 2, 3, 4, 5]
```

Ebenso können mehrdimensionale Arrays erzeugt werden:

```
>>> a = [[1, 2], [3, 4]]
>>> np.array(a)
[[1 2]
 [3 4]]
```

Über das Attribut `.shape` kann jederzeit die Form eines Arrays abgefragt werden.

```
>>> a = [[1, 2], [3, 4]]
>>> b = np.array(a)
>>> b.shape
(2, 2)
```

Leere Arrays

Wenn die Werte des Arrays zum Zeitpunkt seiner Erzeugung noch nicht bekannt sind, kann mit der Funktion `np.empty()` ein leeres Array erzeugt werden. Welche Werte dabei initial im Array stehen, ist nicht definiert, da `np.empty()` lediglich das Array erzeugt, nicht aber dessen Werte initialisiert. Die Werte eines so erzeugten Arrays sind zufällig.

```
>>> np.empty(2)
[1.13224202e+277, 2.00000008e+000]
```

Bei den Funktionen in diesem Abschnitt kann statt einer Zahl als Länge auch eine Sequence mit den Längen der Dimensionen übergeben werden:

```
>>> np.empty([2, 3])
[[ 1.,  4.,  9.]
 [16., 25., 36.]]
```

Der Parameter bestimmt die Länge des Arrays, der Datentyp ist standardmäßig `float64`. Um ein `int64`-Array zu erzeugen, wird das optionale Argument `dtype=int` mit übergeben:

```
>>> np.empty(2, dtype=int)
[8751743591039004782, 4611686018597859880]
```

Achtung:

Diese Funktion sollte mit Vorsicht genutzt werden, da die Werte manuell gesetzt werden müssen!

Analog zu `np.empty()` kann bei den allen Funktionen zur Erzeugung von Arrays der Datentyp explizit angegeben werden:

```
>>> np.array([1, 2], dtype=float)
array([1., 2.])
```

Neben `int` und `float` kann bei Bedarf mit `np.int8`, `np.int16`, `np.int32`, `np.int64`, `np.float16`, `np.float32`, `np.float64` und `np.float128` explizit die Größe des Integer beziehungsweise Float-Wertes im Speicher bestimmt werden.

Besonders interessant ist hierbei `np.float128`, ein Float mit Quadruple Precision und 128 Bit Länge. Damit können mindestens 34 Stellen Präzision gepeichert werden.²

Arrays mit Standardwerten

Die Funktion `np.ones()` erzeugt Arrays, die mit Einsen initialisiert sind:

```
>>> np.ones(7)
[1., 1., 1., 1., 1., 1., 1.]
```

²Für die Mantisse stehen beim `np.float64` nach IEEE 754 113 Bit zur Verfügung. Das ergibt aufgrund von $\log_{10}(2^{113}) \approx 34.016$ 34 Stellen.

Mit `np.zeros()` werden Arrays mit Nullen gefüllt:

```
>>> np.zeros(7)
[0., 0., 0., 0., 0., 0., 0.]
```

Die Funktion `np.full()` füllt ein Array mit dem angegebenen Wert:

```
>>> np.full(3, np.pi)
[3.14159265, 3.14159265, 3.14159265]
```

Arrays kopieren

Um ein vorhandenes Array zu kopieren, wird die `.copy()`-Methode verwendet.

```
>>> a = np.array([1,2,3])
>>> b = a
>>> c = a.copy()
>>> print(a, b, c)
[1 2 3] [1 2 3] [1 2 3]
>>> a += 3
>>> print(a, b, c)
[4 5 6] [4 5 6] [1 2 3]
```

Mit dem Zuweisungsoperator wird lediglich eine Referenz auf das Array erzeugt, die Methode `.copy()` erzeugt eine Objektkopie.

Arrays persistieren

Um Arrays zu persistieren, können die Funktionen `np.save()` zum Speichern und `np.load()` zum Auslesen verwendet werden. Das nachfolgende Beispiel simuliert dieses Verhalten mit der Klasse `TemporaryFile` aus dem `tmpfile`-Paket. `TemporaryFile` verhält sich wie eine normale Datei, mit dem Unterschied, dass der Dateiinhalt im Arbeitsspeicher vorgehalten wird. Mit der Methode `.seek(0)` auf ein `TemporaryFile`-Objekt wird das Schließen und erneute Öffnen der Datei simuliert.

```
>>> from tempfile import TemporaryFile
>>> outfile = TemporaryFile()
>>> a = np.array([1,2,3,4])
>>> np.save(outfile, a)
>>> outfile.seek(0)
```

```
>>> np.load(outfile)
[1, 2, 3, 4]
```

Per Konvention lautet die Dateiendung so gespeicherter Arrays .npy.

Arrays mittels Funktionen berechnen

Mit der Funktion `np.fromfunction()` können die Werte mittels einer gegebenen Funktion berechnet werden. Dabei sind die Parameter der Funktion die Indizes der Position im Array. Die Dimensionen des Arrays *müssen* bei `np.fromfunction()` als Sequence übergeben werden. Wenn kein Datentyp angegeben wird, wird `np.float64` angenommen.

```
>>> f = lambda i: i ** 2
>>> np.fromfunction(f, [7], dtype=int)
array([ 0,  1,  4,  9, 16, 25, 36])
```

Die Form des Arrays ist bei `np.fromfunction()` abhängig von der verwendeten Funktion. Wenn diese skalare Werte zurückgibt, entspricht die Form den in den Paramtern angegebenen Dimensionen.

```
>>> f = lambda i, j: i + j
>>> a = np.fromfunction(f, [3, 2], dtype=int)
[[0 1]
 [1 2]
 [2 3]]
>>> a.shape
(3, 2)
```

Untenstehendes Beispiel zeigt, wie sich die Form des Arrays durch die Rückgabewerte der Funktion verändern kann.

```
>>> f = lambda i, j: np.array([i, j])
>>> a = np.fromfunction(f, [3, 2], dtype=int)
>>> a
[[[0 0]
 [1 1]
 [2 2]]

 [[0 1]
 [0 1]
 [0 1]]]
```

```
>>> a.shape
(2, 3, 2)
```

Arrays aus iterierbaren Objekten erzeugen

Um ein beliebiges iterierbares Objekt in ein eindimensionales Array zu konvertieren, bietet NumPy die Funktion `np.fromiter()` an. Der zweite Parameter gibt den Datentyp des Arrays an. Mit dem optionalen Parameter `count` wird angegeben, wie viele Elemente aus dem Objekt übernommen werden sollen. Wird `count` weggelassen, so werden alle Werte übernommen.

Achtung:

Falls das iterierbare Objekt unendlich viele Werte erzeugen kann, muss `count` angegeben werden, da `np.fromiter()` sonst niemals abbricht!

```
>>> def even_numbers():
...     i = 0
...     while True:
...         yield i
...         i += 2
...
>>> np.fromiter(even_numbers(), int, count=10)
[ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18]
```



Übungsaufgaben

Aufgabe 6.1.1

Erzeugen Sie ein numpy-Array mit den Werten $[-2\pi, -1.5\pi, -\pi, -0.5\pi, 0, 0.5\pi, \pi, 1.5\pi, 2\pi]$. Führen Sie die nachfolgenden Berechnungen darauf aus und geben Sie das jeweilige Ergebnis mit `print()` auf der Konsole aus.

- Berechnen Sie den Sinus, Cosinus und Tangens des Arrays.
- Multiplizieren Sie das Array elementweise mit sich selbst.
- Addieren Sie die Euler-sche Zahl zu jedem Wert im Array.
- Wandeln Sie das Array von Radians in Grad um.

- e) Runden Sie sämtliche Werte im Array auf zwei Nachkommastellen und bilden Sie die Summe des Ergebnisses. Welchen Wert erwarten Sie für diese Summe? Unterscheidet sich das Ergebnis von der Summe der Werte im ursprünglichen Array?

Aufgabe 6.1.2

Erzeugen Sie das Numpy-Array [0, 1, 2, 3, 4, 5] mit verschiedenen Methoden:

- a) Umwandeln einer list.
- b) Umwandeln eines tuple.
- c) Umwandeln eines range.
- d) Mittels np.arange().
- e) Aus der Funktion f = lambda x: x.
- f) Aus der Generatorfunktion natural_numbers().

```
def natural_numbers():
    i = 0
    while True:
        yield i
        i += 1
```

- g) Per Manipulation eines zuvor definierten Arrays:
- a) Erzeugt mit np.empty().
 - b) Erzeugt mit np.ones().
 - c) Erzeugt mit np.zeros().
 - d) Erzeugt mit np.full().

Kapitel 7

Dokumentation

Wie in allen Programmiersprachen ist vor allem bei großen Programmen eine gute Dokumentation wichtig. Viele Funktionen sind ohne ausreichende Beschreibungen nicht gut nachvollziehbar und man kann oft nur erahnen, was ein bestimmter Programmteil letztendlich bewirkt. Da von vielen Programmierern aus Bequemlichkeit und Schreibfaulheit auf eine umfassende Dokumentation verzichtet wird, sind viele Programme kaum bis überhaupt nicht wartbar oder veränderbar.

In Python gibt es für genau dieses Problem Tools, mit dessen Hilfe die Dokumentation von Programmen auf ein angenehmes Maß reduziert wird. Eines dieser plattformunabhängigen Werkzeuge mit dem Namen *epydoc* wird im Folgenden genauer betrachtet. epydoc analysiert Python-Programme anhand des Quellcodes, verwertet diesen und gibt anschließend eine Dokumentation als PDF- oder HTML-Datei aus. Bei dieser Analyse werden Docstrings benötigt, deren Benutzung und Einbindung noch genauer betrachtet wird.

epydoc

Unter <http://epydoc.sourceforge.net> können Sie sich die aktuelle epydoc-Version herunterladen und anschließend installieren.

7.1 Epydoc

Nach der erfolgreichen Installation kann epydoc über die Konsole aufgerufen werden. Dazu wird der Befehl *epydoc.py* benutzt. (in Linux ohne .py) Eine Dokumentation zu einer bestimmten Python-Datei ist dann möglich, wenn epydoc an der selben Stelle im Dateipfad ausgeführt wird.

Um das Format zwischen PDF und HTML zu wechseln, wird der Befehl `-pdf` beziehungsweise `-html` verwendet. Auch den Speicherort der Dokumentation kann frei variiert werden. Dazu wird der Befehl `-output` vor den gewünschten Verzeichnisort geschrieben.

Ein Beispiel für eine korrekte Dokumentationserstellung durch epydoc sieht wie folgt aus:

```
$ epydoc.py --html --output eigenes_verzeichnis testprogramm
```

Dadurch wird die Dokumentation zum Modul *testprogramm* als HTML-Datei im Verzeichnis *gewuenschtesverzeichnis* gespeichert.

7.2 Docstrings

Python bietet uns durch drei einfache oder doppelte Anführungszeichen die Möglichkeit, sogenannte Blockkommentare über mehrere Zeilen zu verfassen. Der darin eingefasste Text wird als Documentation String, kurz *Docstring* bezeichnet.

```
"""
Dieser Text ist ein Docstring.
Er wird von epydoc als solcher erkannt,
analysiert und interpretiert.
"""
```

Docstrings sollten zur Beschreibung aller Programmerteile benutzt werden, vor allem Funktionen und Klassen sollten durchgehend kommentiert werden.

```
class myclass(object):
    """Docstring zur Klasse myclass
    Beschreibung, wozu die Klasse dient.
    """

def myfunction():
    """Docstring zur Funktion myfunction
    Beschreibung, was die Funktion macht.
    """
```

Innerhalb des Programms kann man durch einen Befehl einen beliebigen Docstring aufrufen. Dazu dient das Attribut `_doc_`, das zu jeder Instanz automatisch erstellt wird.

```
>>> print myclass.__doc__
Docstring zur Klasse myclass
    Beschreibung, wozu die Klasse dient.
```

Python hat intern jedoch keine Möglichkeit, um für Variablen Docstrings zu nutzen. Diese Möglichkeit wird durch epydoc ermöglicht. Folgt in der folgenden Zeile nach einer Variablenzuweisung ein Blockkommentar, wird dieser automatisch als Docstring zur Variablen interpretiert. Eine weitere Möglichkeit ist es, den Docstring vor die Wertezuweisung der Variablen zu definieren. In diesem Fall wird auf die dreifachen Anführungszeichen verzichtet und je Zeile eine Raute mit Doppelpunkt #: vorangestellt.

```
a = 42
""" Die Variable a ist 42
"""

#: Die Variable b
#: ist 24
b = 24
```

7.3 Epytext

Um sicherzustellen, dass epydoc die Docstrings richtig interpretiert, wurde die Beschreibungssprache Epytext eingeführt. Darin sind Regeln für die Formatierung und die Umsetzung mit Docstrings festgehalten, um eine einheitliche Benutzung zu gewährleisten. Im Folgenden werden einige der Regeln und Möglichkeiten betrachtet.

Zum einen ist in Epytext, genau wie in Python die korrekte Einrückung von entscheidender Wichtigkeit.

```
def myfunction():
"""
    Wichtig ist die richtige Einrückung
    """
```

Auch Listen sind innerhalb von Docstrings möglich. Diese können als einfache Auflistung oder Nummerierung benutzt werden.

```
"""
Auflistung:
- Eier
- Milch
- Mehl

Nummerierung:
1. Aufstehen
2. Zähne putzen
3. Duschen
"""
```

Weitere Formatierungsmöglichkeiten werden kurz in folgendem Beispiel gezeigt. Der Aufbau ist dabei stets in der Form eines Großbuchstabens mit dem zu formatierenden Strings in geschweiften Klammern x^* .

```
"""
I{Dieser String ist kursiv}
F{Dieser String ist fett}
M{Dies ist ein mathematischer Ausdruck}
U{String ist eine URL und wird als Hyperlink interpretiert}
E{Verhindert die Interpretation}
"""
```

7.4 Zusammenfassung

Vielen Programmierern ist die ständige Dokumentation seit jeher ein Dorn im Auge. Mit epydoc ist in der Python-Programmierung jedoch ein Tool vorhanden, welche einem die Arbeit zu einem gewissen Teil abnimmt. Deshalb sollte dieses oder ein ähnliches Programm verwendet werden, um sicherzustellen, dass auch nach der Fertigstellung des Programms nachvollziehbar wird, was der eigentliche Sinn und Zweck hinter den einzelnen Programmteilen ist. Falls Sie sich nach diesem Grundüberblick noch weiter über epydoc und Epytext informieren möchten, können Sie sich die Dokumentation auf der offiziellen Homepage von epydoc anschauen.

Diese finden Sie unter <http://epydoc.sourceforge.net>.

Kapitel 8

Weiterführende Themen

8.1 Maschinelles Lernen in Python

Das Themengebiet des maschinellen Lernens kann verschiedene Komplexitätslevel erreichen. Grundlegend ist das mathematische Verständnis über die verschiedenen im maschinellen lernen eingesetzten Algorithmen. Diese werden in diesem Tutorial nicht beschrieben.

Sind die Algorithmen bekannt und sollen nun mittels Python angewendet werden, sollte zu Beginn mit einem kleinen Projekt gestartet werden. Hierbei ist es sinnvoll sich bereits am Anfang eine Vorgehensweise zu überlegen, wie auch bei allen anderen Projekten. Python bietet im Bereich des maschinellen Lernens viele unterschiedliche Möglichkeiten an, sodass bereits frühzeitig der Aufbau des Projekts entschieden werden sollte. Grob kann ein Projekt in fünf Schritte aufgeteilt werden, anhand derer später das Ergebnis verifiziert werden kann.

- a) zu lösendes Problem definieren
- b) Daten verstehen und vorbereiten
- c) mögliche Algorithmen evaluieren
- d) Ergebnisse verbessern
- e) Ergebnisse darstellen

Eine weit verbreitete Möglichkeit ist das Einbinden von bereits existierenden Bibliotheken, die viele Funktionalitäten von Haus aus anbieten. Eine eigene

Nachbildung von verbreiteten Algorithmen aus dem Bereich Maschinelles Lernen ist daher meist nicht nötig.

Eine zweite Möglichkeit ist die Integration von R. Bei R handelt es sich um eine eigene Programmiersprache, welche den Schwerpunkt in mathematischen Problemlösungen hat. Python und R lassen sich beide sowohl eigenständig, also auch in Verbindung miteinander einsetzen.

Im Folgenden Abschnitt gibt es eine Übersicht, über wichtige und bekannte Bibliotheken aus dem Bereich maschinelles Lernen. Die Anzahl der Bibliotheken macht den Einstieg nicht ganz leicht. Die Stärken und Schwächen der einzelnen Bibliotheken sollten betrachtet werden.

8.1.1 Bibliotheken

Im Bereich maschinelles Lernen sind schon viele Bibliotheken vorhanden, die unterschiedliche Schwerpunkte in dem Bereich bedienen. Aus diesem Grund erfolgt zuerst eine Übersicht über verbreitete Bibliotheken.

| | |
|----------------------------------|---|
| Download und Installation | Alle Bibliotheken die genutzt werden sollen müssen zuerst installiert werden. Hierfür gibt es je nach Bibliothek und teilweise je nach Betriebssystem mehrere Wege. Es wird empfohlen hier aus der jeweiligen Webseite die geeigneten Variante zu wählen und auszuführen. |
|----------------------------------|---|

| | |
|-------------------------------------|--|
| Versionen und Kompatibilität | Nach der Installation sollten alle Versionen ausgelesen und abgeglichen werden. Die Kompatibilität ist nicht durchgehend gewährleistet, das betrifft vor allem die unterschiedlichen Python-Versionen. |
|-------------------------------------|--|

Der folgende Codeausschnitt zeigt ein Beispiel. Dieser kann entweder direkt in der Eingabeaufforderung bzw. Konsole nach Start von Python ausgeführt werden oder innerhalb einer geeigneten Entwicklungsumgebung.

