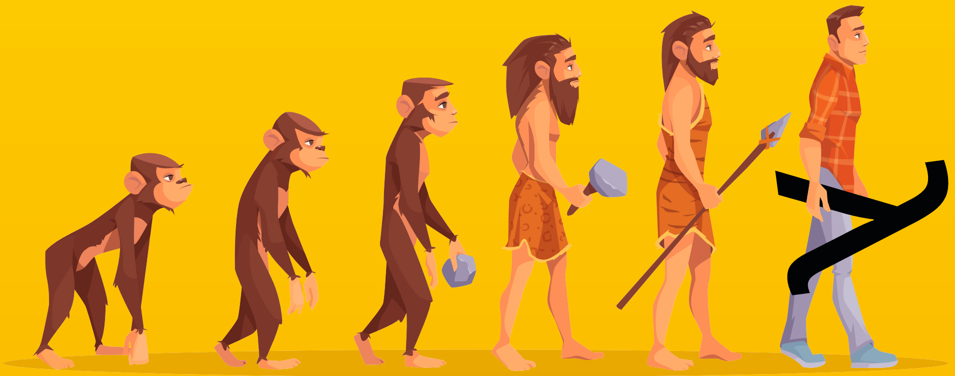

Bernd KLEIN
Philip KLEIN

FUNKTIONALE PROGRAMMIERUNG MIT PYTHON

HANSER

Bernd Klein, Philip Klein

Funktionale Programmierung mit Python

Bleiben Sie auf dem Laufenden!

Unser **Computerbuch-Newsletter** informiert Sie monatlich über neue Bücher und Termine. Profitieren Sie auch von Gewinnspielen und exklusiven Leseproben. Gleich anmelden unter:

www.hanser-fachbuch.de/newsletter

Bernd Klein, Philip Klein

Funktionale Programmierung mit Python

Die Autoren:
Bernd Klein, Singen
Philip Klein, Freiburg

Print-ISBN: 978-3-446-48191-6
E-Book-ISBN: 978-3-446-48200-5
E-Pub-ISBN: 978-3-446-48379-8

Alle in diesem Werk enthaltenen Informationen, Verfahren und Darstellungen wurden nach bestem Wissen erstellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Werk enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autor und Verlag übernehmen infolgedessen keine Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieses Programm-Materials – oder Teilen davon – entsteht. Ebenso übernehmen Autoren und Verlag keine Gewähr dafür, dass beschriebene Verfahren usw. frei von Schutzrechten Dritter sind.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt also auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Dieses Werk ist urheberrechtlich geschützt.
Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren), auch nicht für Zwecke der Unterrichtsgestaltung – mit Ausnahme der in den §§ 53, 54 URG genannten Sonderfälle –, reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Wir behalten uns auch eine Nutzung des Werks für Zwecke des Text- und Data Mining nach § 44b UrhG ausdrücklich vor.

Aus Gründen der besseren Lesbarkeit wird auf die gleichzeitige Verwendung der Sprachformen männlich, weiblich und divers (m/w/d) verzichtet. Sämtliche Personenbezeichnungen gelten gleichermaßen für alle Geschlechter.

© 2025 Carl Hanser Verlag GmbH & Co. KG, München
Kolbergerstraße 22 | 81679 München | info@ hanser.de
www.hanser-fachbuch.de

Lektorat: Brigitte Bauer-Schiewek
Copy editing: Jürgen Dubau, Freiburg
Layout: die Autoren mit LaTeX
Herstellung: le-tex publishing services GmbH, Leipzig
Coverkonzept: Marc Müller-Bremer, www.rebranding.de, München
Covergestaltung: Thomas West
Titelmotiv: © vectorpouch / Shutterstock
Druck: CPI Books GmbH, Leck
Printed in Germany

Inhalt

Teil I Einleitung **1**

1 Vorwort **3**

2 Danksagung **5**

3 Einleitung **7**
3.1 Einführung 7
3.2 Zielsetzung des Buches 7
3.3 Aufbau des Buches 8
3.4 Leserschaft 10
3.5 Zusätzliche Unterlagen 10

Teil II Python-Grundlagen unter funktionalen Aspekten **11**

4 Variablen und Datentypen **13**
4.1 Variablen 13
4.1.1 Gültige Variablennamen 15
4.1.2 Konventionen für Variablennamen 15
4.2 Übersicht Datenstrukturen 16
4.2.1 Unveränderliche (immutable) Datentypen 16
4.2.2 Veränderliche (mutable) Datentypen 16
4.3 Datenstrukturen im Detail 17
4.3.1 Integer 17
4.3.2 Floats 17

4.3.3 Zeichenketten oder Strings ... 18
4.3.4 Bytesequenz ... 20
4.3.5 Listen ... 21
4.3.6 Mengen ... 22
4.3.7 Dictionaries ... 23

5 Kontrollstrukturen ... 25
5.1 Sequenz ... 26
5.2 Bedingte Anweisungen ... 26
5.2.1 Vollständiges if ... 26
5.2.2 Ternäres if ... 28
5.3 Schleifen ... 29
5.3.1 while-Schleife ... 29
5.3.1.1 Allgemeine Arbeitsweise ... 29
5.3.1.2 while-Schleife mit else ... 30
5.3.2 while-Schleife als rekursive Funktion ... 31
5.3.3 Allgemeine while-Funktion ... 33
5.3.4 for-Schleife ... 34
5.3.5 for-Schleife in funktionaler Programmierung ... 36

6 Das Modul collections ... 37
6.1 Übersicht ... 37
6.2 `namedtuple` ... 38
6.3 `Deque` ... 41
6.4 `ChainMap` ... 42
6.5 `Counter` ... 44

Teil III Funktionale Programmierung ... 47

7 Begriffsbestimmung ... 49
7.1 Einführung ... 49
7.2 Verschiedene Programmierparadigmen ... 50
7.3 Funktionale Programmierung ... 50