```
# Python version
import sys
print("Python: {}".format(sys.version))

# numpy version
import numpy
print("numpy: {}".format(numpy.__version__))

# pandas version
import pandas
```

```
print("pandas: {}".format(pandas.__version__))  
  
# rpy2 version  
import rpy2  
print("rpy2: {}".format(rpy2.__version__))
```

Die Ausgabe ist beispielsweise die Folgende:

```
Python: 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52)  
[MSC v.1900 32 bit (Intel)]  
  
numpy: 1.13.3  
pandas: 0.21.0  
rpy2: 2.8.6
```

Auf diese Art und Weise sollten alle Bibliotheken geprüft werden, da so eventuelle Kompatibilitätsprobleme oder fehlerhafte Installation frühzeitig erkannt werden können.

Bekannte Bibliotheken

Grob kann man zwischen Datenanalyse und Visualisierung unterscheiden. Zwei bekannte Bibliotheken aus dem Bereich Datenanalyse sind `numpy` und `pandas`, mit denen beispielsweise Gleichungen und Optimierungsprobleme gelöst werden können. Ergebnisse solcher Berechnungen können beispielsweise mithilfe des Moduls `matplotlib` visualisiert werden. [?]

In der Bibliothek `numpy` [?] wird ein flexibler Datentyp für mehrdimensionale Arrays zur Verfügung gestellt. Dies ermöglicht eine effiziente Durchführung von komplexen Rechnungen.

**Bibliothek
numpy**

Außerdem lassen sich Integrale berechnen, statistische Berechnungen durchführen und auch simulieren. All das wird für maschinelles Lernen benötigt. Da die Berechnungen mit Routinen nah an der Hardware durchgeführt werden, lassen sich bei entsprechender Programmierung effiziente Programme schreiben.

Die Arrays in `numpy` sind dreidimensional und können so eine Vielzahl von Anwendungsfällen abbilden. Der Fokus von `numpy` liegt in der Datenhaltung und Manipulation von Daten. Hier speziell die numerische Manipulation aus dem Bereich der linearen Algebra. Mit den Matrizen können bei-

spielsweise Multiplikationen und Dekompositionen durchgeführt werden. Aus diesem Grund sind `numpy`-Array oft die Datenstruktur, mit der weiterführende Bibliotheken arbeiten können.

Bibliothek pandas Die Webseite zu `pandas` [?] beschreibt dieses als gute Wahl zur schnellen und flexiblen Aufbereitung von Daten. `pandas` bietet verschiedene Möglichkeiten, um schnell auf Einträge zuzugreifen. Dies ist möglich, da mittels `pandas` Serien und Dataframes erzeugt werden können, die im Gegensatz zu einem Array in Python auch Spaltentitel und Indizes anbieten.

Die `describe()`-Methode bietet hierbei einen ersten Überblick über die im Dataframe enthalten Daten. Ohne weitere Programmierung werden Informationen wie Maximalwert, Minimalwert und Durchschnitt für jede Spalte berechnet und angezeigt.

Spalten und Zeilen können mittels `pandas` gefiltert, erweitert und verändert werden, sodass `pandas` oft im ersten Schritt genutzt wird, um die auszuwertenden Daten zu laden und genauer analysieren zu können.

Bibliothek scipy Ergänzend bzw. aufbauen auf `numpy` werden durch `scipy` [?] viele mathematische Operationen bereit gestellt. Das Modul `scipy` ist sehr mächtig und daher nochmal in Untermodule aufgeteilt. Innerhalb der Untermodule werden bestimmte Funktionalitäten gruppiert. Eine Übersicht hierzu gibt die Onlinedokumentation.

Bibliothek scikit-learn Viele DataMining bzw. Machine Learning Funktionalitäten werden bereits durch die `scikit-learn` [?] Bibliothek (manchmal aus `sklearn` abgekürzt) zur Verfügung gestellt. Die klassischen Algorithmen, wie `k-Means` oder `knn` sind bereits integriert.

Des Weiteren ist auch mit `scikit-learn` eine Aufbereitung der Daten möglich. Hier werden beispielsweise Normalisierung und Skalierung unterstützt. Insgesamt handelt es sich um eine sehr mächtige Bibliothek, mit der es möglich ist unterschiedliche Algorithmen zu probieren und verschiedene Test- und Trainingsverfahren zu testen. Es können auch neue Daten anhand der gelernten Modelle klassifiziert und vorhergesagt werden.

Bibliothek rpy2 Eine weitere Möglichkeit ist das Einbinden des Pakets `rpy2` [?]. Hierbei handelt es sich um eine Bibliothek, welche Komponenten aus R zur Verfügung stellt. `rpy2` ist zu sehen wie eine Schnittstelle zwischen Python und R. Es ist möglich R-Pakete mittels Python zu importieren und mit den darin enthaltenen Funktionalitäten zu interagieren.

Mit dem Modul `matplotlib` [?] können Daten in einem Diagramm dargestellt werden. Hiermit kann ein erstes Verständnis der Daten oder Ergebnisse erreicht werden. Es werden unter anderem Liniendiagramme, Histogramme, Balkendiagramme aber auch Heatmaps unterstützt. Hier können sowohl Achsen, Farben und auch Beschriftungen nach Bedarf angepasst werden. `matplotlib` unterstützt sowohl `pandas` als auch `numpy` und ist daher oft bereits am Anfang von hoher Bedeutung, um einen Überblick über die Daten zu bekommen.

**Bibliothek
matplotlib**



Übungsaufgaben

Aufgabe 8.1.1

Bitte beantworte für dich selbst folgenden Fragen:

- a) Ist der Einsatz von weiteren Bibliotheken für den Bereich Machine Learning sinnvoll?
- b) Was ist bei der Installation von verschiedenen Bibliotheken zu beachten?
- c) Welche Bibliothek sollte näher betrachtet werden, wenn es um die Arbeit mit mehrdimensionalen Arrays geht?
- d) In welcher Bibliothek sind die klassischen Algorithmen aus dem maschinellen Lernen bereits integriert?
- e) Zur Darstellung in Diagrammen ist welche Bibliothek besonders geeignet?

8.1.2 Daten laden

Es gibt verschiedene Herangehensweisen, meist bietet es sich an, erst mal einen groben Überblick über die Daten zu erhalten. Für die erste Erläuterung werden Datensätze angenommen, welche in folgender Datenstruktur vorliegen.

Daten aus Datei lesen

Maschinelles Lernen macht nur Sinn mit entsprechenden Daten. Diese sollten im Besten Fall bereits in einer strukturierten Form vorliegen.

Beispieldaten

```
ID|qm|he|ra|ka
1;345;3;33;hau
2;89;2.7;13;hau
3;64;2.5;8;woh
4;243;3;28;hau
5;76;2.8;9;foo
6;133;2.8;12;foo
7;74;3;8;woh
8;55;2.7;6;woh
9;240;3;19;hau
```

- Datei auslesen** Nun kann mit der Entwicklung begonnen werden. Um den Datentyp mit Daten zu versorgen findet sich im Folgenden ein kleiner Codeausschnitt:

```
# Bibliothek einbinden
import numpy as numpy

def readDataSet(filename):

    # Datei-Stream vorbereiten
    fr = open(filename)

    # Anzahl der Zeilen ermitteln
    numberOfRowsLines = len(fr.readlines())

    # Eine Numpy-Matrix erzeugen
    mymatrix = numpy.zeros((numberOfLines-1,3))

    fr = open(filename)
    index = 0

    # Zeilen nach und nach aus Datei lesen
    for line in fr.readlines():
        # Kopfzeile weg lassen
        if index != 0:
            line = line.strip()
            # Zeile in temporäre Liste splitten
            listFromLine = (line.split(','))

            # Liste in Matrix einfügen
            mymatrix[index-1,:] =
                listFromLine[1:4]
```

```
# Kategorien
classLabel = listFromLine[4]

if classLabel == "foo":
    color = 'yellow'
elif classLabel == "blub":
    color = 'blue'
else:
    color = 'red'

# Kategorie als Text-Label
classLabelVector.append(classLabel)
classColorVector.append(color)

index += 1

return mymatrix, classLabelVector, classColorVector

# Aufruf der Methode
dataset, classLabelVector, classColorVector =
    readDataSet("SampleFile.txt")
```

Die aufbereiteten Daten können im nächsten Schritt visualisiert werden. Vorher noch zwei weitere Varianten wie Daten geladen werden können.

Daten aus Paket laden

Eine weitere Möglichkeit ist das Laden von Daten aus Paketen. Hier wird beispielhaft das Lesen aus dem Paket `iris` vorgestellt.

iris-Dataset laden

```
import numpy as numpy
from sklearn import datasets

# Daten laden
dt_iris = datasets.load_iris()
iris = dt_iris.data[:, :4]

# targets von 10, 25 und 50
t = dt_iris.target[[10, 25, 50]]
print(t)
```

```
# target_names
tn = list(dt_iris.target_names)
print(tn)

# In numpy Array unwandeln
dataasny = numpy.array(iris)
```

Die Ausgabe für `target_names` ist folgende:

```
[0 0 1]
['setosa', 'versicolor', 'virginica']
```

Daten aus URL laden

iris-Dataset aus URL laden

Eine weitere Möglichkeit ist das Laden von Daten aus einer URL. Hier wird als Beispiel wieder das Paket `iris` genommen.

```
import pandas as pd

# Importing the Dataset
url = "https://archive.ics.uci.edu/ml/machine-
      learning-databases/iris/iris.data"

# Assign colum names to the dataset
names = ['sepal-length', 'sepal-width',
          'petal-length', 'petal-width', 'Class']

# Read dataset to pandas dataframe
dataset = pd.read_csv(url, names=names)
```



Übungsaufgaben

Aufgabe 8.1.2

- In der ersten Aufgabe werden die Daten aus einer Text-Datei gelesen.
 - Lese Daten aus der txt-Datei aus
 - Stelle sie als numpy-array bereit.
- In der zweiten Aufgaben soll der Datensatz aus `sklearn.datasets` gelesen werden.

- a) Lese aus `sklearn.datasets` den Datensatz `digits`.
- b) Gebe die targets 22, 37 und 54 aus.
- c) Liste schließlich die `target_names` auf.

Tipp:

Die URL <https://scikit-learn.org/stable/modules/classes.html#module-sklearn.datasets> gibt eine gute Auskunft über standardmäßig vorhandene Datasets.

8.1.3 Plots erzeugen

Aus den oben gelesenen Daten kann nun ein Plot erzeugt werden.

Zuerst mal ein ganz einfacher Plot.

Plot erzeugen

```
# Bibliotheken einbinden
import matplotlib.pyplot as pyplot
from sklearn import datasets

def plotplot():
    # Daten laden
    dt_iris = datasets.load_iris()
    iris = dt_iris.data[:, :2]

    # Plot definieren
    pyplot.plot(iris)
    pyplot.title("I'm the title of the plot")
    pyplot.xlabel("X-Label")
    pyplot.ylabel("Y-Label")

    # Plot anzeigen
    pyplot.show()

plotplot()
```

Die erzeugt den Plot wie in 8.1 dargestellt.

Eine weitere Darstellungsform könnte als Histogramm sein.

**Histogramm
erzeugen**

```
# Bibliotheken einbinden
```

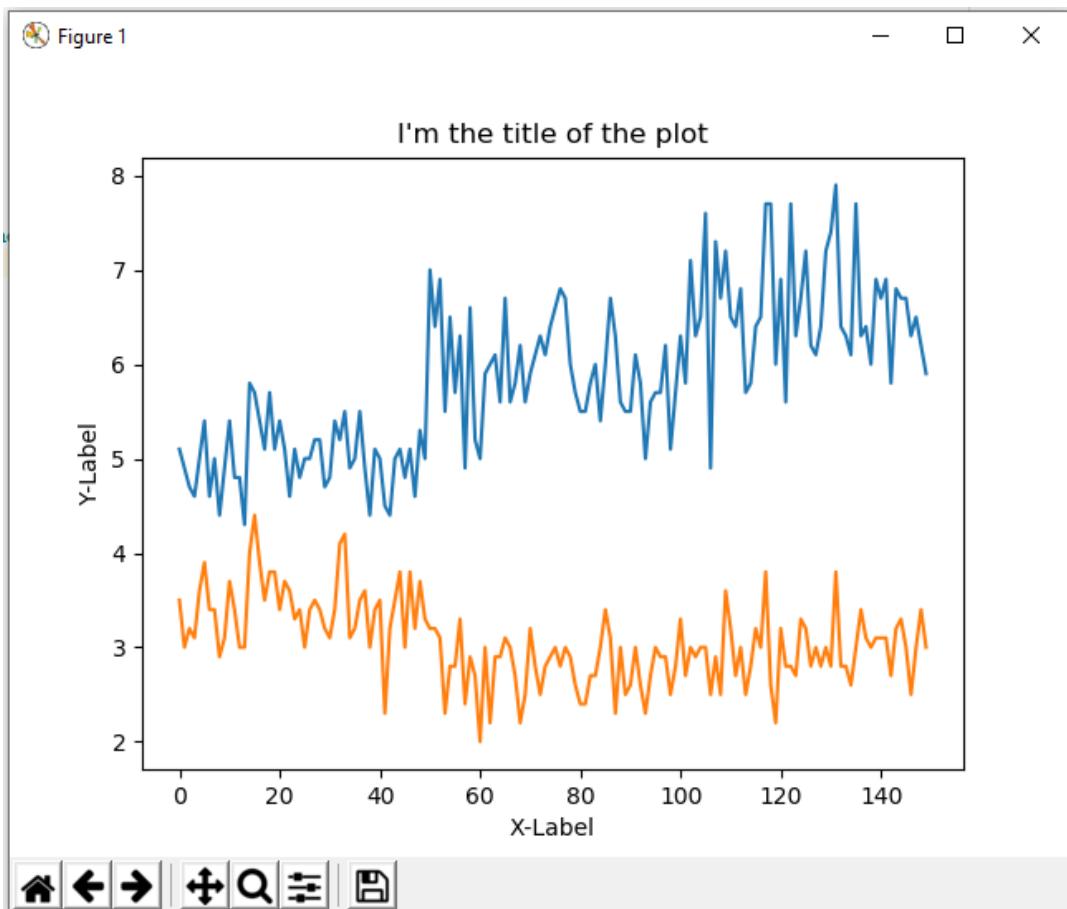


Abbildung 8.1: Plot mit Titel und Achsenbeschriftungen

```
import matplotlib.pyplot as pyplot
from sklearn import datasets

def plothist():
    # Daten laden
    dt_iris = datasets.load_iris()
    iris = dt_iris.data[:, :4]

    # Plot definieren
    pyplot.hist(iris)
    pyplot.title("I'm the title of the histogram")
    pyplot.xlabel("X-Label")
    pyplot.ylabel("Y-Label")
```