8 Funktionen ... 53
8.1 Einleitung ... 53
8.2 Funktionen ... 53
8.2.1 Einfache Funktionen ... 53

8.2.2 Typehints 55
8.2.2.1 Einführende Beispiele 55
8.2.2.2 Typehints mit mypy 56
8.2.3 Default-Parameter und Schlüsselwortparameter 57
8.2.4 Weitere Typehints 57
8.2.5 Lokale Funktionen 58
8.2.6 Globale und lokale Variablen in Funktionen 59
8.2.7 Gültigkeit von Variablen in verschachtelten Funktionen 60
8.3 Dataclasses und Pattern Matching 63
8.3.1 Dataclasses 63
8.3.1.1 Standardargumente und Fabrikfunktionen 64
8.3.1.2 Unterschiede zum namedtuple 66
8.3.2 Pattern Matching 69
8.3.2.1 Einführung 69
8.3.2.2 Binärbäume mit Pattern Matching durchlaufen 69
8.4 Aufgaben 70
8.5 lambda 71
8.5.1 lambda mit sorted 73
8.6 Aufgaben 74
8.7 Rekursive Funktionen 75
8.7.1 Einführung 75
8.7.2 Definition der Rekursion 76
8.7.3 Iterative Lösung im Vergleich zur rekursiven Lösung 77
8.7.4 Kategorien der Rekursion 77
8.7.4.1 Lineare Rekursion 78
8.7.4.2 Gegenseitige/Wechselseitige Rekursion 79
8.7.4.3 Baumartige Rekursion 80
8.7.4.4 Endrekursion 85
8.8 Aufgaben 85
8.9 Funktionen als Erste-Klasse-Objekte 87
8.9.1 Definition 87
8.9.2 Zuweisung an Variable 88
8.9.3 Funktionen in Datenstrukturen 89
8.9.4 Funktionen als Argumente 92
8.9.5 Funktionen innerhalb von Funktionen 93
8.9.5.1 Kapselung und Verbergen 93
8.9.5.2 Fabrikfunktionen 95
8.10 Aufgaben 98

9 Dekoratoren ... 99
9.1 Einführung Dekoratoren ... 99
9.2 Ein einfacher Dekorator ... 100
9.3 @-Syntax für Dekoratoren ... 101
9.4 Dekoratoren für beliebige Signaturen ... 102
9.5 Anwendungsfälle für Dekoratoren ... 103
9.5.1 Erweiterung von Funktionen ... 104
9.5.2 Logging-Dekorator ... 105
9.5.3 Authentifizierung und Autorisierung ... 106
9.5.4 Validierung: Überprüfung von Argumenten ... 107
9.5.5 Profiling: Funktionsaufrufe mit einem Dekorator zählen ... 108
9.6 Dekoratoren mit Parametern ... 109
9.6.1 Einführendes Beispiel ... 109
9.6.2 Weiteres Beispiel ... 110
9.7 Benutzung von Wraps aus functools ... 112
9.8 Mehrfache Dekoration einer Funktion ... 114
9.8.1 Veranschaulichung ... 114
9.8.2 Python-Beispiel ... 115
9.8.3 Praktisches Beispiel einer Mehrfachdekoration ... 116
9.9 Eine Klasse als Dekorator benutzen ... 119
9.10 Klassendekoration ... 120
9.11 Dekorator-Aufgaben ... 123

10 Memoisation ... 125
10.1 Bedeutung und Herkunft des Begriffs ... 125
10.2 Memoisation mit Dekoratorfunktionen ... 126
10.3 Memoisation mit einer Klasse ... 129
10.4 Memoisation mit functools.lru_cache ... 129
10.5 Aufgaben zur Memoisation ... 131

11 Closures ... 133
11.1 Einleitung ... 133
11.1.1 Definition ... 134
11.1.2 Praktische Anwendungen von Closures ... 136
11.2 Aufgaben ... 136

12 Komposition von Funktionen 137
12.1 Einführung 137
12.2 Funktionskomposition in Python 138
12.3 Komposition mit beliebiger Argumentenzahl 140
12.4 Komposition einer beliebigen Anzahl von Funktionen 141
12.5 Aufgaben 142

13 Currying in Python 143
13.1 Einführung 143
13.1.1 Zugrundeliegende Idee 143
13.1.2 Herkunft des Names 144
13.2 Currying von Funktionen 144
13.2.1 Definition und Beispiel 144
13.2.2 BMI als Beispiel 145
13.2.3 BMI als Beispiel 145
13.3 Benutzung von partial 146
13.4 Praktisches Beispiel für Currying: E-Mail-Vorlagen 146
13.5 Dekorator für Currying 148
13.6 Weitere Beispiele 150
13.6.1 Arithmetische Operatoren 150
13.6.2 Beispiel aus der Finanzwelt 151
13.6.3 Currying zur Listenfilterung 154
13.7 Currying-Funktion mit einer beliebigen Anzahl von Parametern 154
13.8 Aufgaben 157

14 Funktionale Emulation von OOP-Konzepten 159
14.1 Einführung 159
14.2 Imitation einer Klasse durch eine Funktion 160
14.2.1 Erstes Beispiel: Geradengenerierung 160
14.2.2 Private Attribute 161
14.2.3 Klasse mit Gettern und Settern als Funktion 162
14.2.4 Nachahmung von Vererbung 165
14.2.5 Überlagern von Funktionen 169
14.3 Aufgaben 171

15 Generatoren und Iteratoren 175
15.1 Definitionen 175
15.2 Einführung 175
15.2.1 Iterierbar 176
15.2.2 Iterator und Arbeitsweise der for-Schleife 176
15.3 Eigenen Iterator erzeugen 179
15.3.1 Beispiel einer Iteratorklasse 179
15.4 Generatoren und Generatorfunktionen 180
15.4.1 Unendliche Generatoren 184
15.5 Endlos-Generatoren zähmen mit firstn und islice 185
15.6 Beispiele aus der Kombinatorik 188
15.6.1 Permutationen 188
15.6.2 Variationen und Kombinationen 189
15.7 Generator-Ausdrücke 191
15.8 yield from 192
15.9 return-Anweisungen in Generatoren 193
15.10 send-Methode 196
15.10.1 Arbeitsweise 196
15.10.2 Umprogrammierung mittels send 198
15.10.3 Radio-Beispiel mit send 199
15.11 Die close-Methode 201
15.12 Die throw-Methode 202
15.13 Dekoration von Generatoren 206
15.14 Aufgaben 207

16 Iteratoren der Standardbibliothek 211
16.1 Einführung 211
16.2 Wichtige Iteratoren und iteratorähnliche Funktionen in der Python-Standardbibliothek 212
16.3 `enumerate` 212
16.4 map, filter und reduce 214
16.4.1 map 214
16.4.2 Filtern von sequentiellen Datentypen mittels „filter“ 215
16.4.3 reduce 216
16.4.4 Zusammenspiel von map, filter und reduce 218
16.5 Listen-Abstraktion 218
16.5.1 Alternative zu map und filter 218
16.5.2 Syntax 219

16.5.3 Weitere Beispiele 220
16.5.4 Die zugrunde liegende Idee 220
16.5.5 Anspruchsvolleres Beispiel 221
16.5.6 Mengen-Abstraktion 222
16.5.7 Generatoren-Abstraktion 222
16.5.8 map und filter oder Listen-Abstraktion 223
16.5.9 Dict-Komprehension in Python 223
16.6 reversed 224
16.7 Die zip-Funktion 224
16.8 zip in Kombination mit map und filter 226
16.9 Aufgaben 227

17 Das Modul itertools 229
17.1 Übersicht 229
17.2 Unendliche Iteratoren 230
17.2.1 itertools.count 231
17.2.1.1 Beispiel: Teilbarkeit 232
17.2.1.2 Beispiel: Quadratpolynome 232
17.2.1.3 Beispiel: Bestellungsmanager 233
17.2.2 chain und chain.from_iterable 234
17.2.3 cycle 237
17.2.4 tee 237
17.2.4.1 Effiziente Implementierung von cycle 238
17.2.4.2 Beispiel: Aufgaben und Mitarbeiter 239
17.2.5 repeat 240
17.3 Iteratoren über endliche Sequenzen 240
17.3.1 slice und islice 241
17.3.1.1 slice 241
17.3.1.2 islice 241
17.3.1.3 slice und islice im Vergleich 243
17.3.2 accumulate 244
17.3.2.1 Praktisches Börsenbeispiel 244
17.3.3 compress 246
17.3.3.1 Einführung 246
17.3.3.2 Umfangreiches Beispiel 247
17.3.4 dropwhile 248
17.3.5 takewhile 249
17.3.6 filterfalse 251

17.3.7 groupby 252
17.3.8 pairwise 254
17.3.9 n_grams und map_n 256
17.3.10 zip_longest 258
17.3.11 starmap 258
17.4 Kombinatorische Generatoren 260
17.4.1 product 260
17.4.2 permutations 262
17.4.3 combinations 265
17.4.3.1 Arbeitsweise 265
17.4.3.2 Beispiel: Speisekartengenerierung 265
17.4.3.3 Beispiel: Benachbarte Kantengraphen 268
17.4.4 combinations_with_replacement 269
17.4.5 Umfangreiches praktisches Beispiel: csv-Datei lesen 270
17.5 Aufgaben 274

Teil IV Lösungen zu den Aufgaben 279

18 Lösungen zu den Aufgaben 281
18.1 Lösungen zu Abschnitt 8.9 (Funktionen als Erste-Klasse-Objekte) 281
18.2 Lösungen zu Abschnitt 8.3 (Dataclasses und Pattern Matching) 283
18.3 Lösungen zu Abschnitt 8.5 (lambda) 284
18.4 Lösungen zu Abschnitt 8.7 (Rekursive Funktionen) 285
18.5 Lösungen zu Abschnitt 8.9 (Funktionen als Erste-Klasse-Objekte) 289
18.6 Lösungen zu Kapitel 9 (Dekoratoren) 290
18.7 Lösungen zu Kapitel 10 (Memoisation) 298
18.8 Lösungen zu Kapitel 11 (Closures) 303
18.9 Lösungen zu Kapitel 16 (Iteratoren der Standardbibliothek) 304
18.10 Lösungen zu Abschnitt 16.5 (Listen-Abstraktion) 306
18.11 Lösungen zu Kapitel 15 (Generatoren und Iteratoren) 308
18.12 Lösungen zu Kapitel 17 (Das Modul itertools) 316
18.13 Lösungen zu Kapitel 12 (Komposition von Funktionen) 331
18.14 Lösungen zu Kapitel 13 (Currying in Python) 334
18.15 Lösungen zu Kapitel 14 (Funktionale Emulation von OOP-Konzepten) 335

Stichwortverzeichnis 341

TEIL I

Einleitung

1 Vorwort

Willkommen zu unserem Buch über funktionale Programmierung in Python. In einer Welt, die zunehmend von datengetriebenen Entscheidungen und komplexen Systemen geprägt ist, wird die Fähigkeit immer wichtiger, robuste und elegante Software zu entwickeln. Dieses Buch bietet eine fundierte Einführung in die funktionale Programmierung unter Python, ein Paradigma, das eine andere Herangehensweise an das Programmieren bietet und helfen kann, Probleme auf eine klarere und effizientere Weise zu lösen.

Obwohl Python keine rein funktionale Sprache ist, bietet sie eine Vielzahl von funktionalen Werkzeugen und Konzepten, die es wert sind, erkundet zu werden. Dieses Buch ist darauf ausgelegt, die Prinzipien und die Möglichkeiten der funktionalen Programmierung unter Python aufzuzeigen und zu demonstrieren, wie man diese Prinzipien effektiv in der Praxis anwenden kann. Neben theoretischen Erklärungen legen wir großen Wert auf praktische Anwendungen, um das Gelernte direkt umzusetzen. Jedes Kapitel enthält praktische Beispiele und Aufgaben, um die erlernten Fähigkeiten weiterzuentwickeln und die Kenntnisse zu vertiefen.

Dieses Buch richtet sich sowohl an erfahrene Programmierer aus anderen Programmiersprachen, die ihre Kenntnisse in Python und funktionaler Programmierung erweitern möchten, als auch an Anfänger mit geringen Kenntnissen in Python, die ein neues Paradigma kennenlernen wollen.

Viel Erfolg und Freude beim Lesen des Buches und beim Ausprobieren der zahlreichen Beispiele und Aufgaben!

2 Danksagung

Ein Buch entsteht über einen langen Zeitraum und ist das Ergebnis der gemeinsamen Anstrengung vieler engagierter Menschen. Neben den Autoren, die den wesentlichen Beitrag leisten, sind auch zahlreiche andere Personen direkt oder indirekt an der Entstehung beteiligt. Unser besonderer Dank gilt Karola und Melisa, die die zusätzliche Belastung mit großem Verständnis und Geduld getragen haben.

Ein solches Werk wäre ohne die Unterstützung eines gut funktionierenden und anerkannten Verlages nicht möglich. Deshalb möchten wir uns herzlich beim Hanser Verlag bedanken, der uns die Gelegenheit gegeben hat, dieses Buch zu veröffentlichen. Unser besonderer Dank gilt Frau Brigitte Bauer-Schiewek und Frau Kristin Rothe für ihre kontinuierliche und hervorragende Unterstützung sowie ihr unermüdliches Engagement, das dieses Projekt maßgeblich vorangebracht hat.

Ein weiterer Dank gilt den zahlreichen Teilnehmerinnen und Teilnehmern unserer Python-Kurse, deren Fragen, Anregungen und Rückmeldungen uns geholfen haben, unsere didaktischen und fachlichen Fähigkeiten stetig weiterzuentwickeln – Fähigkeiten, die beim Verfassen dieses Buches von unschätzbarem Wert waren. Ein besonderer Dank geht zudem an die Besucherinnen und Besucher unserer Online-Tutorials auf *www.python-kurs.eu* und *www.python-course.eu*, insbesondere an diejenigen, die uns durch konstruktive Anmerkungen und wertvolles Feedback unterstützt haben.

Wir möchten auch Herrn Jürgen Dubau für das hervorragende Lektorat danken, das wesentlich zur sprachlichen Klarheit und Präzision dieses Werkes beigetragen hat. Ebenso gebührt unser Dank Herrn Stephan Korell und Frau Irene Weilhart für ihre wertvolle Unterstützung bei der Umsetzung in LaTeX. Ihre Expertise und Sorgfalt haben die technische Umsetzung dieses Buches erheblich erleichtert.