```
# Plot anzeigen  
pyplot.show()  
  
plothist()
```

Das erzeugte Histogramm sieht aus wie in Abbildung 8.2.

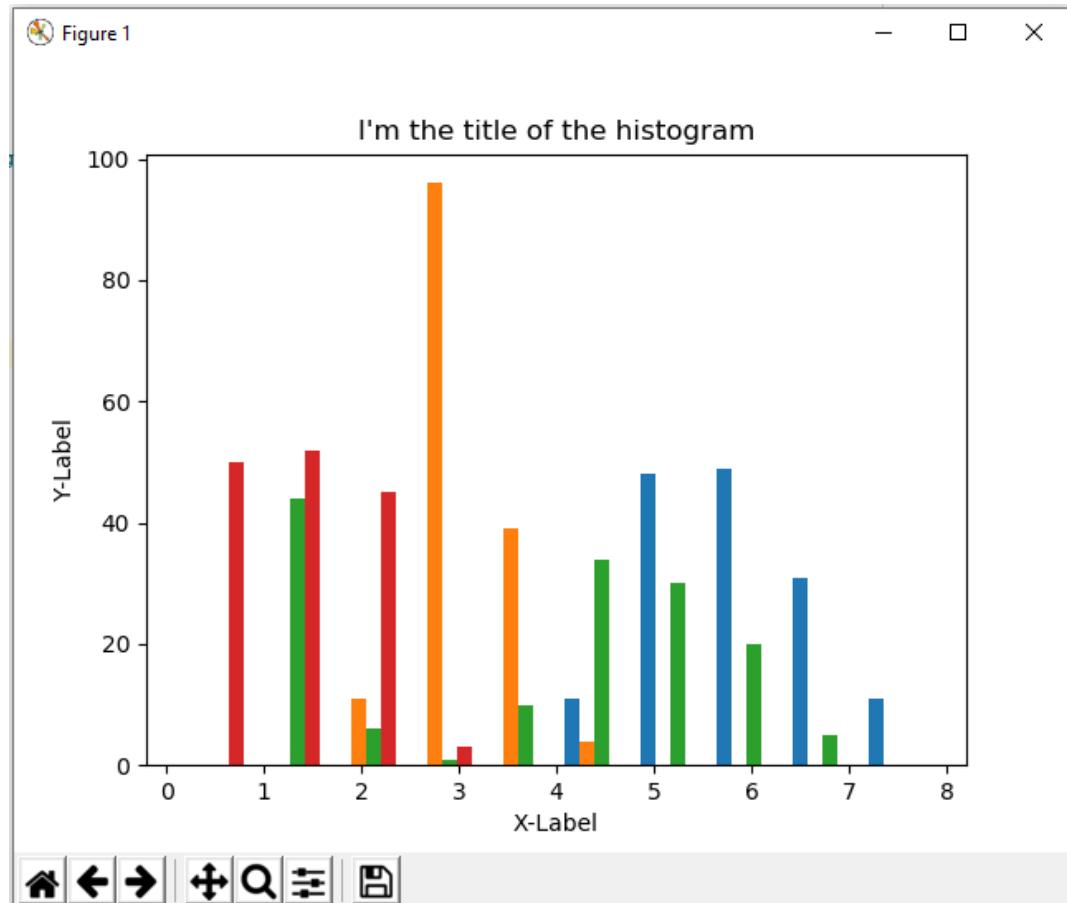


Abbildung 8.2: Histogramm-Plot mit Titel und Achsenbeschriftungen

Für das Beispiel oben bietet sich außerdem die Darstellung in einem Scatterplot an.

```
# Bibliotheken einbinden  
import matplotlib.pyplot as pyplot  
from sklearn import datasets  
  
def plotscatter():
```

Scatterplot erzeugen

```
# Daten laden
dt_iris = datasets.load_iris()
dataSet = dt_iris.data[:, :3]

# Plot definieren
fig = pyplot.figure()
ax = fig.add_subplot(111)

# Plot mit Daten versorgen
ax.scatter(dataSet[:, 0], dataSet[:, 1], dataSet[:, 2],
           marker='o')

# Achsenbeschriftungen festlegen
ax.set_title("I'm a scatter plot")
ax.set_xlabel("X-Label")
ax.set_ylabel("Y-Label")

# Plot anzeigen
pyplot.show()

# Aufruf
plotsscatter()
```

Der Code erzeugt einen Scatterplot wie in Abbildung 8.3.



Übungsaufgaben

Aufgabe 8.1.3

■ Zuerst wird ein normaler Plot erzeugt.

- Lese aus `sklearn.datasets` den `breast_cancer`-Datensatz.
- Schränke die Daten auf eine kleinere Menge (z.B. 20) ein.
- Erzeuge einen neuen Plot und vergabe einen Titel und die Achsenbeschriftungen.
- Zeige den Plot an.

■ In der zweiten Aufgabe werden wir ein Histogramm erzeugen.

- Lese aus `sklearn.datasets` den `breast_cancer`-Datensatz.

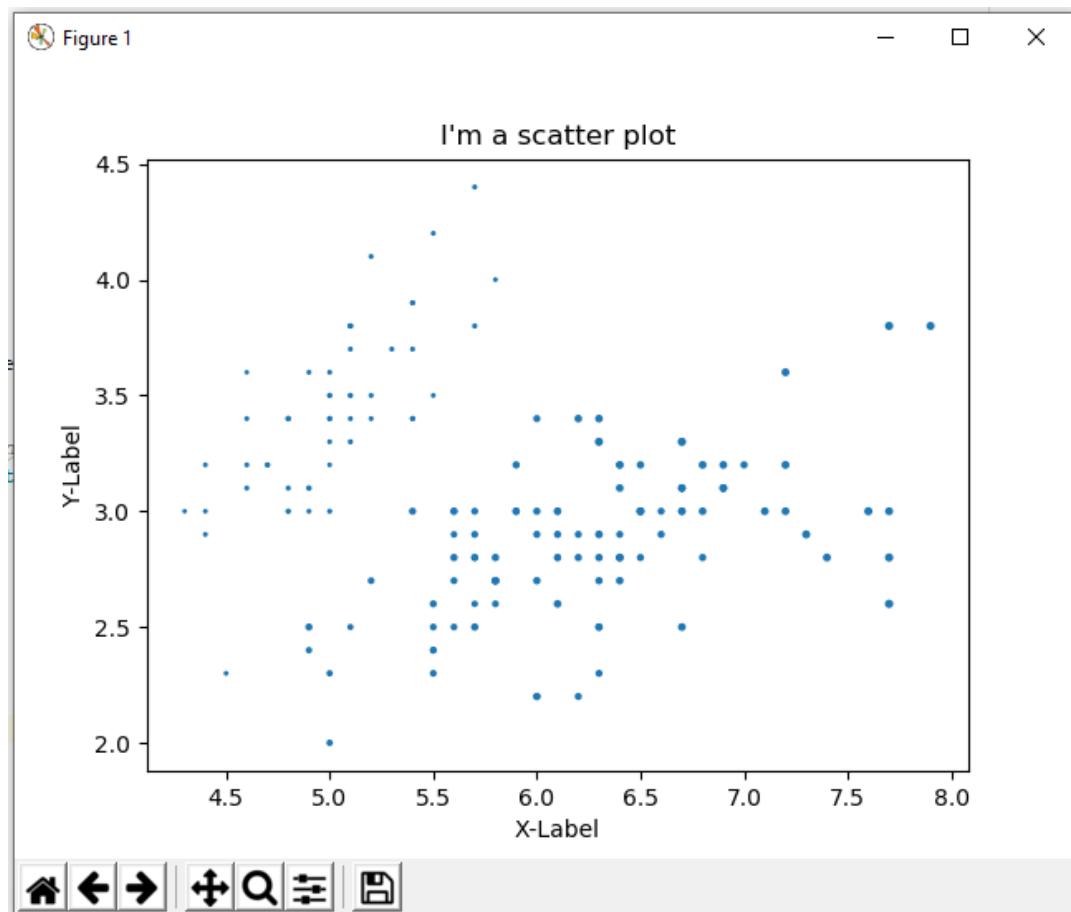


Abbildung 8.3: Scatter-Plot mit Titel und Achsenbeschriftungen

- Schränke die Daten auf zwei Spalten ein.
 - Erzeuge ein neues Histogramm und vergabe einen Titel und Achsenbeschriftungen.
 - Zeige den Plot an.
- Zum Abschluss noch einen Scatterplot.
- Lese aus `sklearn.datasets` den `breast_cancer`-Datensatz.
 - Schränke die Daten auf eine kleinere Menge (z.B. 20) ein.
 - Erzeuge einen neuen Scatter-Plot und vergabe einen Titel und Achsenbeschriftungen.

- Ändere die Farbe der Punkte auf rot.
- Ändere die Symbole zu Sternen.
- Zeige den Plot an.

8.1.4 Beispiel: knn-Klassifikation

Der knn-Algorithmus ist einer der bekanntesten Algorithmus aus dem Bereich Klassifikation und wird daher hier als Codebeispiel dargestellt und erläutert.

Imports

Folgende Imports sind für dieses Beispiel nötig:

```
# Importing Libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report,
```

Daten laden

Anschließend können die Daten geladen und die Spaltennamen definiert werden.

```
# Importing the Dataset
url = "https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data"

# Assign column names to the dataset
names = ['sepal-length', 'sepal-width', 'petal-length',
'petal-width', 'Class']
```

Daten lesen und vorbereiten

Jetzt werden die Daten gelesen und vorbereitet.

```
# Read dataset to pandas dataframe
dataset = pd.read_csv(url, names=names)

# Preprocessing
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, 4].values
```

Zur Berechnung müssen die Daten skaliert werden.

Daten skalieren

```
# feature scaling:
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

Nun kann der KNN-Algorithmus angewendet werden.

Algorithmus anwenden

```
# training und predictions
classifier = KNeighborsClassifier(n_neighbors=5)
classifier.fit(X_train, y_train)

# make predictions
y_pred = classifier.predict(X_test)
```

Jetzt können wir die Daten-Matrix ausgeben.

Matrix ausgeben

```
# print matrix
print(confusion_matrix(y_test, y_pred))
```

Als Ausgabe wird Folgendes angezeigt

```
[[ 9  0  0]
 [ 0 12  1]
 [ 0  1  7]]
```

Zusätzlich kann ein Report angezeigt werden, welcher eine gute Übersicht über die klassifizierten Daten gibt.

Report ausgeben

```
# print report
print(classification_report(y_test, y_pred))
```

Die Ausgabe ist die folgende:

| | precision | recall | f1-score | support |
|-----------------|-----------|--------|----------|---------|
| Iris-setosa | 1.00 | 1.00 | 1.00 | 9 |
| Iris-versicolor | 0.92 | 0.92 | 0.92 | 13 |
| Iris-virginica | 0.88 | 0.88 | 0.88 | 8 |
| | | | | |
| micro avg | 0.93 | 0.93 | 0.93 | 30 |
| macro avg | 0.93 | 0.93 | 0.93 | 30 |
| weighted avg | 0.93 | 0.93 | 0.93 | 30 |

Fehler ausgeben Um ein abschließendes Bild über die Klassifikation zu erhalten können nun noch die Fehler ausgegeben werden.

```
# define error array
error = []

# calculate the mean of error for all the predicted
# values where K ranges from 1 and 40
# Calculating error for K values between 1 and 40
for i in range(1, 40):
    knn = KNeighborsClassifier(n_neighbors=i)
    knn.fit(X_train, y_train)
    pred_i = knn.predict(X_test)
    error.append(np.mean(pred_i != y_test))
```

Fehler-Plot erzeugen Aus den berechneten Fehlern bei der Klassifikation kann nun noch ein Plot erzeugt werden.

```
# plot the error values against K values
plt.figure(figsize=(12, 6))
plt.plot(range(1, 40), error, color='red',
          linestyle='dashed', marker='o',
          markerfacecolor='blue', markersize=10)
plt.title('Error Rate K Value')
plt.xlabel('K Value')
plt.ylabel('Mean Error')
plt.show()
```

Der Plot zeigt sich wie in Abbildung 8.4.



Übungsaufgaben

Aufgabe 8.1.4

- Lese aus der URL https://archive.ics.uci.edu/ml/machine-learning-databases/libras/movement_libras.data die Daten.
- Lese außerdem die Names https://archive.ics.uci.edu/ml/machine-learning-databases/libras/movement_libras.names.
- Teile die Daten in Training- und Testdaten und führe eine Skalierung der Daten durch.

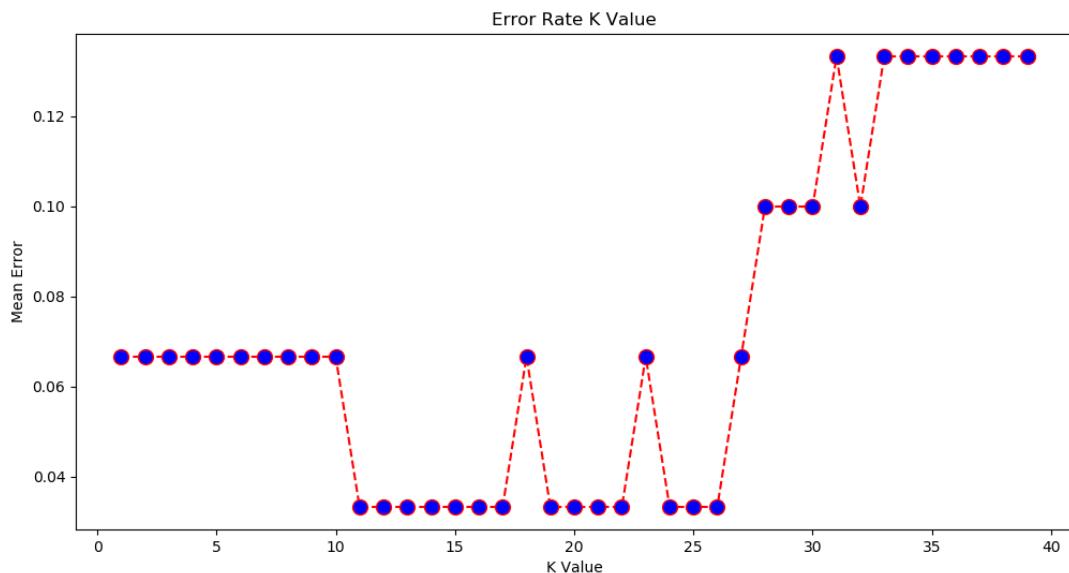


Abbildung 8.4: Ergebnis-Plot der Fehler nach Ausführung KNN

- d) Wende die Daten aus den `KNeighboursClassifier` mit Anzahl Nachbarn = 3 an.
- e) Mache Vorhersagen für die Test- und Trainingsdaten über `predict`.
- f) Erzeuge einen Plot und stelle die beiden Vorhersagen dar.
- g) Beschrifte den Plot und die Achsen. Färbe außerdem die Punkte und Linien.

Tipp:

Unter <https://archive.ics.uci.edu/ml/machine-learning-databases/> lassen sich weitere Beispieldatensätze finden.

8.1.5 Beispiel: Naive Bayes

Ein weiteres Beispiel aus dem Bereich Klassifikation im maschinellen Lernen ist der Naive Bayes Algorithmus.

Es sind wieder einige Importe nötig.

Importe

```
# Import Library of Gaussian Naive Bayes model
```

```
from sklearn.naive_bayes import GaussianNB
import numpy as np
```

Daten laden

Die Beispieldaten müssen definiert werden.

```
# assigning predictor and target variables
x = np.array([[-3, 7], [1, 5], [1, 2], [-2, 0], [2, 3],
              [-4, 0], [-1, 1], [1, 1], [-2, 2], [2, 7],
              [-4, 1], [-2, 7]])
y = np.array([3, 3, 3, 3, 4, 3, 3, 4, 3, 4, 4, 4])
```

Modell erzeugen

Nun kann das Modell erzeugt werden um mit den Daten zu trainieren.

```
# Create a Gaussian Classifier
model = GaussianNB()

# Train the model using the training sets
```

Vorhersage

Anschließend können die Daten vorhergesagt und die Ergebnisse angezeigt werden.

```
model.fit(x, y)

# Predict Output
predicted = model.predict([[1, 2], [3, 4]])
print(predicted)
```

Die Ausgabe dieser Vorhersage lautet:

```
[3 4]
```



Übungsaufgaben

Aufgabe 8.1.5

- Lege eine neue Python-Datei an.
- Lade über `sklearn` den Datensatz `wine`.
- Lasse die `target_names`\lstinline! und einen Teil der Daten anzeigen, um einen Überblick darüber zu bekommen.
- Erzeuge aus den Daten den X-Wert, aus den `target_names` die Y-Werte jeweils als `numpy-array`.