Abschließend möchten wir all jenen danken, die durch ihre kritischen Anmerkungen, ihr Lob und ihre Unterstützung dazu beigetragen haben, dass dieses Buch in seiner jetzigen Form entstehen konnte. Ihre Beiträge sind für uns von unschätzbarem Wert.

Bernd Klein, Singen
Philip Klein, Freiburg

August 2024

3 Einleitung

3.1 Einführung

Dieses Buch widmet sich einem speziellen und faszinierenden Aspekt der Programmierung mit Python: der funktionalen Programmierung. Während Python vor allem für seine Einfachheit und Vielseitigkeit bekannt ist, bietet es auch mächtige Werkzeuge und Paradigmen, die es ermöglichen, Probleme auf eine neue und elegante Weise zu lösen. Eines dieser Paradigmen ist die funktionale Programmierung.

Funktionale Programmierung ist ein Programmierparadigma, das darauf abzielt, Berechnungen als Auswertung mathematischer Funktionen zu behandeln und Zustandsveränderungen zu vermeiden. Dies steht im Gegensatz zum imperativen Paradigma, das den Schwerpunkt auf die Ausführung von Befehlen und die Manipulation von Zuständen legt. Durch die Anwendung der Prinzipien der funktionalen Programmierung können Programme oft klarer, kompakter und weniger fehleranfällig gestaltet werden.

3.2 Zielsetzung des Buches

Dieses Buch ist nicht als umfassende und systematische Einführung in die funktionale Programmierung konzipiert. Der Grund dafür liegt darin, dass Python keine rein funktionale Programmiersprache ist, sondern eine vielseitige Sprache, die sowohl imperative als auch objektorientierte Paradigmen unterstützt. Obwohl Python viele funktionale Prinzipien integriert, bietet es oft Kompromisse zugunsten dieser anderen Paradigmen. Der Schwerpunkt dieses Buches liegt daher auf den Bereichen der funktionalen Programmierung, in denen Python besonders stark ist. Dies wird durch zahlreiche praktische Anwendungen und anschauliche Beispiele verdeutlicht. Jedes Kapitel endet mit einer Reihe von Aufgaben, die das Gelernte vertiefen und erweitern. Zu allen Aufgaben werden ausführliche Lösungen bereitgestellt, um das Verständnis zu fördern und die Lernziele zu erreichen.

3.3 Aufbau des Buches

Das Buch besteht aus zwei Hauptteilen:

1. Python-Grundlagen unter funktionalen Aspekten
2. Funktionale Programmierung

Im ersten Teil des Buches befassen wir uns mit den grundlegenden Datenstrukturen von Python wie Integer, Gleitkommazahlen, Strings, Listen, Tupel und Dictionaries. Darüber hinaus behandeln wir die Kontrollstrukturen, darunter bedingte Anweisungen und Schleifen, wobei wir dabei auf die Besonderheiten im Vergleich zur funktionalen Programmierung bereits ein wenig eingehen.

Das Modul `collections` in Python ist wichtig, weil es erweiterte Datenstrukturen wie `deque`, `defaultdict`, `Counter` und `namedtuple` bietet, die über die Standard-Datenstrukturen hinausgehen und häufige Programmieraufgaben erleichtern. Diese spezialisierten Container ermöglichen effizientere und lesbarere Lösungen für komplexe Aufgabenstellungen. In der funktionalen Programmierung erleichtert das Modul `collections` die Arbeit mit unveränderlichen und strukturierten Daten, indem es z. B. `namedtuple` bereitstellt, um benannte und unveränderliche Datentypen zu erstellen. Außerdem bietet `Counter` eine praktische Möglichkeit, Daten zu aggregieren und zu zählen, was funktionale Techniken wie Map- und Reduce-Operationen unterstützen kann.

Da die funktionale Programmierung bei all diesen Themen nicht im Vordergrund steht, haben wir sie in diesem ersten Teil des Buches zusammengefasst. Dieser Abschnitt richtet sich insbesondere an Personen, die bereits über Programmiererfahrung verfügen, jedoch noch keine Vorkenntnisse in Python besitzen.

Im zweiten Teil geht es dann um die funktionalen Aspekte. Zunächst bemühen wir uns um eine Begriffsbestimmung des funktionalen Programmierstils und einer Abgrenzung zu anderen Programmierparadigmen in Kapitel 7 (Begriffsbestimmung). Das folgende Kapitel 8 (Funktionen) widmet sich ganz den Funktionen: beginnend bei einfachen Funktionen, lambda-Funktionen und rekursiven Funktionen bis hin zum Prinzip der First-Class-Funktionen. Dann sind wir bereit für das Thema Dekoratoren in Kapitel 9 (Dekoratoren). Dekoratorfunktionen in Python ermöglichen es, das Verhalten von Funktionen oder Methoden modular und wiederverwendbar zu erweitern oder zu modifizieren, ohne den ursprünglichen Code zu verändern. In diesem Kapitel finden sich viele interessante Anwendungen für Dekoratoren in Form von Beispielen und Aufgaben. Einer der faszinierendsten Anwendungen, nämlich der Memoisation, ist ein eigenes Kapitel 10 (Memoisation) gewidmet.

Die Komposition von Funktionen ist in der funktionalen Programmierung von großer Bedeutung, da sie hilft, sauberen, wartbaren und flexiblen Code zu schreiben. Sie fördert die Modularität, Lesbarkeit und Wiederverwendbarkeit von Funktionen, unterstützt die Vermeidung von Seiteneffekten und ermöglicht eine höhere Abstraktionsebene, die die Entwicklung komplexer Systeme vereinfacht. Dieses Thema behandeln wir deshalb auch in einem eigenen Kapitel 12 (Komposition von Funktionen).

Mit dem Thema „Currying“ beschäftigen wir uns in Kapitel 13 (Currying in Python). Dabei handelt es sich um ein zentrales Konzept der funktionalen Programmierung, das die

Umwandlung von Funktionen mit mehreren Argumenten in eine Serie von Funktionen mit jeweils einem Argument ermöglicht. Dies verbessert die Modularität und Wiederverwendbarkeit des Codes. In Python unterstützt Currying elegante und flexible Code-Strukturen, die den Prinzipien der funktionalen Programmierung entsprechen.

Funktionale Programmierung und objektorientierte Programmierung sind mächtige, aber unterschiedliche Paradigmen. In Kapitel 14 (Funktionale Emulation von OOP-Konzepten) zeigen wir, dass es sich nicht um unvereinbare oder gegensätzliche Paradigmen handelt. Wir zeigen, wie wir Klassendefinitionen in der funktionalen Programmierung durch First-Class-Funktionen unter Wahrung der objektorientierten Ziele realisieren können. Zudem demonstrieren wir, dass sich selbst objektorientierte Konzepte wie Vererbung funktional emulieren lassen.

Iteratoren und Generatoren sind in Python von großer Bedeutung, da sie eine effiziente Speicherverwaltung ermöglichen. Anstatt Daten vollständig auf einmal zu laden, erzeugen und verarbeiten sie diese bei Bedarf. Dies führt zu einer verbesserten Leistung und Skalierbarkeit von Programmen, insbesondere bei der Verarbeitung großer Datenmengen oder endloser Sequenzen. Deshalb haben wir dieser Thematik gleich drei Kapitel gewidmet. In Kapitel 15 (Generatoren und Iteratoren) zeigen wir, wie man mit Generatorfunktionen Iteratoren erzeugt, die Werte schrittweise mit dem Schlüsselwort `yield` zurückgeben, wodurch sie speichereffizient große Datenmengen oder unendliche Sequenzen verarbeiten können. In Kapitel 16 (Iteratoren der Standardbibliothek) lernen wir dann wichtige Iteratoren wie `zip`, `enumerate`, `map`, `filter`, `reduce`, der Standardbibliothek kennen. In den Beispielen demonstrieren wir, wie man mit ihrer Hilfe effizienteren Code schreiben kann. Das effizientere Laufzeitverhalten bei der Verwendung von Iteratoren und Generatoren kann allgemein dadurch erklärt werden, dass sie Daten „on-the-fly" erzeugen, anstatt ganze Listen oder andere Datenstrukturen im Speicher zu halten.

Besonders intensiv gehen wir in Kapitel 17 (Das Modul itertools) auf das Modul `itertools` ein, da es im Zusammenhang mit funktionaler Programmierung von besonderer Wichtigkeit ist, weil es eine Sammlung von effizienten, speichersparenden Iteratoren zur Verfügung stellt, die sich ideal für funktionale Programmierparadigmen eignen. Diese Werkzeuge ermöglichen es, komplexe Iterationen und Kombinationen von Daten in einem deklarativen Stil auszudrücken, ohne explizit Schleifen oder temporäre Datenstrukturen zu verwenden. Dies führt zu sauberem, lesbarem und wartbarem Code, der die Prinzipien der funktionalen Programmierung wie Unveränderlichkeit und First-Class-Funktionen unterstützt.

Die `itertools`-Bibliothek in Python bietet vielseitige Funktionen für die Arbeit mit Iterationen. Zu den wichtigsten gehören unendliche Iteratoren wie `count`, `cycle` und `repeat`, die unendlich lange Sequenzen erzeugen. Endliche Kombinations- und Permutations-Iteratoren wie `chain`, `product`, `permutations` und `combinations` ermöglichen die Erzeugung komplexer Kombinationen und Anordnungen von Daten. Zudem bietet `itertools` leistungsfähige Filter- und Mapping-Tools wie `starmap`, `takewhile`, `dropwhile` und `accumulate`, die es ermöglichen, Sequenzen nach bestimmten Kriterien zu filtern und zu transformieren. Insgesamt erleichtert `itertools` die effiziente und speicherschonende Verarbeitung von Daten.

3.4 Leserschaft

Auch wenn wir im Buch extra einen Abschnitt zu den Grundlagen von Python aufgenommen haben, wäre es dennoch hilfreich, bereits etwas Programmiererfahrung mit Python zu besitzen. Ansonsten richtet sich dieses Buch an Personen mit Programmiererfahrung in Sprachen wie C, C++, C#, JavaScript oder Java, die die funktionale Denkweise und die Programmierung in Python erlernen möchten. Es vermittelt sowohl das notwendige theoretische Wissen als auch praktische Fähigkeiten, um die Prinzipien der funktionalen Programmierung erfolgreich in der Praxis anzuwenden.

Letztlich ist es jedoch immer eine individuelle Entscheidung, ob ein Buch für jemanden geeignet ist oder nicht. Kein Buch kann für jeden „ideal“ sein. Wir sind jedoch überzeugt, dass dieses Buch für viele Programmierende und Studierende neue Perspektiven eröffnen und ihren Programmierhorizont erheblich erweitern kann.

3.5 Zusätzliche Unterlagen

Auf unserer Webseite *https://python-kurs.eu/funkional* stellen wir die Beispiele und Lösungen als Jupyter-Notebooks zur Verfügung. Denn was nutzt das beste Buch, wenn man den Code der Programme nicht zur Verfügung hat und mühsam eintippen muss, um Beispiele und Aufgaben nachvollziehen zu können? Deshalb haben wir uns entschieden, Jupyter-Notebooks mit Programmcode online zur Verfügung zu stellen.

TEIL II

Python-Grundlagen unter funktionalen Aspekten

4 Variablen und Datentypen

Auch erfahrenen Programmierenden anderer Programmiersprachen empfehlen wir, dieses Kapitel keinesfalls zu überspringen. Selbst bei Python-Programmierenden gibt es leider immer wieder falsche Vorstellungen bezüglich der Implementierung von Variablen in Python, was dann in manchen Kontexten immer wieder zu Verwirrungen, Verständnisproblemen und in vielen Fällen sogar Fehlern führt.

4.1 Variablen

Es bestehen signifikante Unterschiede zwischen Python und anderen Programmiersprachen hinsichtlich der Behandlung von Variablen. In Python sind Variablen lediglich und ohne Ausnahme Referenzen zu Objekten und keine festen Speicherplätze wie in vielen anderen Programmiersprachen. Eine Variable in Python ist ein Bezeichner, der auf ein Objekt verweist. Anders als in den meisten Programmiersprachen ist weder ein Speicherplatz noch ein Datentyp einer Variablen fest zugewiesen. Variablen ist also kein Datentyp zugeordnet. Sie können während des Programmlaufes verschiedene Objekte mit verschiedenen Datentypen referenzieren.

Schauen wir uns dies an einem konkreten Beispiel an. Im Folgenden weisen wir einer Variablen x einen Integer-Wert 42 zu. Genau genommen müsste man sagen, dass wir ein Integer-Objekt „42“ erzeugen und es mit dem Namen x referenzieren.

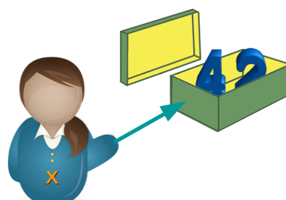

Bild 4.1 Variable, eine Referenz auf ein Objekt

Besonders interessant wird es, wenn wir nun die Zuweisung y = x ausführen. Wie in Bild 4.2 gezeigt, referenzieren beide Variablen nun dasselbe Objekt.