- e) Erzeuge eine Instanz des `sklearn.naive_bayes.GaussianNB`.
- f) Führe mittels `train_test_split` eine Aufteilung der Daten durch.
- g) Führe eine Vorhersage durch und lasse das Ergebnis anzeigen.

8.1.6 Beispiel: Lineare Regression

Beispiel aus dem Bereich Regression. Konkret wird hier eine lineare Regression dargestellt.

Es sind verschiedene Importe für nötig. Die werden im ersten Schritt importiert. **Importe**

```
import numpy as np
from sklearn import datasets
from scipy.stats import linregress
import matplotlib.pyplot as plt
```

Daten aus dem Datensatz `boston` auslesen und in `numpy`-array überführen. **Daten laden**

```
dt = datasets.load_boston()
boston = dt.data[:, :4]
datasetny = np.array(boston)
```

Die Berechnung der linearen Regression kann direkt aus dem Modul `scipy` erfolgen, es ist keine eigene Implementierung nötig. Die Daten die analysiert werden sollen müssen entsprechend mitgegeben werden. **Berechnung durchführen**

```
b, a, r, p, std = linregress(datasetny[2, :], datasetny[1, :])
```

Nun kann der Plot für die berechneten Werte erzeugt werden. **Plot erzeugen**

```
plt.scatter(datasetny[2, :], datasetny[1, :])
plt.plot([0, 130], [a, a + 130 * b], c="red", alpha=0.5)
plt.plot()
plt.title("LinRegress")
plt.xlim(0, 120)
plt.ylim(0, 800)
plt.xlabel("X-Label")
plt.ylabel("Y-Label")
plt.grid(alpha=0.4)
plt.xticks([x for x in range(42) if x % 10 == 0])
```

```
plt.yticks([x for x in range(42) if x % 100 == 0])
plt.show()
```

Die Anzeige im Plot ist wie in Abbildung 8.5.

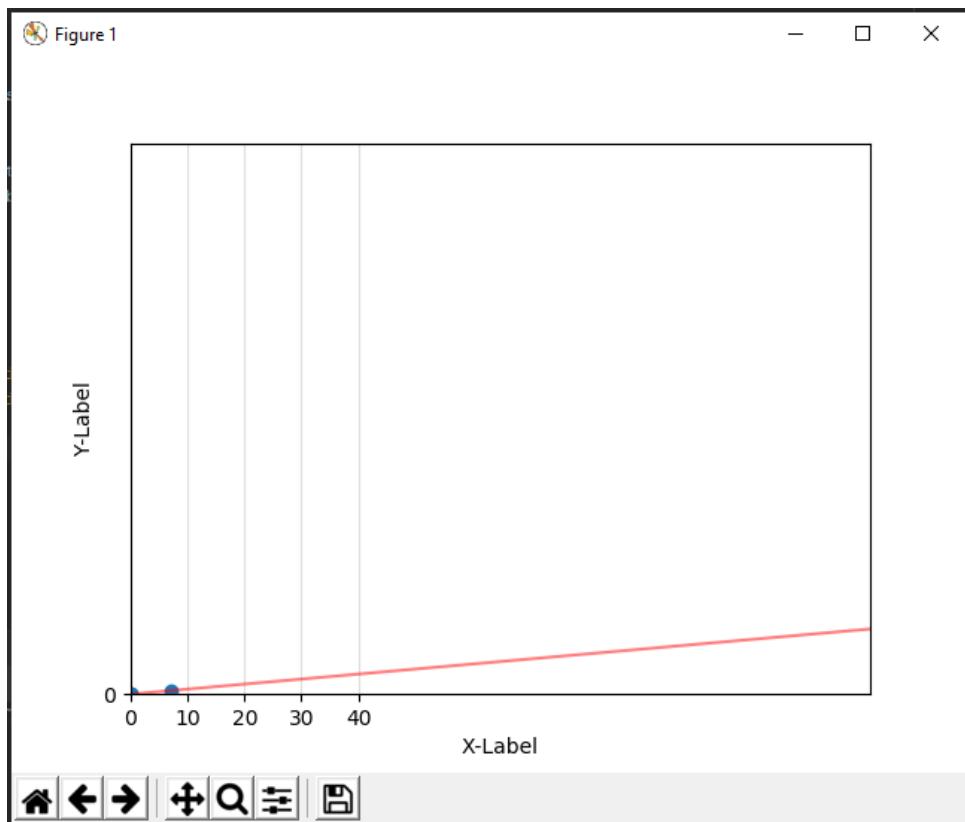


Abbildung 8.5: Ergebnis-Plot der linearen Regression

Somit haben wir nun auch eine lineare Regression erfolgreich durchgeführt.



Übungsaufgaben

Aufgabe 8.1.6

- Erstelle ein Skript, welches eine lineare Regression aus dem `wine`-Datensatz erzeugt. Dafür kann `datasets.load_wine` aus der `sklearn`-Bibliothek benutzt werden.
- Mit Spalte 3 und 5 erhält man die Abhängigkeit zwischen dem Alkohol und dem Magnesiumgehalt.

- c) Stelle die Datensätze und die Linie für die Regression in einem Scatterplot dar. Hierfür empfiehlt sich für x- und y-Achse der Maximalwert von 25.
- d) Gebe dem Plot außerdem einen Titel und Achsenbeschriftungen.

8.2 Datenbanken

In diesem Kapitel wird die Benutzung zweier verschiedener Datenbanksysteme, in der Programmiersprache Python, demonstriert. Zudem wird aufgeführt, wie eine Datenbank angelegt wird und wie die SQL-Befehle durchgeführt werden können.

8.2.1 Relationale Datenbanken

Relationale Datenbanken, die bereits im dritten Semester in der Vorlesung „Datenbanken“ durchgenommen wurden, dienen zur elektronischen Speicherung von Daten. Im folgendem Kapitel wird das Einbinden von SQLite in eine Python-Anwendung genauer erläutert.

SQLite

SQLite ist ein relationales Datenbankmanagementsystem, welches in einer C-Bibliothek enthalten ist. Anders als beispielsweise MySQL nutzt SQLite nicht das Client-Server-Model, sondern ist später im fertigen Programm lokal integriert. Eingesetzt wird SQLite sehr häufig in mobilen Applikationen. Diese speichern Nutzerdaten Lokal auf dem Gerät. Im weiteren Verlauf des Kapitels wird das Einbinden von SQLite in eine Python-Anwendung erläutert.

Einbinden von SQLite

Um SQL in einem Python-Projekt verwenden zu können, wird lediglich ein Import Statement benötigt.

```
#import sqlite
import sqlite3
```

Anschließend muss eine neue Datenbank angelegt werden. Hierfür kann die Methode `connect()` verwendet werden. Für diese Methode muss der Übergabeparameter aus einem String mit der Endung „.db“ übergeben werden.

```
#connect sqlite
connection = sqlite3.connect("beispiel.db")
```



8.1 Wie wird in SQLite eine neue Datenbank angelegt?

Um einen String, welcher das SQL-Statement beinhaltet, in die Datenbank einzubinden, muss zusätzlich ein Cursor angelegt werden.

```
#connect curser anlegen
cursor = connection.cursor()
```

Auch muss, damit der Cursor richtig verwendet wird, dies innerhalb der Datenbankverbindung ausgeführt werden. Mit `cursor.execute(sql_command)` wird eine Anfrage ausgeführt. Um den Cursor richtig zu verwenden, muss er innerhalb der Datenbankverbindung ausgeführt werden.

```
with connection:
    cur = connection.execute('sqlstatement')
```

Falls alle Tabellen angelegt und wie gewollt befüllt wurden, können mit der Methode `commit()` alle Änderungen an der Datenbank abgespeichert werden.

```
#commit
connection.commit()
```

Um die Verbindung zur Datenbank zu beenden wird die Methode `close()` genutzt.

```
#close
connection.close()
```

CREATE Create bei SQLite

In Folge dessen können nun einzelne Tabellen zur Datenbank hinzugefügt werden. Dies kann umgesetzt werden, indem ein neuer `sql_command` angelegt wird, welcher ein korrektes SQL-Kommando beinhalten muss.

```
#create
sql_command = """
```

```
CREATE TABLE mitarbeiter(
mitarbeiterid INTEGER PRIMARY KEY,
vname VARCHAR(20),
nname VARCHAR(30),
geschlecht CHAR(1),
beitritt DATE,
geburtstag DATE);"""
```

Insert bei SQLite**INSERT**

Um eine Tabelle im Anschluss zu befüllen wie folgt befüllt.

```
#insert
sql_command = """INSERT INTO mitarbeiter
(mitarbeiterid, vname, nname, geschlecht, geburtstag)
VALUES (NULL, "Peter", "Maffay", "m", "30.08.1949");"""
```

Select bei SQLite**SELECT**

Daten aus der SQLite Datenbank werden mit einem Select Befehl ausgelesen. Dieser ermöglicht es uns, einen oder mehrere Beiträge auszulesen. Mit dem SQL-Statement `sql_command1` im Listing 8.2.1 werden alle Einträge aus der Tabelle `mitarbeiter` ausgelesen.

```
#select
sql_command1 = """SELECT * FROM mitarbeiter;"""
sql_command2 = """SELECT * FROM mitarbeiter
WHERE mitarbeiterid = 1;"""
```

Update bei SQLite**UPDATE**

Um einen Eintrag im Nachhinein zu ändern, kann durch den Update Befehl ein oder mehrere bestimmte Einträge geändert werden. Im Listing wird der Vorname des Mitarbeiters mit der MitarbeiterId = 1 auf Peter gesetzt.

```
#update
sql_command = """UPDATE mitarbeiter SET vname="Peter"
WHERE mitarbeiterid = 1;"""
```



8.2 Welcher Befehl muss ausgeführt werden um einen bestehenden Eintrag zu ändern?

Delete bei SQLite**DELETE**

Eine Tabelle oder einen bestimmten Mitarbeiter kann durch einen `Delete` Befehl wieder entfernt werden. In folgendem Listing 8.2.1 werden zwei Möglichkeiten Daten aus der Datenbank zu entfernen aufgezeigt.

```
#delete
sql_command1 = """DELETE FROM mitarbeiter
WHERE mitarbeiterid = 1;???
sql_command2 = """DELETE FROM mitarbeiter;???
```

8.2.2 NoSQL Datenbanken

NoSQL steht für „not only SQL“. Hierbei wird SQL als Synonym für relationale Datenbanksysteme verwendet. Die Grundidee ist, dass nicht unbedingt aus alten Gewohnheiten heraus ein relationales Datenbanksystem gewählt wird. Vielmehr soll sich für ein Datenbanksystem entschieden werden, welches am Besten zum geplanten Projekt passt. Entstanden sind solche NoSQL Datenbanken unter anderem durch soziale Netzwerke. Hier müssen mehrere Millionen Daten sehr schnell gespeichert und abgerufen werden. Eine solche Masse an Anfragen stellt ganz neue Anforderung an Datenbanksysteme.[?] Die wichtigsten Kategorien von NoSQL-Datenbanken sind Key-Value, spaltenorientierte, Value und dokumentenorientierte Datenbanken. Im folgenden Kapitel, wird das Einbinden der dokumentenbasierten NoSQL Datenbank MongoDB genauer beschrieben.[?]

MongoDB

MongoDB ist eine open-source dokumentenbasierte NoSQL-Datenbank, die unter anderem eine hohe Performance und automatische Skalierung bietet. Als `record` wird in MongoDB eine Datenstruktur (key/value) mit Name und den dazugehörigen Werten bezeichnet. Diese MongoDB Dokumente sind ähnlich zu den uns bekannten JSON-Objekten. Wie die Einbindung und die Benutzung einer MongoDB Datenbank umgesetzt wird, wird nachfolgend genauer erläutert.

Einbinden von MongoDB

Um eine Verbindung mit MongoDB herzustellen, muss zunächst die Python Distribution PyMongo installiert werden.

```
#import mongodb
import pymongo
```

Anschließend muss mit mongod eine MongoDB Instanz gestartet werden.

```
#mongod Instanz starten  
  
$ mongod  
client = MongoClient()
```

Im Anschluss darauf, muss ein MongoClient erstellt werden, welcher auf die laufende mongod Instanz zugreift und sich dabei mit dem Standart-Host und Standart-Port verbindet.

```
#create client  
from pymongo import MongoClient  
client = MongoClient()
```

Host und Port können aber auch durch eines der folgenden Formate explizit spezifiziert werden.

```
#Host Name und Passwort spezifizieren  
client1 = MongoClient('localhost', 12345)  
client2 = MongoClient('mongodb://localhost:12345/')
```

In PyMongo wird mit attribute style access auf die Datenbanken zugegriffen, da eine Instanz von MongoDB mehrere unabhängige Datenbanken unterstützt. Dabei können die beiden folgenden Formate verwendet werden.

```
#Zugriff auf die Datenbank in zwei Varianten  
db1 = client.test_database  
db2 = client['test-database']
```

Das Äquivalent zu Tabellen in relationalen Datenbanken in MongoDB wird „Connections“ genannt. Diese bestehen aus mehreren Dokumenten. Der Zugriff erfolgt genau wie bei einer SQL-Datenbank.

```
#Zugriff auf Connections in zwei Varianten  
collection = db.test_collection  
collection = db['test-collection']
```

Die oben genannten Dokumente (JSON-Style) sind die Repräsentanten und Speicher der Daten in der Datenbank.

```
#JSON  
import datetime  
test = { "author": "Lukas",
```

```
"text": "Hallo",
"tags": ["hallo", "pymongo"]
"date": datetime.datetime.utcnow() }
```

**INSERT,
DELETE****Insert und Delete bei MongoDB**

Ein Dokument wird mit der Methode `insert_one()` hinzugefügt.

```
#insert_one
test = db.test
test_id = test.insert_one(test).inserted_id
test_id
ObjectId('...') (Ausgabe)
```

Beim Einfügen eines Dokuments wird diesem automatisch eine `_id` zugeschenkt, falls diese noch keine vorher bestimmte `_id` hat. Die `_id` muss einzigartig in einer Collection sein. Nachdem das Dokument hinzugefügt wurde, wird eine Test Collection erstellt. Mit Hilfe der Ausgabe einer Liste aller Collections wird dies bestätigt.

```
#collection_names
db.collection_names(include_system_collections=False)
```

Um ein bereits hinzugefügtes Dokument zu löschen wird die Methode `delete_one()` verwendet.

SELECT**Select bei MongoDB**

Mit der Methode `find_one()` kann auf bestimmte oder das erste (kein Parameter) Dokument einer Collection zugegriffen werden.

```
#find_one mit autor
test.find_one({"author": "Lukas"})
```

Mit Hilfe der `_id` kann auch auf einzelne Dokumente zugegriffen werden.

```
#find_one mit id
test.find_one({_id?: test_id})
```



8.3 Wofür wird bei MongoDB die Methode `find()` verwendet?

UPDATE**Update bei MongoDB**

Um einem Dokument neue Parameter geben zu können, kann die Methode `update_one(altes Dokument, neues Dokument)` genutzt werden. Das neue Dokument muss den Operator `$set` enthalten.

```
#update
newvalue { "$set": { "author": "Lukas" } }
```

Die Methoden `insert_one()` und `find_one()` können in abgeänderter Form auch für mehrere Dokumente genutzt werden. Mehrere Dokumente werden mit `insert_many()` hinzugefügt.