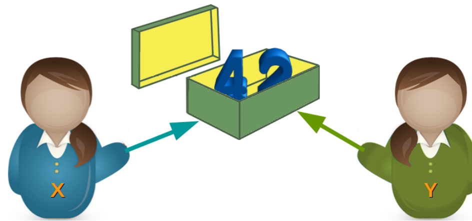

Bild 4.2 Zwei Variablen referenzieren dasselbe Objekt.

Wir können dies mithilfe der Identitätsfunktion `id` beweisen. Dies ist eine Funktion, die für eine Variable die Identität des referenzierten Objektes zurückgibt. Die Identität ist ein eindeutiger Bezeichner für ein Objekt, die in den meisten Python-Implementierungen der Speicheradresse des Objektes entspricht. Erhält man für zwei Variablennamen die gleiche Identität, so weiß man, dass sie das gleiche Objekt referenzieren.

```
x = 42
y = x
print(id(x) == id(y))
```

Ausgabe:

```
True
```

Im folgenden Code benutzen wir eine Variable `i`. Auch in der Zuweisung `i = i + 1` ebenso wie `i += 1` wird immer ein neues Integerobjekt erzeugt, da Integerobjekte unveränderlich sind. Unveränderliche Objekte sind natürlich im Sinne der funktionalen Programmierung. Aus der Sicht der Variablenreferenz: Zunächst referenziert `i` ein Integerobjekt welches 42 enthält. Bei der Ausführung des Codes `i = i + 1` wird ein neues Integerobjekt 43 erzeugt, und `i` zeigt dann auf dieses. Das alte Objekt 42 wird automatisch entsorgt, wenn es von keiner anderen Variablen referenziert wird.

```
i = 42
print(f'{id(i)=}')
i = i + 1
print(f'{id(i)=}')
i += 1
print(f'{id(i)=}')
```

An den drei verschiedenen Werten für die `id`-Funktion in der Ausgabe können wir erkennen, dass jeweils ein neues Objekt erzeugt worden ist.

Ausgabe:

```
id(i)=140633329536592
id(i)=140633329536624
id(i)=140633329536656
```

Wichtige funktionale Aspekte: Beim Einsatz von Referenzen in funktionalen Programmiersprachen besteht die Gefahr unerwarteter Seiteneffekte, insbesondere wenn veränderliche Zustände (mutable states) über diese Referenzen manipuliert werden. Dies widerspricht dem Prinzip der Unveränderlichkeit (immutability) und kann die Verständlichkeit sowie die Wartbarkeit des Codes beeinträchtigen. Diese Problematik betrifft natürlich auch andere, nicht-funktionale Programmiersprachen.

4.1.1 Gültige Variablennamen

Variablennamen müssen mit einem Buchstaben oder Unterstrich „_" beginnen. Die folgenden Zeichen dürfen sich aus einer beliebigen Folge von Buchstaben, Ziffern und dem Unterstrich zusammensetzen. Variablennamen sind case sensitive.[1] Dies bedeutet, dass Python zwischen Groß- und Kleinschreibung unterscheidet:

```
# Gültige Variablennamen
name = "Max"
alter = 25
ist_student = True
durchschnitts_note = 2.3
liste_von_zahlen = [1, 2, 3, 4, 5]
meine_funktion = lambda x: x * 2
MAX_WERT = 1000
gesamt_verkauf_2023 = 500000.0
```

Zu den gültigen Buchstaben gehören aber nicht nur „lateinische" Buchstaben, sondern auch alle anderen Unicode-Buchstaben wie beispielsweise kyrillische, griechische usw.

4.1.2 Konventionen für Variablennamen

Man bevorzugt in der Python-Community Kleinschreibung bei Variablennamen, z. B. `minimum` statt `Minimum`. Außerdem werden Variablennamen, die aus mehreren Wörtern bestehen, mittels Unterstrich (_) separiert, also beispielsweise `minimale_breite`. Variablennamen mit Binnenversalien – bei Programmierenden besser als „camel case" oder „CamelCase" bekannt – sind in der Python-Welt nicht beliebt: `MinimaleBreite` oder `minimaleBreite`.

Ansonsten zeichnet sich ein guter Programmierstil dadurch aus, dass man möglichst sprechende Variablennamen verwendet. Also beispielsweise `aktueller_kontostand` oder `maximale_beschleunigung` statt nichtssagender Namen wie `ak` oder `mb`.

Wie wir in den vorigen Beispielen gesehen haben sind Typdeklaration nicht erforderlich. Sie sind sogar nicht möglich. Python kennt keine Typdeklarationen.[2]

Auch wenn den Variablennamen keine Typen zugeordnet sind, so entspricht jedes in Python definierte Objekt einem Typ oder genauer gesagt der Instanz einer Klasse.

---

[1] Für diesen Begriff gibt es keine schöne deutsche Übersetzung, im Prinzip müssten wir sagen „groß-/kleinschreibungsabhängig".

[2] Typehints und Type Annotations sind zwar ein ähnliches Konzept, aber sie werden von Python wie Kommentare betrachtet.

4.2 Übersicht Datenstrukturen

In Python kann man die Grunddatentypen in unveränderliche (engl. immutable) und veränderliche (engl. mutable) untergliedern. Dies ist besonders im Hinblick auf funktionale Programmierung wichtig. In der funktionalen Programmierung werden unveränderliche Datentypen oft bevorzugt, da sie viele Vorteile bieten wie z. B. eine einfachere Handhabung von parallelen Threads, eine bessere Referenztransparenz und einfachere Debugging- und Testmöglichkeiten. Veränderliche Datentypen können jedoch in bestimmten Situationen effizienter sein, insbesondere wenn große Datenmengen manipuliert werden müssen. Es ist wichtig, die Vor- und Nachteile jeder Art von Datenstruktur zu verstehen und sie entsprechend den Anforderungen und Zielen des Programms auszuwählen.

4.2.1 Unveränderliche (immutable) Datentypen

1. Integer
2. Float
3. String
4. Bytesequenz

4.2.2 Veränderliche (mutable) Datentypen

1. **Listen (`list`):**
   - Eine Liste repräsentiert eine geordnete Sammlung von Elementen.
   - Beispiele: `[1, 2, 3]`, `['a', 'b', 'c']`, `[]`.
   - Veränderlich: Die Elemente einer Liste können nach ihrer Erstellung geändert, hinzugefügt oder entfernt werden.
2. **Dictionary (`dict`):**
   - Ein Dictionary repräsentiert eine Sammlung von Schlüssel-Wert-Paaren.
   - Beispiele: `{'name': 'John', 'age': 30}`, `{1: 'one', 2: 'two'}`, `{}`.
   - Veränderlich: Die Schlüssel und Werte eines Dictionaries können nach ihrer Erstellung geändert, hinzugefügt oder entfernt werden.
3. **Mengen (`set`):**
   - Eine Menge repräsentiert eine Sammlung eindeutiger Elemente.
   - Beispiele: `{1, 2, 3}`, `{'a', 'b', 'c'}`, `set()`.
   - Veränderlich: Die Elemente eines Sets können nach ihrer Erstellung hinzugefügt oder entfernt werden.