Einen Zugriff auf mehrere Dokumente wird mit `find()` umgesetzt. Die Anzahl der Dokumente wird in unserem Fall mit `test.count_documents()` ausgelesen. Mit `.sort` werden Ergebnisse nach verschiedenen Parametern sortiert. In Listing 8.2.2 wird ein neuer Index erstellt.

```
#Create Index
result = db.profiles.create_index(
[('user_id', pymongo.ASCENDING)], unique=True)
```

Dies hat zur Folge, dass nun `_id` und `user_id` existieren, welche einzigartig sein müssen. Zuletzt wird die Verbindung mit der Datenbank unterbrochen.

```
#close
client.close
```



8.4 Wofür wird die Methode `count_documents()` verwendet?



Übungsaufgaben

Aufgabe 8.2.1

Welche Methode muss benutzt werden, um folgendes Dokument in einer Collection zu finden?

```
#Aufgabe 2
uebung = {
"author": "Sebastian",
"text": "Hallo",
"tags": ["hallo", "pymongo"]
?date?: datetime.datetime(2009, 11, 12, 11, 14) }
```

Aufgabe 8.2.2

Erstellen Sie mit Hilfe von SQLite eine Datenbank. Anschließend erstellen Sie eine Tabelle „Person“ mit den Spalten **Vorname**, **Nachname** und **Alter**. Zuletzt befüllen Sie Ihre Tabelle mit zwei Datensätzen Ihrer Wahl.

8.3 Nebenläufigkeit

Mit Nebenläufigkeit ist eine Eigenschaft von zwei oder mehr Aktivitäten gemeint. Eine beliebige Anzahl an Aktivitäten wird als nebenläufig bezeichnet, wenn die Reihenfolge der Ausführung der einzelnen Aktivitäten irrelevant für das Ergebnis ist. Hierbei ist auf den Unterschied zwischen Nebenläufigkeit und Parallelität zu achten. Während Nebenläufigkeit eine Eigenschaft darstellt, ist Parallelität eine mögliche Herangehensweise an das Ausführen von nebenläufigen Aktivitäten (vgl. [?]).

Im folgenden Kapitel wird beschrieben, wie in Python durch Threads und Prozesse nebenläufige Programmierung realisiert werden kann. Python unterscheidet sich in Bezug auf Parallelität stark von anderen Programmiersprachen. In Abschnitt 8.3.1 wird auf diesen Unterschied eingegangen. Anschließend wird in Abschnitt 8.3.2 beschrieben, wie in Python Nebenläufigkeit durch Threads realisiert werden kann. Neben der Erzeugung von Threads werden Synchronisations- und Steuerungsmechanismen besprochen. Nachdem nun Nebenläufigkeit durch Threads erreicht wurde, wird sich in Abschnitt 8.3.3 auf Prozesse bezogen.

8.3.1 Parallelität in Python

In der Referenzimplementierung CPython des Python-Interpreters existiert ein Konstrukt, welches eine echte parallele Ausführung von Python-Code verhindert. Bei diesem Konstrukt handelt es sich um das sogenannte Global-Interpreter-Lock, oder kurz GIL. Für die Verwendung eines GILs in Python sprechen mehrere Punkte. Python wurde so entworfen, dass es leicht zu verwenden ist, um den Entwicklungsprozess zu beschleunigen. Ein GIL verhindert, dass sich mehrere Threads gleichzeitig in der Ausführung befinden können, was die Entwicklung von Multithreaded-Programmen erheblich erleichtert. Weiterhin wurde der Funktionsumfang von Python durch viele in C geschriebene Erweiterungen ergänzt. Um Inkonsistenzen zu verhindern, benötigen diese C-Erweiterungen eine threadsichere Speicherverwaltung, die

durch das GIL garantiert ist. Die Verwendung des GILs erleichtert auch die Integration von nicht threadsicheren C-Bibliotheken. Da das Einbinden von C-Bibliotheken durch das GIL leicht zu realisieren ist, existieren viele Erweiterungen zu Python, die zur weiten Verbreitung von Python führten.

Durch die Verwendung des GILs ist eine parallele Programmierung in Python allerdings nicht gänzlich ausgeschlossen. Lediglich Aufgaben, die CPU gebunden sind, sind hierdurch betroffen. I/O-gebundene Aufgaben, wie zum Beispiel die Anfrage von Daten aus einer Datenbank, oder die Abfrage von Benutzereingaben, können auch trotz des GILs parallel ausgeführt werden. Durch die Verwendung von mehreren Prozessen ist es auch möglich, parallele Ausführung von Python-Code zu erreichen. Dies funktioniert, da jeder Python-Prozess seinen eigenen Python-Interpreter und somit auch ein eigenes GIL besitzt.

8.3.2 Threads

In Python werden zwei APIs zur Verwendung von Threads angeboten, die Low-Level API aus dem `_thread`-Modul und die Higher-Level API aus dem `threading`-Modul. Es wird sich an dieser Stelle auf das `threading`-Modul beschränkt, da es intern auf dem `_thread`-Modul basiert und eine Schnittstelle anbietet, die das Programmieren von Multithreaded-Programmen erleichtert. Diese Schnittstelle ist an der Thread-Schnittstelle von Java angelehnt und sollte daher für Java-Entwickler leicht zu verwenden sein. Allerdings gibt es einige Unterschiede zwischen dem Python-Modul und der entsprechenden Implementierung in Java. So sind Bedingungsvariablen und Locks separate Objekte in Python und es ist auch nur eine Teilmenge des Verhaltens eines Java-Threads in Python verfügbar. Ein Python-Thread kennt keine Prioritäten und Thread-Gruppen und er kann nicht zerstört, gestoppt, angehalten, fortgesetzt oder unterbrochen werden. Soweit vorhanden sind die statischen Methoden aus der Java-Thread-Klasse auf Modul-Ebene in Python implementiert.

Thread Objekte

Es gibt zwei Möglichkeiten einen Thread zu erzeugen. Entweder wird dem Konstruktor ein aufrufbares Objekt übergeben,

```
# chapters/nebenlaufigkeit/src/thread_erzeugung.py
# Beispiel zur Threaderzeugung
```

```
def task():
    # Nebenläufig ausgeführte Aufgabe

thread = threading.Thread(target=task)
thread.start()
```

oder die `run()`-Methode wird in einer von `Thread` abgeleiteten Klasse überschrieben.

```
# chapters/nebenlaufigkeit/src/thread_erzeugung.py
# Beispiel zur Threaderzeugung

class MyThread(threading.Thread):
    def run(self):
        # Nebenläufig ausgeführte Aufgabe

thread = MyThread()
thread.start()
```

Der Konstruktor der `Thread`-Klasse bietet noch weitere Parameter an:

- `group` sollte immer `None` sein. Es ist aktuell reserviert für spätere Erweiterungen.
- `name` setzt den Namen des Threads.
- `args` ist ein Tupel aus Parametern für das mit `target` definierte aufrufbare Objekt.
- `kwargs` ist ein Dictionary aus Schlüsselwort-Parametern für `target`.
- `daemon` setzt die Dämon-Eigenschaft des Threads.

Es ist anzumerken, dass ein `Thread`-Objekt bei seiner Erzeugung noch nicht gestartet wird. Hierzu muss explizit die `start()`-Methode aufgerufen werden. Wurde ein Thread gestartet, wird er als „lebendig“ angesehen. Dies bleibt er solange, bis seine `run()`-Methode verlassen wurde. Hierbei macht es keinen Unterschied, ob sie regulär verlassen wurde oder Aufgrund einer Exception. Der aktuelle Status eines Threads kann mittels der `is_alive()`-Methode abgefragt werden. Soll auf das Beenden eines anderen Threads ge-

wartet werden, so kann seine `join()`-Methode aufgerufen werden. Hiermit wird der aufrufende Thread blockiert, bis der andere beendet ist. Die `join`-Methode nimmt einen optionalen Parameter des Typen `float` entgegen, der als Timeout in Sekunden dient. Wird ein Timeout angegeben, ist es wichtig, dass nach dem `join()`-Aufruf die Methode `is_alive()` aufgerufen wird. Da `join()` immer `None` zurückgibt, ist es ansonsten nicht möglich, zu wissen, ob der Thread tatsächlich beendet wurde, oder nur der Timeout abgelaufen ist.



8.1 Wie kann auf das Ende der Ausführung eines Threads gewartet werden?

Jeder Thread besitzt einen Namen, der initial über den Konstruktor oder direkt über das `name`-Attribut gesetzt werden kann. Threads können als Dämon gekennzeichnet werden. Sobald nur noch Dämon-Threads aktiv sind, wird das Python-Programm beendet. Die Dämon-Eigenschaft kann initial über den Konstruktor gesetzt werden. Wird kein Wert übergeben, übernimmt der Thread standardmäßig den Wert des erzeugenden Threads. Über das `deamon`-Attribut eines Threads kann die Eigenschaft abgefragt und gesetzt werden. Hierbei ist es wichtig, dass die Eigenschaft immer vor dem Aufruf der `start()`-Methode gesetzt wird. Wird sie nach dem Starten des Threads geändert, so wird ein `RuntimeError` geworfen.

Achtung:

Dämon-Threads werden sofort beendet, wenn keine normalen Threads mehr aktiv sind. Das heißt, dass ihre Ressourcen, wie zum Beispiel geöffnete Dateien oder Datenbanktransaktionen, gegebenenfalls nicht ordentlich freigegeben werden. Um dies zu verhindern, sollten die betroffenen Threads nicht die Dämon-Eigenschaft besitzen und es sollten geeignete Signalisierungsmechanismen eingesetzt werden (siehe Abschnitt 8.3.8).



Übungsaufgaben

Aufgabe 8.3.1

Schreiben Sie ein Programm, das einen Thread erzeugt, der die Zahlen 1 bis 10 ausgibt, und sich dann beendet. Zwischen den Ausgaben soll der Thread eine Sekunde warten.

Tipp:

Um einen Thread warten zu lassen, können Sie die `sleep()`-Methode verwenden. Diese Methode kommt aus dem `time`-Modul und nimmt die Zeit in Sekunden, die zu schlafen ist, als Parameter entgegen.

Aufgabe 8.3.2

Erweitern Sie Ihr Programm aus Aufgabe 8.3.1 so, dass nun 100 Threads mit unterschiedlichen Namen erzeugt werden. Diese Threads sollen wieder die Zahlen 1 bis 10 ausgeben und zusätzlich ihren Namen angeben, damit es nachvollziehbar ist, welcher Thread gerade die Ausgabe tätigt. Was fällt Ihnen auf, wenn Sie Ihr Programm mehrmals ausführen?

Synchronisation

Die meisten Anwendungen, in denen mehrere Threads zum Einsatz kommen, erfordern einen Mechanismus, der die Zugriffe der einzelnen Threads auf gewisse Daten synchronisiert. Hierdurch wird unter anderem vermieden, dass auf invaliden Datensätzen gearbeitet wird, oder ein Datenupdate verloren geht. Im Folgenden wird ein Beispiel betrachtet, bei dem es zu Fehlern aufgrund von fehlender Synchronisationsmechanismen kommt. Es werden anschließend neue Konstrukte eingeführt, die die Fehler beheben werden.

Betrachtet wird nun die folgende `Counter`-Klasse. Sie besitzt das Attribut `count`, welches durch Aufruf von `increment()` in Einer schritten erhöht wird.

```
# chapters/nebenlaufigkeit/src/synchronisation_fehler.py
# Beispiel zum Auftritt von Fehlern ohne
# Verwendung von Synchronisation

class Counter:
    def __init__(self):
        self.count = 0

    def increment(self):
        self.count += 1
```

Es ist weiterhin die `IncrementerThread`-Klasse gegeben, welche bei der Initialisierung ein `Counter`-Objekt erwartet. Dieser Thread ruft eine Millionen mal die `increment()`-Methode des `Counters` auf und beendet sich anschließend.

```
# chapters/nebenlaufigkeit/src/synchronisation_fehler.py
# Beispiel zum Auftritt von Fehlern ohne
# Verwendung von Synchronisation

class IncrementerThread(threading.Thread):
    def __init__(self, counter):
        threading.Thread.__init__(self)
        self.counter = counter

    def run(self):
        for _ in range(1000000):
            self.counter.increment()
```

Für dieses Beispiel werden nun 10 IncrementerThreads erzeugt und gestartet. Anschließend wird auf ihre Terminierung gewartet und dann der Wert des count-Attributs des Counters ausgegeben:

```
# chapters/nebenlaufigkeit/src/synchronisation_fehler.py
# Beispiel zum Auftritt von Fehlern ohne
# Verwendung von Synchronisation

counter = Counter()
threads = []
for _ in range(10):
    thread = IncrementerThread(counter)
    thread.start()
    threads.append(thread)

for thread in threads:
    thread.join()

print(counter.count)
```



8.2 Welche Ausgabe würde erwartet werden, wenn 10 Threads den Counter eine Millionen mal inkrementieren?

Dieser Programmcode würde vermuten lassen, dass bei jeder Ausführung der Wert 10000000 ausgegeben wird, da jeder der 10 Threads den Counter eine Millionen mal inkrementiert. Erstaunlicherweise werden allerdings bei mehrmaliger Ausführung unterschiedliche Werte ausgegeben. Diese können zum Beispiel wie folgt aussehen:

```
# Mögliche Ausgaben:
```

```
5237496  
3561559  
4089438  
4526494
```

Dieses Phänomen lässt sich so erklären, dass die Operation in `increment()` nicht atomar ist. Genau genommen werden in ihr drei Operationen, eine lesende, eine addierende und eine schreibende, ausgeführt. Somit kann es vorkommen, dass zum Beispiel der erste Thread den aktuellen `count`-Wert liest und dann die aktive Ausführung an einen anderen Thread abgeben muss. Dieser zweite Thread liest nun den selben `count`-Wert wie der erste Thread, inkrementiert ihn und schreibt den neuen Wert zurück in das Attribut. Nun wechselt die aktive Ausführung zurück zum ersten Thread, welcher noch den alten `count`-Wert gelesen hat. Dieser alte Wert wird nun erneut inkrementiert und zurückgeschrieben. Somit wurde der `Counter` effektiv nicht zweimal, sondern nur einmal inkrementiert. Um dieses Verhalten zu verhindern, muss sichergestellt werden, dass die drei einzelnen Operationen atomar ausgeführt werden, sprich, dass sie entweder ganz oder gar nicht ausgeführt werden.

Locks

Als unterste Synchronisationsebene bietet Python die Klasse `Lock` an. Ein Objekt dieser Klasse befindet sich immer in einem von zwei Zuständen, es ist entweder offen oder geschlossen. Nach der Initialisierung befindet es sich zuerst im geöffneten Zustand. Ein `Lock`-Objekt stellt die beiden Methoden `acquire()` und `release()` zur Verfügung. Wird `acquire()` auf einem offenen `Lock` aufgerufen, so begibt sich das `Lock` in den geschlossenen Zustand und die Methode kehrt sofort zurück. Sollte die `acquire()`-Methode aufgerufen werden, wenn sich das `Lock` im geschlossenen Zustand befindet, so blockiert sie solange, bis `release()` in einem anderen Thread aufgerufen wird und somit den Zustand des `Locks` wieder zu geöffnet ändert. Die blockierte `acquire()`-Methode schließt dann das `Lock` wieder und kehrt zurück. Wird auf einem offenen `Lock` die `release()`-Methode aufgerufen, so wird ein `RuntimeError` geworfen. Falls mehrere Threads durch `acquire()` blockiert werden, wird nur ein Thread fortgesetzt, sobald `release()` aufgerufen wurde. Welcher der blockierten Threads fortgesetzt wird, ist hierbei nicht definiert.

Wurde `acquire()` aufgerufen, sollte garantiert sein, dass auch `release()` aufgerufen wird. Wird eine `Exception` geworfen, kann dies allerdings nicht immer garantiert sein. Aus diesem Grund wird empfohlen, den durch das

Lock geschützten Programmcode in einen `try`-Block zu schreiben und den Aufruf von `release()` in den `finally`-Block zu schreiben:

```
# chapters/nebenlaufigkeit/src/lock_acquire_release.py
# Beispiel zur Verwendung von Lock-Objekten

lock.acquire()
try:
    # kritischer Code
finally:
    lock.release()
```

Das Gleiche kann mit dem `with`-Statement erreicht werden, da die `Lock`-Klasse das Context-Management-Protokoll unterstützt. Hierbei werden die beiden Methoden `aquire()` und `release()` automatisch aufgerufen:

```
# chapters/nebenlaufigkeit/src/lock_acquire_release.py
# Beispiel zur Verwendung von Lock-Objekten

with lock:
    # kritischer Code
```

Ist es nicht gewünscht, dass `aquire()` blockiert, wenn sie mehrmals aufgerufen wird, ohne dass das `Lock` freigegeben wurde, so kann ihr auch der optionale Parameter `blocking=False` übergegeben werden. In diesem Fall kehrt `aquire()` sofort zurück, egal in welchem Zustand sich das `Lock` befindet. Es muss nun der Rückgabewert von `aquire()` betrachtet werden, um zu erfahren, ob das `Lock` offen oder geschlossen ist. Ist das `Lock` bereits geschlossen, wird der Wert `False` zurückgegeben. Andernfalls wird `True` zurückgegeben und das `Lock` ändert seinen Zustand zu geschlossen. Weiterhin ist es möglich, einen Timeout mittels des optionalen Parametes `timeout` zu spezifizieren. Hierbei kann eine beliebige Zeit in Sekunden als `float` Wert angegeben werden. In diesem Fall blockiert `aquire()` maximale die spezifizierte Zeit. Wurde in dieser Zeit das `Lock` erlangt, so gibt `aquire()` `True` zurück, andernfalls `False`. Die Angabe eines Timeouts ist nur erlaubt, wenn der Parameter `blocking` den Wert `True` besitzt.