4.3 Datenstrukturen im Detail

4.3.1 Integer

Integers sind in der Mathematik als ganze Zahlen bekannt:

```
..., -3, -2, -1, 0, 1, 2, 3, ...
```

Das heißt, die Zahlen reichen von minus unendlich bis unendlich. Selbstverständlich können wir „unendlich“ in „int“ nicht darstellen, aber die Zahlen in Python können extrem groß bzw. extrem klein werden.

In Python sind Ganzzahlen (Integers) nicht durch eine feste Anzahl von Bits begrenzt, wie es beispielsweise bei anderen Programmiersprachen der Fall ist. Stattdessen kann eine Ganzzahl in Python so groß sein, wie es der verfügbare Speicherplatz im System erlaubt.

Das bedeutet, dass Python-Ganzzahlen dynamisch an die Größe des benötigten Speicherplatzes angepasst werden können. Daher kann man in Python extrem große Zahlen ohne explizite Deklaration von Datentypen verwenden. Dies macht Python besonders flexibel und einfach zu verwenden, wenn es um numerische Berechnungen mit großen Zahlen geht.

```python
x = 42 ** 42
print(x)
```

Dies liefert folgende Zahl:

```
150130937545296572356771972164254457814047970568738777235893533016064
```

4.3.2 Floats

Floats in Python sind Datentypen, die verwendet werden, um Gleitkommazahlen darzustellen. Sie ermöglichen die Darstellung von Dezimalzahlen sowie sehr großer oder sehr kleiner Zahlen mit hoher Genauigkeit.

Hier ist ein einfaches Beispiel, wie Floats in Python verwendet werden können:

```python
x = 3.141592
y = 1.23e-5
z = 2.5e6
```

In diesem Beispiel werden die Variablen `x`, `y` und `z` als Floats initialisiert. `x` repräsentiert die Dezimalzahl `3.141592`, `y`, repräsentiert die wissenschaftliche Notation `1.23e-5` (entspricht 1.23×10^{-5}), und `z` repräsentiert die Zahl `2.5e6` (entspricht 2.5×10^{6}).

Floats bieten eine hohe Genauigkeit, aber aufgrund von Rundungsfehlern können sie manchmal unerwartetes Verhalten aufweisen, insbesondere bei Berechnungen mit sehr kleinen oder sehr großen Zahlen.

4.3.3 Zeichenketten oder Strings

Ein weiterer wichtiger Datentyp in Python sind die Zeichenketten, die man meist auch als Strings bezeichnet. Strings können auf verschiedene Arten definiert werden: mit einfachen Anführungszeichen, mit doppelten Anführungszeichen oder mit drei einfachen bzw. drei doppelten Anführungszeichen. Wir demonstrieren diese Varianten im folgenden Beispiel. Wir verwenden auch das Nummernzeichen „#“, um Kommentare einzuleiten:

```
# Strings mit einfachen Anführungszeichen
string1 = 'Hallo Welt!'

# Strings mit doppelten Anführungszeichen
string2 = "Python ist großartig!"

# Strings mit drei einfachen Anführungszeichen für mehrzeilige Strings
string3 = '''Um wirklich zu sagen
wie toll Python ist,
langt meistens nicht eine einzige Zeile'''

# Strings mit drei doppelten Anführungszeichen für mehrzeilige Strings
string4 = """Python
ist eine
wunderbare
Programmiersprache"""

# Ausgabe der Strings
print(string1)
print(string2)
print(string3)
print(string4)
```

Dieser Code liefert folgende Ausgabe:

```
Hallo Welt!
Python ist großartig!
Um wirklich zu sagen
wie toll Python ist,
langt meistens nicht eine einzige Zeile
Python
ist eine
wunderbare
Programmiersprache
```

Strings können indiziert werden, dabei entspricht das erste Zeichen dem Index 0. Das letzte Zeichen eines Strings können wir mit dem Index -1 ansprechen, d. h. mit negativen Zahlen erhält man eine Indizierung von rechts:

```
text = 'Python'
print(f'1. Zeichen: text[0]=')
print(f'2. Zeichen: text[1]=')
print(f'Letztes Zeichen: text[-1]=')
```

Ausgabe

```
1. Zeichen: text[0]=
2. Zeichen: text[1]=
Letztes Zeichen: text[-1]=
```

Während man mit dem Indizieren nur einzelne Zeichen, also Strings der Länge 1, aus einem String erhalten kann, ermöglicht der Teilbereichsoperator (Slicing) das „Herausschneiden" von beliebigen Teilstrings:

```
text = "Ein grüner Fisch an der Wand singt nie schräg!"
# Teil des Strings von Index 7 bis Index 10 (exklusive):
teil_string1 = text[7:10]
print(teil_string1)

# Teil des Strings von Index 7 bis zum Ende
teil_string2 = text[7:]
print(teil_string2)

# Teil des Strings vom Anfang bis Index 6 (exklusiv)
teil_string3 = text[:6]
print(teil_string3)

# Negativer Index: Teil des Strings vom vorletzten Zeichen bis zum
↪  Ende
teil_string4 = text[-2:]
print(teil_string4)
```

Ausgabe

```
ner
ner Fisch an der Wand singt nie schräg!
Ein gr
g!
```

Finden von Teilstrings in Strings:

```
s = "A horse, a horse! "
s += "My kingdom for a horse!"
print(f'{s.find("horse")=}')
print(f'{s.find("horse", 3)=}')
print(f'{s.find("horse", 16)=}')
print(f'{s.find("horse", 36)=}')
print(f'{s.rfind("horse")=}')   # Suche beginnt von hinten
```

Ausgabe:

```
s.find("horse")=2
s.find("horse", 3)=11
s.find("horse", 16)=35
s.find("horse", 36)=-1
s.rfind("horse")=35
```

Es gibt noch zahlreiche Stringmethoden, die es verdienten aufgeführt zu werden, dies würde aber den Rahmen des Buches sprengen. Außerdem sind sie im Großen und Ganzen sehr leicht zu verstehen. Eine Übersicht der verfügbaren Methoden, kann man sich mittels `help(str)` verschaffen.

Wichtig ist vor allen Dingen im Hinblick auf die funktionale Programmierung, wie wir bereits geschrieben haben, die Unveränderlichkeit von Strings. Diese Eigenschaft ermöglicht eine effiziente Speicherung und Verarbeitung von Zeichenfolgen in Python und fördert zugleich die Sicherheit und Vorhersehbarkeit von Programmen.