Übungsaufgaben

Aufgabe 8.3.3

Passen Sie die beschriebene Counter-Klasse so an, dass sie auch dann korrekt funktioniert, wenn sie von mehreren Threads gleichzeitig verwendet wird.

Aufgabe 8.3.4

Erweitern Sie die `increment()`-Methode der Counter-Klasse um die Möglichkeit, einen Wert anzugeben, um den inkrementiert werden soll. Ist der angegebene Wert größer 1, so soll die `increment()`-Methode sich rekursiv selbst aufrufen. Der Wert des Counters soll also immer nur direkt um 1 erhöht werden. Ergänzen Sie in der IncrementerThread-Klasse den neuen Parameter beim Aufruf von `increment()`. Was fällt Ihnen auf, wenn Sie das Programm ausführen?

Reentrant Locks
Um das Problem aus Aufgabe 8.3.4 zu lösen, bietet Python eine weitere Möglichkeit zur Synchronisation an. Hierbei handelt es sich um die `RLock`-Klasse. Das R steht für `reentrant`, was Wiedereintritt bedeutet. Im Gegensatz zu Objekten der `Lock`-Klasse, die nie einem Thread zugeordnet werden, werden Objekte der `RLock`-Klasse an den Thread gebunden, der zuerst die `aquire()`-Methode aufruft. Neben den beiden Zuständen, die die `Lock`-Klasse besitzt, merkt sich die `RLock`-Klasse nun auch, wie oft die `aquire()`-Methode aufgerufen wurde. Beim ersten Aufruf von `aquire()` merkt sich das `RLock`-Objekt, welcher Thread die Methode aufgerufen hat und setzt einen internen Zähler auf 1. Bei jedem weiteren Aufruf von `aquire()` desselben Threads wird der Zähler inkrementiert. Wird `release()` aufgerufen, so wird der Zähler wieder dekrementiert. Das `RLock` ist erst dann wieder offen, wenn die Methode `release()` genau so oft aufgerufen wurde, wie `aquire()`, und der interne Zähler wieder auf 0 steht. Ruft ein zweiter Thread die `aquire()`-Methode auf, während der erste Thread das `RLOCK` besitzt, so muss er warten, bis der Zähler wieder auf 0 steht. Es ist nun also möglich, einen gewissen Codeabschnitt auch rekursiv vor konkurrierenden Zugriffen zu schützen. Wie auch schon beim `Lock`, können der `aquire()`-Methode des `RLocks` die beiden optionalen Parameter `blocking` und `timeout` übergeben werden.



Übungsaufgaben

Aufgabe 8.3.5

Passen Sie Ihr Programm aus Aufgabe 8.3.4 so an, dass der rekursive Aufruf von `increment()` nicht mehr zu einem Deadlock führt.

Im Folgenden wird das Beispiel mit dem Counter und dem Incrementer-Thread etwas angepasst. Der IncrementerThread soll nun den Counter wieder nur um eins erhöhen. Weiterhin wird dem IncrementerThread ein Wert übergeben, mit dem gesteuert wird, wann der Thread den Counter erhöht. Den 10 erstellten IncrementerThreads wird nun eine Zahl von 0 bis 9 übergeben. Der Counter soll von den einzelnen Threads immer nur dann erhöht werden, wenn der aktuelle Wert des Counters auf die Ziffer endet, die dem Thread bei der Erzeugung übergeben wurde. Demnach müssen die IncrementerThreads auf einen bestimmten geteilten Zustand warten, bevor sie increment() aufrufen dürfen. Für einen solchen Anwendungsfall stellt Python die Condition-Klasse zur Verfügung. Der Mechanismus, der hierdurch implementiert wird, ist allgemein als Condition Variable (deutsch Bedingungsvariable) bekannt. Objekte der Condition-Klasse sind immer einem Lock- oder einem RLock-Objekt zugeordnet. Dieses kann dem Konstruktor eines Condition-Objekts übergeben werden. Wird kein Lock-Objekt übergeben, erzeugt der Konstruktor ein neues. Das so erzeugte Condition-Objekt kann nun überall wie das Lock-Objekt verwendet werden, das Lock-Objekt muss nicht weiterhin zusätzlich verwaltet werden. Es kann nun also das RLock aus der Counter-Klasse gegen eine Condition Variable ausgetauscht werden. Die neue Counter-Klasse sieht dann wie folgt aus:

```
# chapters/nebenlaufigkeit/src/condition_variable.py
# Beispiel zur Verwendung von Condition-Objekten

class Counter:
    def __init__(self):
        self.count = 0
        self.cv = threading.Condition(threading.RLock())

    def increment(self, value=1):
        if 0 >= value:
            return

        with self.cv:
            self.count += 1
            self.increment(value - 1)
```

Condition-Variable

Eine Condition Variable kann also wie ein einfaches Lock verwendet werden. Die acquire()- und release()-Methoden verhalten sich hierbei wie die des hinterlegten Lock-Objekts. Darüber hinaus bietet die Condition-Klasse noch weitere Methoden an. Diese Methoden dürfen nur aufgerufen werden, wenn

zuvor `aquire()` aufgerufen wurde. Wurde von einem Thread `aquire()` aufgerufen, aber der aktuelle geteilte Zustand nicht den gewünschten Bedingungen entspricht, so wird `wait()` aufgerufen. Die `wait()`-Methode gibt das Lock wieder frei und blockiert den Thread, bis er aufgeweckt wird. Ein Thread wird durch Aufruf der `notify()`- oder der `notify_all()`-Methode aufgeweckt. Diese Methoden sollten immer dann aufgerufen werden, wenn der geteilte Zustand von einem Thread geändert wurde. Sobald ein Thread aufgeweckt wurde, fordert `wait()` wieder das Schloss an und kehrt dann zurück. Nachdem `wait()` zurückgekehrt ist, sollten die Bedingungen an den geteilten Zustand auf jeden Fall wieder geprüft werden, da eine unbestimmte Zeit zwischen dem Aufruf von `notify()` oder `notify_all()` und dem Zurückkehren von `wait()` vergehen kann. Weiterhin ist es möglich, `wait()` den optionalen Parameter `timeout` mitzugeben. Läuft diese Zeit ab, bevor ein anderer Thread `notify()` oder `notify_all()` aufruft, kehrt `wait()` mit dem Rückgabewert `False` zurück.

Tipp:

Um die Entscheidung zwischen `notify()` und `notify_all()` zu erleichtern, sollte die Frage gestellt werden, ob die Änderung des geteilten Zustands für nur einen Thread oder mehrere Threads interessant ist.

Durch einen Aufruf von `notify_all()` werden alle Threads, die `wait()` auf dem entsprechenden `Condition`-Objekt aufgerufen haben, aufgeweckt. Mit `notify()` wird nur ein Thread aufgeweckt. Es kann ein optionaler Parameter `n` an `notify()` übergeben werden, der angibt, wie viele Threads aufgeweckt werden sollen.

Achtung:

Durch den Aufruf von `notify()` und `notify_all()` wird das Lock nicht freigegeben. Das heißt, dass Threads, die `wait()` aufgerufen haben, erst dann wieder aufwachen, wenn der Thread, der `notify()` oder `notify_all()` aufgerufen hat das Lock wieder explizit frei gibt.



8.3 Ist es für das `Counter` und `IncrementerThread` Beispiel ausreichend, `notify()` aufzurufen?

Das `Counter` und `IncrementerThread` Beispiel würde nicht funktionieren, wenn immer nur ein beliebiger Thread geweckt wird. Wird der `Counter` inkrementiert, muss `notify_all()` aufgerufen werden, um einen Deadlock zu vermeiden. Dies liegt daran, dass immer nur genau ein wartender Thread

fortschreiten kann und es ist nicht garantiert werden kann, dass genau dieser Thread aufgeweckt wird.

Wie das generische Producer-Consumer-Pattern mithilfe von Condition Variablen implementiert werden kann, ist im folgenden Listing gezeigt (vgl. [?]). Die Consumer warten so lange, bis der geteilte Zustand der Bedingung entspricht. In diesem Fall heißt das, dass mindestens ein Element verfügbar ist. Sobald ein Producer ein Element erstellt hat, ruft er `notify()` auf. Somit wird genau ein Consumer aufgeweckt. In diesem Beispiel reicht es vollkommen aus, nur einen Consumer aufzuwecken.

```
# chapters/nebenlaufigkeit/src/producer_consumer.py
# Producer-Consumer-Pattern mit Condition-Variablen

# Element konsumieren
with cv:
    while not an_item_is_available():
        cv.wait()
    get_an_available_item()

# Element produzieren
with cv:
    make_an_item_available()
    cv.notify()
```

Der Consumer ruft `wait()` innerhalb der `while`-Schleife auf und prüft jedes mal seine Bedingung. Dies ist notwendig, da sich der Zustand zwischen dem Aufruf von `notify()` und dem Zurückkehren von `wait()` erneut ändern kann. Diese Problematik ist in der Multithreaded-Programmierung inhärent. Die `Condition`-Klasse bietet neben `wait()` eine weitere Methode an, die das Testen der Bedingung automatisieren kann. Bei dieser Methode handelt es sich um `wait_for()`. Ihr Parameter `predicate` nimmt ein Callable-Objekt entgegen, welches einen boolischen Wert zurück gibt. Es kann zudem ein Timeout angegeben werden, der sich wie bei `wait()` verhält. Wird `wait_for()` verwendet, ändert sich das Producer-Consumer-Pattern Beispiel folgender Maßen (vgl. [?]):

```
# chapters/nebenlaufigkeit/src/producer_consumer.py
# Producer-Consumer-Pattern mit Condition-Variablen

# Element konsumieren
with cv:
```

```
cv.wait_for(an_item_is_available)
get_an_available_item()
```



Übungsaufgaben

Aufgabe 8.3.6

Erweitern Sie das Counter und IncrementerThread Beispiel um Condition Variablen.

- a) Übernehmen Sie die Änderungen aus der angepassten Counter-Klasse in Ihre eigene.
- b) Stellen Sie sicher, dass der richtige Thread aufgeweckt wird, sobald der geteilte Zustand verändert wird.
- c) Erweitern Sie IncrementThread um ein neues Attribut digit. Dem Konstruktur soll per Parameter ein Wert für das neue Attribut übergeben werden.
- d) Ergänzen Sie in IncrementThread eine check_condition()-Methode, die True zurück gibt, wenn die letzte Ziffer des Counters dem neuen Attribut digit entspricht.
- e) Passen Sie die run()-Methode des IncrementThreads so an, dass die increment() Methode nur dann aufgerufen wird, wenn seine Bedingung erfüllt ist.

Tipp:

Sie erhalten das Condition-Objekt des Counters innerhalb der IncrementerThread-Klasse mit folgendem Aufruf: self.counter.cv

- f) Ergänzen Sie das Programm mit passenden Ausgaben, um nachzuvollziehen, welcher Thread aufgeweckt wurde und gerade inkrementiert.

Achtung:

Achten Sie darauf, dass Sie Ihren Counter nur um 1 erhöhen!

Eine weitere Problematik, die bei der Multithreaded-Programmierung kommt, ist das Schützen einer Ressource mit einer begrenzten Kapazität. Als Beispiel aus dem Alltag dient hierfür ein Parkhaus, welches nur eine be-

grenzte Anzahl an Parkplätzen besitzt. Es dürfen sich immer nur maximal so viele Autos im Parkhaus befinden, wie Parkplätze vorhanden sind. Die Klasse `CarPark` simuliert ein Parkhaus und verwendet eine Condition Variable, um sich vor konkurrierenden Zugriffen zu schützen.

```
# chapters/nebenlaufigkeit/src/parkhaus.py
# Anschauungsbeispiel: Parkhaus

class CarPark:
    def __init__(self, capacity):
        self.cv = threading.Condition()
        self.capacity = capacity
        self.occupied = 0

    def is_free(self):
        return self.occupied < self.capacity

    def enter(self):
        with self.cv:
            self.cv.wait_for(self.is_free)
            self.occupied += 1
            self.show()

    def exit(self):
        with self.cv:
            self.occupied -= 1
            self.cv.notify()
            self.show()

    def show(self):
        currentCapacity = self.capacity - self.occupied
        print("CarPool capacity is ", currentCapacity)
```

Ein Auto, dass zuerst eine gewisse Zeit umher fährt, bevor es eine zufällige Zeit parkt, wird durch die Klasse `Car` simuliert. Um eine zufällige Zeit im Intervall von `x` bis `y` zu schlafen wird hier die `uniform(x,y)`-Methode aus dem `random`-Modul verwendet.

```
# chapters/nebenlaufigkeit/src/parkhaus.py
# Anschauungsbeispiel: Parkhaus

class Car(threading.Thread):
    def __init__(self, carPark, id):
```

```

threading.Thread.__init__(self)
self.carPark = carPark
self.id = id

def run(self):
    while True:
        # Fahre eine zufällige Zeit umher
        time.sleep(random.uniform(0, 10))

        # Fahren in das Parkhaus
        print("Car", self.id, " wants to park")
        self.carPark.enter()
        print("Car", self.id, " entered the car park")

        # Parke eine zufällige Zeit
        print("Car", self.id, " is parking")
        time.sleep(random.uniform(0, 15))

        # Fahre aus dem Parkhaus
        self.carPark.exit()
        print("Car", self.id, " exited the car park")

```

Es wird nun ein neues CarPark-Objekt erzeugt, das fünf Parkplätze besitzt. Anschließend werden zehn Car-Objekte initialisiert und gestartet. Wird das folgende Programm ausgeführt, kann über die Ausgaben nachvollzogen werden, welches Auto gerade in das Parkhaus einfahren möchte, einen Parkplatz gefunden hat und wieder ausfährt.

```

# chapters/nebenlaufigkeit/src/parkhaus.py
# Anschauungsbeispiel: Parkhaus

carPark = CarPark(5)
for i in range(10):
    cars = Car(carPark, i)
    cars.start()

```

Dieses Beispiel stellt eine vereinfachte Ansicht auf das reale Geschehen. Es wird hier nämlich keine Rücksicht auf die Reihenfolge genommen, in der die Autos in das Parkhaus einfahren möchten. Demnach ist es möglich, dass Autos in der Warteschlange übersprungen werden.



8.4 Weshalb wird hier die Reihenfolge der Autos in der Warteschlange nicht beachtet?

Es ist etwas lästig, selbst über die aktuell belegten Ressourcen Buch zu führen. Durch die `Semaphore`-Klasse wird in Python eine Synchronisationsprimitive angeboten, die das Verwalten der zu schützenden Ressourcen übernimmt. Eine `Semaphore` ähnelt einem `Lock`, mit dem Unterschied, dass ihr bei der Erzeugung über den `value`-Parameter eine Kapazität übergeben werden kann. Intern verwaltet sie einen Zähler, der mit der angegebenen Kapazität initialisiert wird. Jedes mal, wenn `aquire()` aufgerufen wird, wird dieser Zähler dekrementiert. Die `aquire()`-Methode blockiert nur dann, wenn der Zähler auf 0 steht. Erst wenn `release()` aufgerufen wird, wird der Zähler wieder inkrementiert. Werden Threads durch `aquire()` blockiert, wird genau einer dieser Threads automatisch aufgeweckt, sobald `release()` aufgerufen wird. Wie auch zuvor bei `Lock` unterstützt die `aquire()`-Methode der `Semaphore`-Klasse die beiden optionalen Parameter `blocking` und `timeout`. Hierbei ist zu beachten, dass die `Semaphore`-Klasse es zulässt, `release()` öfter aufzurufen, als `aquire()`. Dies hat zur Folge, dass der interne Zähler größer anwächst, als er initial gesetzt wurde, und somit die maximale Anzahl an gleichzeitigen Zugriffen auf die geschützte Ressource erhöht wird. Ist ein solches Verhalten nicht gewünscht, bietet Python die `BoundedSemaphore`-Klasse an. Würde hier durch einen Aufruf von `release()` der interne Zähler größer werden, als der initial gesteckte Wert, so wird anstelle des Inkrementierens des Zählers ein `ValueError` geworfen.