4.3.4 Bytesequenz

Eine *Bytesequenz* (englisch: *byte sequence*) wird zur Darstellung von Daten im Binärformat verwendet wird. Im Gegensatz zu regulären Zeichenketten, die aus lesbaren Zeichen bestehen, enthalten binäre Zeichenketten rohe Byte-Daten, die nicht unbedingt als Text interpretiert werden können.

In Python werden Bytesequenzen mit einem Präfix `b` vor der Zeichenkette dargestellt, z. B. `b'A'`, was äquivalent zu `b'\x41'` ist. Jede Zeichenkette, die mit `b'` beginnt, wird als eine Sequenz von Bytes interpretiert. Diese Bytesequenzen sind nützlich für die Verarbeitung von Daten auf niedriger Ebene wie z. B. beim Umgang mit Netzwerkprotokollen, kryptografischen Operationen oder Binärdateien.

Ein binäres Zeichen kann Werte von 0 bis 255 (8 Bit) annehmen. Da es sich um Rohdaten handelt, können binäre Zeichenketten keine speziellen Zeichen oder Unicode-Zeichen direkt enthalten, es sei denn, sie werden korrekt codiert.

```
unicode_string = 'Grüße aus Österreich, wo Bäume und Flüsse fröhlich
 ↪  fließen!'

# Konvertierung zu einer Binärzeichenkette
binary_string = unicode_string.encode('utf-8')

# Rückkonvertierung zu einer Unicode-Zeichenkette
decoded_string = binary_string.decode('utf-8')

# Ergebnisse anzeigen
print("Originale Unicode-Zeichenkette:\n", unicode_string)
print("Binärzeichenkette:\n", binary_string)
print("Decodierte Zeichenkette:\n", decoded_string)
```

Die Ausgabe sieht wie folgt aus:

```
Originale Unicode-Zeichenkette:
 Grüße aus Österreich, wo Bäume und Flüsse fröhlich fließen!
Binärzeichenkette:
 b'Gr\xc3\xbc\xc3\x9fe aus \xc3\x96sterreich, wo B\xc3\xa4ume und
  ↪  Fl\xc3\xbcsse fr\xc3\xb6hlich flie\xc3\x9fen!'
Decodierte Zeichenkette:
 Grüße aus Österreich, wo Bäume und Flüsse fröhlich fließen!
```

4.3.5 Listen

Listen werden mittels eckiger Klammern in Python erzeugt. Eine Liste kann beliebige Python-Objekte enthalten, die durch Komma getrennt sind:

```
lst = ["rot", "grün", "blau"]
lst2 = ["rot", 12, [3, 6.78]]
```

Auf Listenelemente kann man wie bei Strings über Indices zugreifen oder man greift auf mehrere Listenelemente mit dem Teilbereichsoperator zu. Außerdem können wir mittels Indizierung der Liste auch neue Werte zuweisen:

```
lst = ["rot", 12, "gelb", 123, [3, "Noch ein String"]]
print(f'{lst[0]=}')
print(f'{lst[4]=}')
print(f'{lst[4][1]=}')
print(f'{lst[1:4]=}')
lst[0] = "orange"
print(f'{lst=}')
```

Ausgabe:

```
lst[0]='rot'
lst[4]=[3, 'Noch ein String']
lst[4][1]='Noch ein String'
lst[1:4]=[12, 'gelb', 123]
lst=['orange', 12, 'gelb', 123, [3, 'Noch ein String']]
```

Weitere interessante Funktionalitäten des `list`-Datentyps zeigen wir in folgendem selbsterklärenden Beispiel:

```
einkaufsliste = ["Milch", "Eier", "Brot"]

# Ein Element am Ende hinzufügen, wenn es noch nicht in der Liste ist
neues_element = "Butter"
if neues_element not in einkaufsliste:
    einkaufsliste.append(neues_element)

print("Liste nach dem Append:", einkaufsliste)
# Entfernen des letzten Elements mit pop
letztes_element = einkaufsliste.pop()
print("Entferntes Element:", letztes_element)
print("Liste nach dem Pop:", einkaufsliste)

# Einfügen eines Elements an Index 1 mit insert, falls nicht in Liste
neues_element = "Käse"
if neues_element not in einkaufsliste:
    einkaufsliste.insert(1, neues_element)

print("Liste nach dem Insert:", einkaufsliste)

# Entfernen des Elements "Eier" mit remove
element_zum_entfernen = "Eier"
```

```
if element_zum_entfernen in einkaufsliste:
    einkaufsliste.remove(element_zum_entfernen)

print("Liste nach dem Remove:", einkaufsliste)

# Konkatenation von Listen
weitere_einkaeufe = ["Mehl", "Zucker"]
einkaufsliste += weitere_einkaeufe
print("Liste nach der Konkatenation:", einkaufsliste)
# Kopieren einer Liste
kopie_einkaufsliste = einkaufsliste.copy()
print("Kopie der Einkaufsliste:", kopie_einkaufsliste)
```

Ausgabe des Listenbeispiels:

```
Liste nach dem Append: ['Milch', 'Eier', 'Brot', 'Butter']
Entferntes Element: Butter
Liste nach dem Pop: ['Milch', 'Eier', 'Brot']
Liste nach dem Insert: ['Milch', 'Käse', 'Eier', 'Brot']
Liste nach dem Remove: ['Milch', 'Käse', 'Brot']
Liste nach der Konkatenation: ['Milch', 'Käse', 'Brot', 'Mehl', 'Zucker']
Kopie der Einkaufsliste: ['Milch', 'Käse', 'Brot', 'Mehl', 'Zucker']
```

4.3.6 Mengen

Mengen, in Python als `set` bezeichnet, sind eine Sammlung von einzigartigen und unveränderlichen Elementen. Hier sind einige wichtige Eigenschaften und Operationen von Mengen in Python:

- **Einzigartigkeit von Elementen**: In einer Menge kann jedes Element nur einmal vorkommen. Wenn man versucht, ein bereits vorhandenes Element hinzuzufügen, wird die Menge nicht verändert, es wird also nicht nochmals hinzugefügt.
- **Veränderlichkeit**: Nachdem eine Menge erstellt wurde, können Elemente hinzugefügt oder entfernt werden.
- **Ungeordnet**: Die Elemente in einer Menge werden nicht in einer bestimmten Reihenfolge gespeichert. Daher gibt es keine Indexierung oder Sortierung in Mengen.
- **Effiziente Operationen**: Mengen bieten effiziente Operationen wie das Hinzufügen von Elementen, das Entfernen von Elementen und die Überprüfung der Mitgliedschaft (überprüfen, ob ein Element in der Menge enthalten ist).

Hier ist ein einfaches Beispiel, wie Mengen in Python verwendet werden können:

```
meine_menge = {1, 2, 3, 4, 5}

# Hinzufügen eines Elements
meine_menge.add(6)

# Entfernen eines Elements
meine_menge.remove(3)
```