Semaphoren

Tipp:

Um eventuelle Programmierfehler zu reduzieren, sollte `BoundedSemaphore` bevorzugt verwendet werden, da andernfalls der Fehler unbemerkt bleibt.

Wie eine `Semaphore` den gleichzeitigen Zugriff auf eine Datenbankverbindung beschränken kann ist im folgenden Beispiel gezeigt (vgl. [?]). Hier wird eine Beschränkung von maximal fünf gleichzeitigen Zugriffen gesetzt. Das `with`-Statement ruft wieder automatisch `aquire()` und `release()` auf. So mit ist garantiert, dass sich immer maximal fünf Threads innerhalb des `with`-Statements befinden.

```
# chapters/nebenlaufigkeit/src/semaphore.py
# Beispiel zur Verwendung von Semaphore-Objekten

# Initialisierung der Semaphore
```

```

maxconnections = 5
# ...
pool_sema = threading.BoundedSemaphore(value=maxconnections)

# Zugriff auf die Datenbank in den Arbeiterthreads
with pool_sema:
    conn = connectdb()
    try:
        # ... Verbindung nutzen ...
    finally:
        conn.close()

```



Übungsaufgaben

Aufgabe 8.3.7

Passen Sie die `CarPark`-Klasse so an, dass nun eine passende `Semaphore` verwendet wird. Übernehmen Sie die `Car`-Klasse und den Code zur Ausführung. Führen Sie Ihr Programm aus und überprüfen Sie, dass es korrekt funktioniert.

Tipps:

Um den aktuellen Zählerwert der `Semaphore` zu erhalten kann das Attribut `_value` verwendet werden. Dieses sollte allerdings immer nur für Debug-Zwecke eingesetzt werden!

In manchen Situationen ist es notwendig, dass bestimmte Threads aufeinander warten, bevor sie mit ihrer Ausführung fortschreiten. So kann es zum Beispiel sein, dass mit der eigentlichen Aufgabe gewartet werden muss, bis gewisse Initialisierungen geschehen sind. Es könnte auch ein Algorithmus betrachtet werden, dessen Teilschritte zwar für sich betrachtet nebenläufig ausgeführt werden können, es aber zu Fehlern kommt, wenn mit Schritt 2 begonnen wird, bevor Schritt 1 vollständig bearbeitet wurde. Für solche Anwendungsfälle bietet Python mit der `Barrier`-Klasse eine weitere Synchronisationsprimitive an. Ihr wird bei der Erzeugung über den `parties`-Parameter angegeben, wie viele Threads aufeinander warten müssen. Die einzelnen Threads rufen die `wait()`-Methode der `Barrier` auf, welche so lange blockiert, bis die mit `parties` spezifizierte Anzahl an Threads `wait()` aufgerufen haben. Wurde diese Anzahl erreicht, werden alle Threads zeit-

gleich fortgesetzt. Neben `parties` nimmt der Konstruktor von `Barrier` die beiden optionalen Parameter `timeout` und `action`. Durch `timeout` wird der Default-Timeout der `wait()`-Methode gesetzt. Läuft dieser Timeout ab, bevor alle Threads an der `Barrier` warten, werden die bereits wartenden Threads wieder aufgeweckt und die `Barrier` wird in einen „zerstörten“ Zustand gebracht. Der `wait()`-Methode kann zudem ebenfalls ein `timeout`-Parameter übergeben werden. In diesem Fall hat der in `wait()` definierte `Timeout` Vorrang. Durch den `action`-Parameter kann ein Callable-Objekt definiert werden, dass aufgerufen wird, sobald alle Threads `wait()` aufgerufen haben und bevor sie ihre Ausführung fortsetzen. Der Rückgabewert von `wait()` ist ein Integer im Intervall $[0, \text{parties}]$. Hierüber kann zum Beispiel ein Thread ausgewählt werden, um eine besondere Aufgabe auszuführen. Wurde die `Barrier` genutzt, kann sie über `reset()` in ihren Initialzustand versetzt werden und erneut verwendet werden. Durch Aufruf von `abort()` wird die `Barrier` in einen „zerstörten“ Zustand überführt. Es kann jederzeit die Anzahl der Threads, auf die gewartet wird, und die Anzahl der bereits wartenden Threads über die Attribute `parties` und `n_waiting` abgefragt werden. Das Attribut `broken` gibt an, ob sich die `Barrier` im „zerstörten“ Zustand befindet.

Tipp:

Wenn in einem Thread ein Problem auftritt, und er nicht ordnungsgemäß fortgesetzt werden kann, kann `abort()` genutzt werden, um so die bereits wartenden Threads aufzuwecken und einen Deadlock zu vermeiden.



Übungsaufgaben

Aufgabe 8.3.8

Passen Sie die `Car`-Klasse unter Verwendung von einer `Barrier` so an, dass auf das Starten aller Autos gewartet wird, bevor sie beginnen, umher zu fahren.

Thread-Kommunikation

Es gibt verschiedene Wege, eine Kommunikation zwischen verschiedenen Threads zu realisieren. In Python wurde einer der simplesten Mechanismen durch die `Event`-Klasse implementiert. Hierbei warten Threads darauf, dass ein gewisses Ereignis eintritt, welches durch einen anderen Thread ausge-

Events

löst wird. Das `Event`-Objekt verwaltet hierfür eine interne binäre Variable, welche initial den Wert `False` erhält. Sie kann durch `set()` und `clear()` gesetzt und gelöscht werden. Die `is_set()`-Methode gibt den aktuellen Wert der binären Variable zurück. Soll auf das Eintreten des Ereignisses gewartet werden, so kann die `wait()`-Methode aufgerufen werden. Diese blockiert solange, bis `set()` aufgerufen wurde. Wurde `set()` bereits vor `wait()` aufgerufen, wird nicht blockiert, und `wait()` kehrt sofort zurück. Es kann wieder der `timeout`-Parameter an `wait()` übergeben werden. Nur falls dieser Timeout ausläuft, bevor `set()` aufgerufen wurde, gibt `wait()` `False` zurück. Andernfalls ist der Rückgabewert immer `True`.



Übungsaufgaben

Aufgabe 8.3.9

In Aufgabe 8.3.8 haben Sie durch eine `Barrier` implementiert, dass die `Car`-Threads erst dann beginnen, sobald alle erstellt wurden und bereit sind. Implementieren Sie nun die selbe Funktionalität mit einem `Event` anstelle der `Barrier`.

Eine einfache boolesche Variable ist in sehr vielen Anwendungsfällen nicht als Kommunikationsgrundlage geeignet. In Python gibt es eine weitere Methode, wie Threads untereinander Informationen sicher austauschen können. Im `queue`-Modul sind drei Multi-Producer, Multi-Consumer Queues implementiert, welche die nötigen Synchronisationssemantiken realisieren. Diese drei Queues unterscheiden sich in der Reihenfolge, in welcher die hinzugefügten Elemente wieder entnommen werden. Durch die `Queue`-Klasse wird ein FIFO-Queue realisiert und `LifoQueue` handelt nach dem LIFO-Prinzip. So kann `LifoQueue` zum Beispiel auch als Stack verwendet werden. Der `PriorityQueue` hält seine hinzugefügten Elemente sortiert und gibt zuerst immer das niedrigste Element zurück. Zur Sortierung wird hierbei der Heap-Queue-Algorithmus, auch bekannt als Priority-Queue-Algorithmus, aus dem `heapq`-Modul (vgl. [?]) verwendet. Diese drei Klassen verwenden intern Lock-Objekte, weshalb Threads bei gleichzeitigen Zugriffen blockiert werden. Den Konstruktoren aller Queues kann der optionale Parameter `maxsize` übergeben werden, um die maximale Anzahl an Elementen zu setzen. Wird der Parameter nicht angegeben oder Werte kleiner 1 angegeben, so ist die Größe des Queues unbegrenzt.



8.5 Kennen Sie ein Anwendungsbeispiel, bei dem das Konzept von Queues häufig zum Einsatz kommt?

Über die `qsize()`-Methode kann die ungefähre Größe der Queues abgefragt werden. Durch `empty()` und `full()` kann entsprechend geprüft werden, ob der Queue leer oder gefüllt ist. Beide geben erwartungsgemäß einen booleschen Wert zurück. Um ein Element in den Queue einzufügen, wird `put()` aufgerufen. Der Parameter `item` nimmt hierbei das Element entgegen. Ist der Queue bereits voll, blockiert `put()` so lange, bis wieder ein Platz frei geworden ist. Durch Angabe des optionalen Parameters `timeout` kann eine maximale Zeit angegeben werden, nach welcher `put()` zurückkehrt, auch wenn kein Platz freigeworden ist. In diesem Fall würde eine `Full`-Exception geworfen werden. Wird der optionale Parameter `block` auf `False` gesetzt, so kehrt `put()` sofort zurück und wirft die `Full`-Exception, wenn in der Queue kein Platz frei ist. Es wird auch die Methode `put_nowait()` angeboten. Sie verhält sich wie ein Aufruf von `put()`, wenn `block` den Wert `False` besitzt. Um ein Element aus der Queue heraus zu holen, wird `get()` aufgerufen. Sollte zum Zeitpunkt des Aufrufs kein Element in der Queue enthalten sein, wird so lange blockiert, bis ein anderer Thread eines hinzufügt. Es kann wieder ein optionaler Parameter `timeout` angegeben werden. Läuft dieser ab, bevor ein Element in den leeren Queue eingefügt wurde, kehrt `get()` zurück und wirft die `Empty`-Exception. Wird der optionale Parameter `block` mit `False` angegeben, wirft `get()` sofort die `Empty`-Exception, falls kein Element vorhanden ist. Das selbe Verhalten tritt auch beim Aufruf der Methode `get_nowait()` auf. Neben dieser grundlegenden Funktionalität für einen Queue implementieren die drei vorgestellten Implementationen noch eine weitere Funktionalität. So kann geprüft werden, ob alle Elemente vollständig abgearbeitet wurden. Hierfür werden zwei weitere Methoden angeboten. Durch den Aufruf von `task_done()` wird den Queues mitgeteilt, dass ein mittels `get()` erhaltenes Element vollständig abgearbeitet wurde. Wird `join()` auf einem Objekt der drei Queues aufgerufen, so blockiert sie so lange, bis `task_done()` so häufig aufgerufen wurde, wie `get()`.

Tipp:

Diese Funktionalität kann zum Beispiel dazu verwendet werden, um zu warten, bis alle Dämon-Consumer-Threads ihre Elemente bearbeitet haben, bevor das Programm beendet wird.

Neben den bereits vorgestellten Queues gibt es noch eine weitere Implementierung, den `SimpleQueue`. Er stellt, wie auch `Queue`, einen FIFO-Queue dar und bietet zusätzliche Garantien an, welche auf Kosten von geringerer Funktionalität kommen. Für einen `SimpleQueue` kann keine maximale Größe de-

finiert werden, er ist somit immer unbegrenzt Groß. Aufrufe von `put()` blockieren garantiert nie. Die beiden optionalen Parameter `block` und `timeout` werden ignoriert und sind nur aufgeführt, um kompatibel zur `Queue`-Klasse zu sein. Die Funktionalität der anderen Klassen, die von `SimpleQueue` nicht unterstützt wird, ist das Warten auf die vollständige Abarbeitung aller Elemente. Die beiden Methoden `task_done()` und `join()` werden von `SimpleQueue` nicht angeboten.



Übungsaufgaben

Aufgabe 8.3.10

Im Folgenden soll die Klasse `Queue` in einem Beispiel verwendet werden.

- a) Erstellen Sie einen `Queue` und befüllen Sie ihn mit den Zahlen von 1 bis 20.
- b) Erzeugen Sie drei Dämon-Threads, welche Elemente aus dem erzeugten `Queue` entnehmen und das Quadrat der Elemente ausgeben.
- c) Nachdem die Threads gestartet wurden, soll der Main-Thread darauf warten, dass alle Elemente vollständig abgearbeitet werden, bevor er sich beendet.

8.3.3 Prozesse

Die bisher betrachteten Beispiele und Aufgabe waren nicht CPU-gebunden, weshalb die Auswirkungen des in Abschnitt 8.3.1 angesprochenen GILs nicht ersichtlich wurden. Wird nun aber das folgende Beispiel betrachtet, in dem die Anzahl der Primzahlen im Zahlenbereich 1 bis 10 Millionen gesucht wird, ist dies nicht mehr der Fall. Hierzu wird die Methode `countPrims()` betrachtet, die eine Liste an Zahlen entgegen nimmt und die Anzahl der enthaltenen Primzahlen ausgibt.

```
# chapters/nebenlaufigkeit/src/primzahlen_threads.py
# Primzahlen zählen mit Threads

def countPrims(numbers):
    count = 0
    for number in numbers:
        if 2 == number:
```

```
        count += 1
        continue
    if 2 > number or 0 == number % 2:
        continue

    for i in range(3, int(math.sqrt(number)) + 2), 2):
        if 0 == number % i:
            break
        else:
            count += 1
    print("Found ", count, " primes")
```

Es werden nun 10 Threads erzeugt, die jeweils für 1 Millionen Zahlen prüfen, ob es sich bei ihnen um eine Primzahl handelt. Zusätzlich wird die Zeit gemessen, die die Threads benötigen, um die Primzahlen zu zählen.

```
# chapters/nebenlaufigkeit/src/primzahlen_threads.py
# Primzahlen zählen mit Threads

threads = []
amount = 1000000
for i in range(10):
    numbers = [j + 1 + i * amount for j in range(amount)]
    thread = threading.Thread(target=countPrims,
                               args=(numbers,))
    threads.append(thread)

start = time.clock()
for thread in threads:
    thread.start()

for thread in threads:
    thread.join()
end = time.clock()

print("Finished in ", end - start, " seconds")
```

Wird diese Programm ausgeführt, kann eine mögliche Ausgabe wie folgt aussehen:

```
Found 67883 primes
Found 78498 primes
Found 70435 primes
```

```
Found  63799  primes
Found  65367  primes
Found  64336  primes
Found  66330  primes
Found  62712  primes
Found  62090  primes
Found  63129  primes
Finished in  102.9065528  seconds
```

Um nun den Effekt des GILs zu zeigen, wird die selbe Aufgabe durch mehrere Prozesse gelöst. Hierzu bietet Python im `multiprocessing`-Modul die Klasse `Process` an. Die API, mit der neue Prozesse in Python erzeugt und gestartet werden, ähnelt der des `threading`-Moduls. Demnach muss der Code nur minimal angepasst werden, um Primzahlen von Prozessen zählen zu lassen:

```
# chapters/nebenlaufigkeit/src/primzahlen_prozesse.py
# Primzahlen zählen mit Prozessen

if __name__ == "__main__":
    processes = []
    amount = 1000000
    for i in range(10):
        numbers = [j + 1 + i * amount for j in range(amount)]
        process = multiprocessing.Process(target=countPrims,
                                         args=(numbers,))
        processes.append(process)

    start = time.clock()
    for process in processes:
        process.start()

    for process in processes:
        process.join()
    end = time.clock()

    print("Finished in ", end - start, " seconds")
```

Wird dieses Programm ebenfalls ausgeführt, ist es möglich, die Primzahlen schneller zu zählen. Eine mögliche Ausgabe kann wie folgt aussehen:

```
Found  78498  primes
Found  70435  primes
```

```
Found 67883 primes
Found 65367 primes
Found 66330 primes
Found 64336 primes
Found 63129 primes
Found 63799 primes
Found 62712 primes
Found 62090 primes
Finished in 29.6911255 seconds
```

Beide Varianten erhalten das selbe Ergebniss, allerdings sind die Prozesse knapp 70 Sekunden schneller. Ein größerer unterschied der beiden Varianten ist die Zeile `if __name__ == '__main__'`. Diese Zeile ist im Prozess-Beispiel notwendig, da der Programmcode innerhalb des `if`-Statements ansonsten jedesmal ausgeführt wird, wenn das Modul importiert wird. Wird ein neuer Prozess gestartet, lädt der neue Python-Interpreter das Modul ein zweites mal. Sobald er nun die Codezeile erreicht, in der der neue Prozess gestartet wurde, wird ein dritter Prozesse gestartet, dessen Python-Interpreter das Modul ein drittes mal lädt. Somit werden solange Prozesse erzeugt, bis das System keine Ressourcen mehr zur Verfügung hat. Durch Angabe des `if`-Statements ist es garantiert, dass der enthaltene Code nur einmal ausgeführt wird, wenn das Programm gestartet wird.

Prozess Objekte

Wie aus dem betrachteten Prozess-Beispiel erkenntlich ist, ist die Verwendung der `Process`-Klasse sehr stark an die der `Thread`-Klasse angelehnt. Genau genommen gibt es jede Methode von `Thread` auch in `Process`. Sogar die Konstruktoren sind identisch. Der `group`-Parameter des `Process`-Konstruktors existiert allerdings nur zur Kompatibilität zum Konstruktor der `Thread`-Klasse. Der Aufruf von `join()` gibt immer `None` zurück, auch wenn der optionale Timeout abgelaufen ist. Um zu prüfen, ob der entsprechende Prozess tatsächlich beendet wurde, ist über seinen `exitcode` einsehbar. Es ist zu bemerken, dass ein Dämon-Prozess keine weiteren Kindprozesse starten kann. Sobald sich sein Elternprozess beendet, wird er terminiert. Zusätzlich bietet die `Process`-Klasse noch weitere Attribute und Methoden an. Über das `pid`-Attribut kann die ID des jeweiligen Prozesses abgefragt werden. Durch `exitcode` kann abgefragt werden, mit welchem Status sich der Prozess beendet hat. Wurde er noch nicht beendet, hat `exitcode` den Wert `None`. Trägt `exitcode` einen negativen Wert, so bedeutet das, dass der Pro-

zess durch ein Signal beendet wurde. Ein Wert von `-N` entspricht dann dem Signal `N`. Beim Attribut `authkey` handelt es sich um einen Byte-String, welcher für gewisse Authentifizierungen genutzt wird und standardmäßig den Wert des Elternprozesses übernimmt. Genauere Informationen zur Verwendung von `authkey` sind in [?] zu finden. Die beiden Methoden `kill()` und `terminate()` beenden den jeweiligen Prozess, nicht aber dessen Kindprozesse. Verwendet der Prozess Locks, Semaphoren, Queues oder Pipes, so kann ein Aufruf der beiden Methoden dazu führen, dass die verwendeten Objekte unnutzbar werden und andere Prozesse in einen Deadlock geraten. Unter Windows wird zum Beenden der Prozesse `TerminateProcess()` aufgerufen. Unter Unix-Systemen sendet die Methode `terminate()` das `SIGTERM` Signal, während `kill()` das Signal `SIGKILL` sendet. Durch Aufruf der `close()`-Methode werden alle Ressourcen des jeweiligen Prozesses freigegeben, falls er bereits beendet ist. Andernfalls wird ein `ValueError` geworfen. Nachdem `close()` erfolgreich zurückgekehrt ist, wird beim Zugriff auf die meisten Methoden und Attributen von `Process` ebenfalls ein `ValueError` geworfen.

Achtung:

Die Methoden `start()`, `join()`, `is_alive()`, `terminate()` und das Attribut `exitcode` sollten immer nur vom erzeugenden Prozess verwendet werden.

Das `multiprocessing`-Modul unterstützt, je nach Betriebssystem, drei verschiedene Arten einen Prozess zu starten. Bei diesen drei Arten handelt es sich um `spawn`, `fork` und `forkserver`, welche sich folgendermaßen unterscheiden:

a) `spawn`:

Diese Art der Prozesserzeugung starten einen neuen Python Interpreter. Es werden nur diejenigen Ressourcen an den neuen Prozess vererbt, die zum Ausführen seiner `run()`-Methode nötig sind. Im Vergleich zu den beiden anderen Varianten ist diese eher langsam. Sie ist unter Unix und Windows Systemen verfügbar und der Standard unter Windows.

b) `fork`:

Durch diese Startmethode wird ein Fork des aktuellen Python Interpreters durch den Aufruf von `os.fork()` erstellt. Das heißt, dass der erzeugte Kindprozess effektiv identisch zum Elternprozess ist. Alle Ressourcen werden vom Elternprozess geerbt. Erzeugen Multithreaded-Prozesse mit

dieser Startmethode Kindprozesse, kann es sehr schnell problematisch werden. Unter Windows ist diese Startmethode nicht verfügbar, unter Unix Systemen ist sie die Standardvariante.

c) `forkserver`:

Wurde beim Starten des Programms diese Variante zum Starten von Prozessen gewählt, wird ein Server gestartet. Immer wenn ein neuer Prozess erzeugt werden soll, verbindet sich der erzeugende Prozess mit diesem Server und fordert an, einen Fork erstellen zu lassen. Hierbei werden nur die notwendigen Ressourcen an den Kindprozess vererbt. Da es sich bei dem Fork-Server um einen Single-Threaded-Prozess handelt, ist ein Aufruf von `os.fork()` unproblematisch. Diese Variante wird nur von Unix Systemen unterstützt, die auch das Übergeben von Dateideskriptoren über Unix-Pipes unterstützen.

Um eine der drei Startmethoden zu wählen, kann die `set_start_method()`-Methode aufgerufen werden. Sie sollte nur innerhalb von `if __name__ == '__main__'` aufgerufen werden und nie mehrmals in einem Programm. Die `spawn` Startmethode wird also wie im folgenden Beispiel gezeigt ausgewählt.

```
# chapters/nebenlaufigkeit/src/prozess_start_methode.py
# Wahl einer Prozess-Startmethode

if __name__ == "__main__":
    multiprocessing.set_start_method("spawn")
    process = multiprocessing.Process(target=foo)
    process.start()
    process.join()
```

Sollen mehrere Prozesse mit unterschiedlichen Startmethoden gestartet werden, so kann alternativ `get_context()` aufgerufen werden, um ein `Context`-Objekt zu erhalten. Hierbei ist darauf zu achten, dass Objekte, welche mit einem `Context` erzeugt wurden, nicht immer kompatibel mit Prozessen sind, welche mit einem anderen `Context` gestartet wurden. So ist ein `Lock`-Objekt, welches mit einem `fork`-Context erzeugt wurde, nicht mit Prozessen kompatibel, die mittels der `spawn` oder der `forkserver` Startmethode erzeugt wurden. Im folgenden Beispiel ist gezeigt, wie ein Prozess mit einer bestimmten Startmethode durch ein `Context`-Objekt erzeugt wird.

```
# chapters/nebenlaufigkeit/src/prozess_start_methode.py
# Wahl einer Prozess-Startmethode
```

```
if __name__ == "__main__":
    context = multiprocessing.get_context("spawn")
    process = context.Process(target=foo)
    process.start()
    process.join()
```



Übungsaufgaben

Aufgabe 8.3.11

Schreiben Sie eine Klasse, die von `Process` erbt. Dieser eigene Prozess soll seinen Namen und seine PID ausgeben und sich anschließend beenden. Starten Sie 10 dieser Prozesse mit der `spawn` Startmethode.

Häufig wird eine Aufgabe in kleinere Schritte unterteilt und dann an mehrere Arbeiterprozesse aufgeteilt. Ein Beispiel hierfür ist das vorherige Zählen von Primzahlen. Für solche Fälle existiert in Python die `Pool`-Klasse, welche das Auslagern von Aufgaben an Arbeiterprozesse erleichtert. Wird ein neuer `Pool` erzeugt, kann seinem Konstruktor über den Parameter `processes` die Anzahl der Arbeiterthreads mitgegeben werden. Werden die Parameter `initializer` und `initargs` angegeben, so wird das Callable-Objekt, welches an `initializer` übergeben wurde, von jedem der Arbeiterprozesse mit `initargs` als Parameter aufgerufen, wenn die Arbeiterprozesse gestartet werden. Durch `maxtasksperchild` wird die maximale Anzahl an Aufgaben, welche die Arbeiterprozesse abarbeiten, angegeben, bevor sie beendet und von einem neuen Arbeiterprozess ersetzt werden. Dies hat zur Folge, dass ungenutzte Ressourcen wieder freigegeben werden. Der letzte Parameter ist `context`. Wird er angegeben, so werden die Arbeiterprozesse mit dem angegebenen `Context`-Objekt erzeugt. Um den erzeugten Arbeiterprozessen Aufgaben zu übergeben, bietet `Pool` verschiedene Methoden an. Diese sollten immer nur von dem Prozess aufgerufen werden, der auch den `Pool` erzeugt hat. Durch `apply()` wird von einem der Arbeiterprozesse das Callable-Objekt, welches über den Parameter `func` angegeben wird, mit den Parametern, welche durch `args` angegeben wurden, aufgerufen. Es ist auch möglich Schlüsselwort-Parameter über `kwds` anzugeben. Der Aufruf von `apply` blockiert, bis die Aufgabe abgearbeitet wurde. Die beiden Methoden `map()` und `imap()` nehmen ebenfalls ein Callable-Objekt mit dem `func` Parameter entgegen. Durch den Parameter `iterable` kann ein Iterable-Objekt übergeben werden, welches aufgeteilt wird und den Arbeiterprozes-

sen als separate Aufgaben übergeben wird. Der Aufruf der beiden Methoden blockiert, bis die Aufgaben abgearbeitet wurden. Bei sehr langen Iterable-Objekten ist `imap()` effizienter. Ist die Reihenfolge der Ergebnisse irrelevant, kann auch `imap_unordered()` genutzt werden, welche sich andernfalls genau wie `imap()` verhält. Benötigt eine Aufgabe eine Parameterlist, welche größer 1 ist, kann `starmap()` angewendet werden. Hierbei wird davon ausgegangen, dass die Elemente des Iterable-Objektes, welches an `iterable` übergeben wird, ebenfalls Iterable-Objekte sind. Diese werden als Parameter für das Callable-Objekte in `func` entpackt. Wie die ersten 10 Quadratzahlen mit einem Pool aus 5 Prozessen berechnet werden können, wird im folgenden Beispiel gezeigt.

```
# chapters/nebenlaufigkeit/src/prozess_pool.py
# Beispiel zu Prozess-Pools

def square(x):
    return x * x

if __name__ == "__main__":
    with multiprocessing.Pool(5) as p:
        results = p.map(square, range(1, 11))
        print(results)
```

Die Methoden `apply_async()`, `map_async()` und `startmap_async()` verhalten sich wie ihre normalen Varianten, aber sie blockieren nicht. Sie geben ein `AsyncResult`-Objekt zurück, über das das Ergebnis der Aufgaben erhalten werden kann, sobald es verfügbar ist. Die Methode `ready()` gibt an, ob die Aufgaben bereits abgearbeitet wurden. Durch `successful()` kann erfahren werden, ob eine Exception während dem Bearbeiten der Aufgaben geworfen wurde. Wurden noch nicht alle Aufgaben vollständig bearbeitet, wird ein `AssertionError` geworfen. Soll darauf gewartet werden, dass alle Aufgaben abgearbeitet wurden, kann `wait()` aufgerufen werden. Um das Ergebnis der Aufgaben zu erhalten, kann `get()` aufgerufen werden. Für `wait()` und `get()` kann ein optionaler Timeout angegeben werden. Wurden alle Aufgaben an den Pool übergeben, kann `close()` aufgerufen werden. Nachdem diese Methode aufgerufen wurde, kann dem Pool keine neue Aufgabe übergeben werden. Sobald alle Aufgaben abgearbeitet wurden, werden die Arbeiterprozesse beendet. Durch den Aufruf von `terminate()` werden die Arbeiterprozesse sofort beendet, ohne dass sie ihre Aufgaben fertig bearbeiten. Nachdem einer dieser beiden Methoden aufgerufen wurde, kann

durch `join()` darauf gewartet werden, dass sich alle Arbeiterprozesse beenden.



Übungsaufgaben

Aufgabe 8.3.12

Implementieren Sie das Zählen von Primzahlen aus dem besprochenen Beispiel unter Verwendung eines Pools.

Prozess-Kommunikation

Um Daten zwischen Prozessen auszutauschen bietet Python die `Pipe`-Funktion und die `Queue`-Klasse an. Dieser `Queue` aus dem `multiprocessing`-Modul ist dem bereits bekannten `Queue` aus dem `queue`-Modul sehr ähnlich. Intern wird eine Pipe genutzt, welche durch Locks und Semaphoren geschützt wird. Wird von einem Prozess zum ersten Mal ein Element hinzugefügt, wird ein neuer Thread gestartet, der das Element aus einem Puffer in die Pipe schreibt. Die `Queue`-Klasse implementiert alle bekannten Methoden, bis auf `join()` und `task_done()`. Darüber hinaus werden auch neue Methoden implementiert. Durch `close()` wird der `Queue` signalisiert, dass der aufrufende Prozesse keine neuen Daten mehr hinzufügt. Sobald alle Elemente aus dem Puffer in die Pipe geschrieben wurden, beendet sich der Hintergrundthread. Durch den Aufruf von `join_thread()` kann auf dieses Ereignis gewartet werden. Es kann auch `cancel_join_thread()` aufgerufen werden, wenn der Prozess sofort beendet werden soll, ohne dass die Elemente aus dem Puffer in die Pipe geschrieben werden. Neben `Queue` wird auch ein `SimpleQueue` angeboten. Er besitzt nur drei Methoden, `empty()`, `get()` und `put()`, welche alle selbsterklärend sind. Darüber hinaus wird die `JoinableQueue`-Klasse angeboten. Sie erbt von `Queue` und erweitert diese Funktionalität um die beiden Methoden `task_done()` und `join()`, welche sich wie die bereits bekannten Methoden verhalten. Die `Pipe()`-Funktion gibt zwei `Connection`-Objekte zurück, welche über eine Pipe verbunden sind. Jedes Ende der Pipe wird durch eines der beiden `Connection`-Objekte repräsentiert. Sollten zwei Prozesse oder Threads gleichzeitig versuchen von einem Ende der Pipe zu lesen oder hinein zu schreiben, so können die Daten in der Pipe korruptiert werden. Die beiden `Connection`-Objekte bieten die `send()`-Methode an, welche das übergebene Objekt an das andere Ende der `Connection` überträgt. Wird auf der anderen Seite `recv()` aufgerufen, wird das übertragene Objekt empfangen. Wenn noch keine Objekt gesen-

Queues

Pipes

det wurde, blockiert `recv()` bis ein Objekt zum Empfang bereit ist. Um eine Connection zu schließen, muss `close()` aufgerufen werden. Die Methode `poll()` gibt an, ob aktuell Daten zum Empfangen bereit sind. Es ist möglich, einen optionalen Timeout anzugeben. Die `poll()`-Methode blockiert dann maximal die spezifizierte Zeit in Sekunden. Wurde kein Timeout angegeben, kehrt die Methode sofort zurück. Sollen Byte-Daten versendet werden, kann die Methode `send_bytes()` aufgerufen werden. Sie nimmt ein Objekt entgegen, das das Buffer Protokoll (vgl. [?]) unterstützt, und einen optionalen Offset. Durch `recv_bytes()` kann dieses Objekt empfangen werden. Sollte aktuell nichts zu empfangen sein, blockiert `recv_bytes()`. Es kann optional eine maximale Länge über `maxlength` angegeben werden.

Achtung:

Sollte die empfangene Nachricht allerdings größer sein als die angegebene Länge, wird ein `OSError` geworfen und von der Connection kann nicht weiter gelesen werden.

Alternativ kann `recv_bytes_into()` genutzt werden. Diese Methode gibt die Anzahl der empfangenen Bytes zurück und schreibt sie in das durch den Parameter `buffer` angegebene Objekt. Es kann zudem ein optionaler Parameter `offset` angegeben werden, um die Startposition im Buffer zu definieren. Sollten keine Daten empfangsbereit sein, blockiert `recv_bytes_into()`.



Übungsaufgaben

Aufgabe 8.3.13

Implementieren Sie Aufgabe 8.3.10 mit Prozessen, anstelle von Threads. Es sollen nun 3 Prozesse gestartet werden, die die ersten 20 Quadratzahlen berechnen. Die Prozesse erhalten ihre aktuellen Zahlen über einen Queue.

